simuPOP User's Guide

Release 0.8.8 (Rev: 2025)

Bo Peng

December 2004

Last modified
December 3, 2008

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

Abstract

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

simuPOP is 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 desciprion 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?

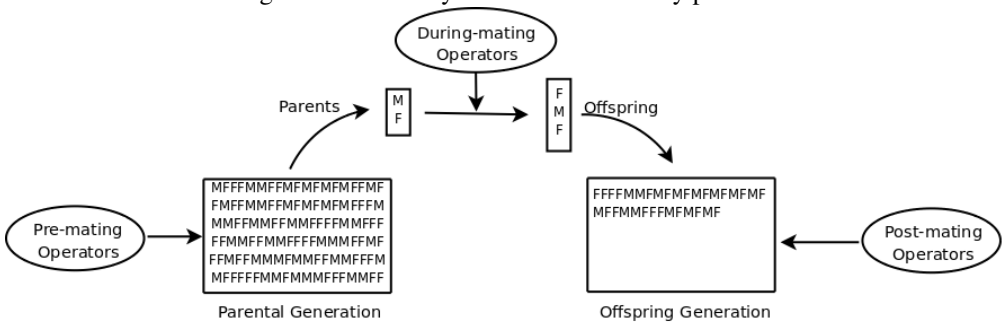
simuPOP is an individual-based forward-time population genetics simulation environment based on Python, a dynamic object-oriented programming language that has been widely used in biological studies. simuPOP provides a large number of Python objects and functions, and a mechanism to evolve populations 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.

Figure 1.1: A life cycle of an evolutionary process



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

Example 1.1: A simple example

```
>>> from simuPOP import *
>>> pop = population(size=1000, loci=[2])
>>> simu = simulator(pop, randomMating(), rep=3)
>>> simu.evolve(
...     preOps = [initByValue([1, 2, 2, 1])],
...     ops = [
...         recombinator(rate=0.01),
...         stat(LD=[0, 1]),
...         pyEval(r"'%.2f\t' % LD[0][1]", step=10),
...         pyOutput('\n', rep=-1, step=10)
...     ],
...     gen=100
... )
0.24    0.25    0.24
0.22    0.23    0.22
0.21    0.20    0.20
0.19    0.18    0.18
0.16    0.17    0.17
0.14    0.15    0.17
0.12    0.12    0.14
0.12    0.12    0.13
0.11    0.10    0.14
0.10    0.10    0.14
(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 disequi-

librium 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 interoperate 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. Several functions, such as the graphical version of the `ListVars()` function, are only available for wxPython.

1.5 Getting help

1.5.1 Online help system

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

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

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

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

        x.addInfoField(field, init=0)

    Details:

        Add an information field field to a population and initialize its
```

```
        values to init.

>>>
```

It is important that you understand that

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

  ```
  >>> help(population.__init__)
  ```

- Some classes are derived from other classes and have access to member functions of their base classes. For example, class `population`, `individual` and `simulator` are all 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 writting 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 Naming Conventions

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 (objects, 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 as well. They are usually prefixed with a category name. For example, `MigrByProportion` specifies a migration mode.

Finally, simuPOP uses the abbreviated forms of the following words in function and parameter names:

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

Chapter 2

simuPOP components

2.1 simuPOP modules

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

- Core simuPOP modules (`simuPOP_XXX.py`, `_simuPOP_XXX.pyd`) and a number of utility modules (`simuUtil.py, simuOpt.py` etc) under `c:\python2X\Lib\site-packages`.
- `c:\python2X\share\simuPOP\doc`: This directory contains the pdf version of this user's guide and the *simuPOP reference manual*.
- `c:\python2X\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.
- `c:\python2X\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.

There are six flavors of the core simuPOP module: short, long and binary allele modules, and their optimized versions. The short allele modules use 8 bits to store each allele which limits the possible allele states to 256. This is enough most of the times but not so if you need to simulate models such as the infinite allele model. In those cases, you should use the long allele version of the modules, which use 16 bits for each allele and can have 2^{16} possible allele states. On the other hand, if you would like to simulate a large number of binary (SNP) markers, binary libraries can save you a lot of RAM because they use 1 bit for each allele. Despite of differences in internal memory layout, all these modules have the same interface.

Standard libraries have detailed debug and run-time validation mechanism to make sure a simulation executes correctly. Whenever something unusual is detected, simuPOP would terminate with detailed error messages. The cost of such run-time validation varies from case to case but can be high under some extreme circumstances. Because of this, optimized versions for all modules are provided. They bypass all parameter checking and run-time validations and will simply crash if things go wrong. It is recommended that you use standard libraries whenever possible and only use the optimized version when performance is needed and you are confident that your simulation is running as expected.

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

Example 2.1: Use of standard simuPOP modules

```
>>> from simuPOP import *
simuPOP : Copyright (c) 2004-2008 Bo Peng
Version snapshot (Revision 9999, Dec  2 2008) for Python 2.4.3
[GCC 4.1.2 20071124 (Red Hat 4.1.2-42)]
Random Number Generator is set to mt19937 with random seed 0x7e45d314e7252911
This is the standard short allele version with 256 maximum allelic states.
For more information, please visit http://simupop.sourceforge.net,
or email simupop-list@lists.sourceforge.net (subscription required).
>>> pop = population(10, loci=[2])
>>> pop.locusPos(10)
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
IndexError: src/genoStru.h:552 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 ?
    #
IndexError: src/population.h:417 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 supress 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.

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 gurantees 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. `AvailableRNG()`, `SetRNG(name, seed)`). It is also possible to save the random seed of a simuPOP session (c.f. `rng().seed()`) and use it to replay the session later.

2.2 Pythonic issues

2.2.1 References and the `clone()` member function

Assignment in Python only creates a new refernce to an exsting 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=[20, 30], loci=[5, 6])
>>> print pop.chromLocusPair(7)
(1, 2)
>>> print pop.absLocusIndex(1, 1)
6
>>> print pop.absIndIndex(10, 1)
30
```

```
>>> print pop.subPopIndPair(40)
(1, 20)
>>>
```

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 2.7 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
...
1 0 0 1 1 1
2 0 2 1 0 1
>>>
```

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 2.7 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, 0, 1, 0, 2, 2, 1, 1, 1, 1, 2, 0, 2]
>>> print geno
[1, 1, 0, 1, 0, 2, 2, 1, 1, 1, 1, 2, 0, 2]
```

```
>>> arr.count(1)            # count
7
>>> arr.index(2)            # index
5
>>> ind.setAllele(5, 3)     # change underlying genotype using setAllele
>>> print arr               # arr is change
[1, 1, 0, 5, 0, 2, 2, 1, 1, 1, 1, 2, 0, 2]
>>> print geno              # but not geno
[1, 1, 0, 1, 0, 2, 2, 1, 1, 1, 1, 2, 0, 2]
>>> arr[2:5] = 4            # can use regular Python slice operation
>>> print ind.genotype()
[1, 1, 4, 4, 4, 2, 2, 1, 1, 1, 1, 2, 0, 2]
>>>
```

2.3 Genotypic structure

Genotypic structure refers to stuctural 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 auxillary information attached to each individual (c.f. `infoField(idx)`, `infoFields()`). In addition to property access functions, a number of utility functions are provided to, for example, look up the index of a locus by its name (c.f. `locusByName()`, `chromBegin()`, `chromLocusPair()`).

A genotypic structure can be retrieved from *individual, population* and *simulator* objects. Because a population consists of individuals of the same type, and a simulator consists of populations of the same type, genotypic information can only be changed for all individuals at the population level, or for all populations at the simulator level. Example 2.7 demonstrates how to access genotypic structure functions at the population and individual levels.

Example 2.7: Genotypic structure functions

```
>>> pop = population(size=[2, 3], ploidy=2, loci=[5, 10],
...     lociPos=[range(0, 5), range(0, 20, 2)], chromNames=['Chr1', 'Chr2'],
...     alleleNames=['A', 'C', 'T', 'G'])
>>> # access genotypic information from the population
>>> pop.ploidy()
2
>>> pop.ploidyName()
'diploid'
>>> pop.numChrom()
2
>>> pop.locusPos(2)
2.0
>>> pop.alleleName(1)
'C'
>>> # access from an individual
>>> ind = pop.individual(2)
>>> ind.numLoci(1)
10
>>> ind.chromName(0)
'Chr1'
>>> ind.locusName(1)
'loc1-2'
>>> # utility functions
```

```
>>> ind.chromBegin(1)
5
>>> ind.chromByName('Chr2')
1
>>>
```

2.3.1 Haploid, diploid and haplodiploid populations

simuPOP supports

2.3.2 Sex chromosomes

Currently, only `randomMating()` in diploid population supports sex chromosomes. When `sexChrom()` is `False`, the sex of an offspring is determined randomly with probability $1/2$. Otherwise, it is determined by the existence of Y chromosome, I.e., what kind of sex chromosome an offspring get from his father.

Recombinations on sex chromosomes of females (`XX`) are just like those on autosomes. However, this is not true in males. Currently, recombinations between male sex chromosomes (`XY`) are *not* allowed (a bug/feature of recombinators). This may change later if exchanges of genes between pseudoautosomal regions of `XY` need to be modeled.

Information fields refer to float numbers attached to each individual, such as fitness value, parent index, age. They are used to store auxiliary information of individuals, and are essential to the operations of some simuPOP components. For example, `'fitness'` field is required by all selectors. Details please refer to section 2.4.5.

- it is assumed that males have XY and females have XX chromosomes. The sex chromosomes of male individuals are in the order of XY.

A population can be haplodiploid (Females with two sets of chromosomes, and males with one set of chromosomes) if you specify `ploidy=Haplodiploid` when a population is created. Such a population actually store two copies of chromosomes for both male and female individuals. The difference between a diploid and a haplodiploid population is that some operators, such as a recombinator, will recognize a haplodiploid population and act acoordingly.

Individuals in the same population share the same genotypic structure. Consequently, *the genotypic information can be accessed from individual, population and simulator levels*.

2.3.3 Information fields

2.4 Population

`population` objects are essential to simuPOP. They are composed of subpopulations each with certain number of individuals, all have the same genotypic structure. A population can store arbitrary number of ancestral populations to facilitate pedigree analysis.

simuPOP uses one-level population structure, but aribitrary temprary subpopulation structure can be defined. Such temporary subpopulations are called *virtual subpopulations*, where individuals can be grouped by sex, affection status, genotype, or values of information fields. Mating is within subpopulations only. Exchange of genetic information across subpopulations can only be done through migration. Population and subpopulation sizes can be changed, as a result of mating or migration.

A very important feature of this population object is that you can store many generations of the population in a single population object. You can choose to store all or a limited number of generations during evolution. In the latter case, the oldest generation will be removed if a new generation is pushed in and the number of stored generations has exceeded the specified level.

simuPOP provides a large number of population related functions, they are used to

- access genotype structure
- access individuals and their genotypes
- manipulate subpopulations
- access ancestral generations
- manipulate genotype
- sample (subset) from the population
- access population variables
- save/load populations in various formats
- control virtual subpopulation structure.

You usually do not need to use these functions explicitly unless you need to write pure python functions/operators that involves complicated manipulation of populations, or when you need to manipulate populations directly for gene mapping, import/export purposes.

`population` objects are essential to simuPOP. They are composed of subpopulations each with certain number of individuals having the same genotypic structure. Class `population` has a large number of member functions, ranging from reviewing simple properties to generating a new population from the current one. Fortunately, you do not have to know all the member functions to use a population unless you need to write pure Python functions/operators that involves complicated manipulation of populations.

simuPOP supports subpopulations with boundary, and virtual subpopulations within subpopulations. Mating is within subpopulations only. Exchanges of genetic information across subpopulations can only be done through migration. Population and subpopulation sizes can be changed, as a result of mating or migration. More specifically,

- migration can change subpopulation size; create or remove subpopulations. Since migration can not generate new individuals, the total population size will not be changed.
- mating can fill any population/subpopulation structure with offspring. Both population and subpopulation sizes can be changed. Since mating is within subpopulations, you can not create a new subpopulation through mating.
- a special operator `pySubset` can shrink the population size. It removes individuals according to their `subPopID()` status. (Will explain later.) This can be used to model a sudden population decrease due to some natural disaster.
- subpopulations can be split or merged.

Note that migration will most likely change the subpopulation sizes. To keep the subpopulation sizes constant, you can set the subpopulation sizes during mating so that the next generation will have desired subpopulation sizes.

2.4.1 Virtual subpopulations

simuPOP 0.8.2 introduces the concept *virtual subpopulations*. Virtual subpopulations are groups of individuals in a subpoulation, defined by certain criteria. 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 virtual subpopulations. Virtual subpopulations do not have to add up to the whole subpopulation, nor do they have to be distinct. Because properties of individuals are variable, virtual subpopulations do not have fixed sizes as subpopulations do.

Virtual subpopulations allow easy handling of heterogeneous populations, and can facilitate some computations that are previously very difficult to do. For example, mating schemes can work on virtual subpopulations. This allows

complicated mating schemes such as mating in aged population, and mixed mating schems. By limiting operators to virtual subpopulations, one can apply different genetic forces to different groups of individuals. A good example is to migrant only male from a subpopulation to other subpopulations. It is also easy to calculate statistics at a finer scale, such as allele frequency of all males.

Virtual subpopulations are defined by virtual splitters. A splitter splits a subpopulation into pre-determined number of virtual subpopulations. It also assign a name, such as `age=5` to each virtual subpopulation. For example

- A `sexSplitter` splits the population into male and female virtual subpopulations. A `affectionSplitter` splits the population into unaffected and affected virtual subpopulations.
- A `infoSplitter` splits the population according to values of a given information field. It can split the population by given values, or by some cut-off values.
- A `proportionSplitter` splits the population with given proportions, and a `rangeSplitters` choose individuals from given ranges.
- A `genotypeSplitter` splits the population with genotype values at given loci. Multiple genetypes are allowed for a virtual subpopulation. For example, `genotypeSplitter(1, [0, 0, 0, 1])` defines a virtual subpopulation with individuals having genotype `0, 0` or `0, 1` at locus `1`.
- A `combinedSplitter` allows the specification of multiple splitter at the same subpopulation. For example, the unaffected and affected virtual subpopulation of a subpopulation split by `combinedSplitter([sexSplitter(), affectionSplitter()])` are 2 and 3, respectively.

There is currently no easy way to get the intersection or superset of two virtual subpopulations, such as a virtual subpopulation with male and/or affected individuals. It is possible, though, to define an information field that reflect these logics and define a virtual subpopulation according to this information field.

The population class provies several functions to assign a splitter to a given population, retrieve virtual subpopulation sizes and names. Note that one splitter is used for all subpopulations. If different splitters are needed for different subpopulations, a combined splitter can be used. More interestingly, the `individuals(subPop, virtualSubPop)` member function allows you to iterate through all individuals of a virtual subpopulation.

Member functions `pop.numVirtualSubPop(sp), pop.virtualSubPopSize(sp, vsp)` can be used to determine the number of virtual subpopulation a subpopulation has, and the size of the virtual subpoulation. Operator `stat(popSize=True)` also calculates virtual subpoulation sizes, and save them in a variable `virtualPopSize`.

Example 2.11 demonstrates how to assign virtual splitter, and how to use them.

Example 2.8: Virtual subpopulation related functions

```
>>> import random
>>> pop = population(1000, loci=[2, 3], infoFields=['age'])
>>> InitByFreq(pop, [0.2, 0.8])
>>> for ind in pop.individuals():
...     ind.setInfo(random.randint(0, 5), 'age')
...
>>> # split by age
>>> pop.setVirtualSplitter(infoSplitter('age', values=[2, 4]))
>>> pop.subPopSize([0, 0])
171
>>> pop.virtualSubPopName(1)
'age = 4'
>>>
>>> # split by genotype
>>> a = pop.setVirtualSplitter(
...     genotypeSplitter(locus=2, alleles=[[0, 1], [1, 1]], phase=True))
>>> pop.subPopSize([0, 0])
```

```
191
>>> pop.subPopSize([0, 1])
611
>>>
>>> for ind in pop.individuals([0, 0]):
...     assert ind.allele(2, 0) == 0 and ind.allele(2, 1) == 1
...
>>>
```

2.4.2 Access individuals from a population object

Individuals can not be created without population. You can create a population and access its individuals through the `individual(), individuals()` functions. The returned `individual` object has its own member functions, with which you can

- access genotype structure
- retrieve/set genotype
- retrieve/set sex, affected status and some other auxiliary information (information fields)

Individuals of a population can be accessed through `individual()`, or its iteration form `individuals()` function:

- `individual(ind)` returns the `ind`'th individual (absolute index) of the whole population.
- `individual(ind, subPop)` returns the `ind`'th (relative index) individual in the `subPop`'th subpopulation.
- `individuals()` return an iterator that can be used to iterate through all individuals in a population.
- `individuals(subPop)` return an iterator that can be used to iterate through all individuals in the `subPop`'th subpopulations.
- `ancestor(ind, gen)` returns the `ind`'th individual (absolute index) of the gen'th ancestral generation.
- `ancestor(ind, subPop, gen)` returns the `ind`'th (relative index) individual in the `subPop`'th subpopulation.

For example, example 2.9 iterates through all individuals in subpopulaiton 2 using `population::individual()` function, while 2.10 uses `population::individuals()`. The latter is usually easier to use.

You can also access individuals from the ancestral generations directly. There is no batch access functions such as `individuals()`. If they are needed, use `useAncestralPop()` to switch to that ancestral generation and run `individuals()` for the current generation.

Example 2.9: Function population::individual()

```
for i in range(pop.subPopSize(2)):
    ind = pop.individual(i, 2)
    print ind.affected()
```

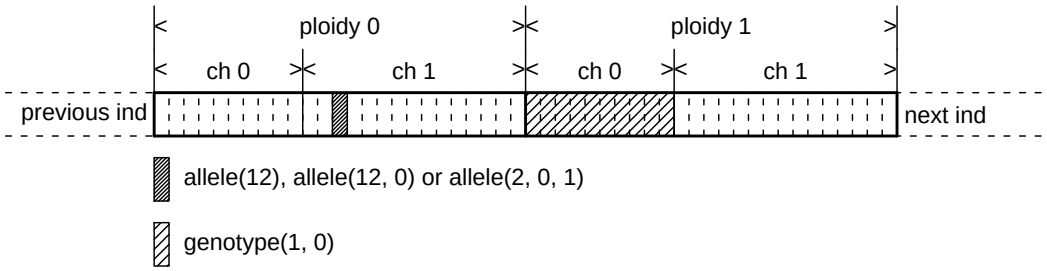
Example 2.10: Function population::individuals()

```
for ind in pop.individuals(2):
    # do something to ind
    print ind.affected()
```

2.4.3 Access individual and population genotypes

simuPOP provies several functions to read/write individual genotype. It is important to understand how simuPOP store these genotypes before you use these functions. Regardless of internal implementation, you can consider the genotype of an individual as an array of alleles, ordered by marker, chromosome and ploidy. This is illustrated in Figure 2.1.

Figure 2.1: 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)

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. This is demonstrated in the following example.

Example 2.11: Virtual subpopulation related functions

```
>>> pop = population(size=[3, 2], loci=[2])
>>> # single allele access
>>> 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
>>> # batch access
>>> ind = pop.individual(4)
>>> geno = ind.genotype()
>>> print geno
[0, 1, 0, 1]
>>> geno[2] = 3
>>> print ind.genotype()
[0, 1, 3, 1]
>>> # direct modification of the underlying genotype
>>> geno[2:4] = [3, 4]
```

```
>>> print ind.genotype()
[0, 1, 3, 4]
>>> # set genotype
>>> ind.setGenotype([2, 1])
>>> print geno
[2, 1, 2, 1]
>>> # print genotypes of all individuals in the second subpopulation.
>>> print pop.genotype(1)
[0, 1, 0, 1, 2, 1, 2, 1]
>>>
```

The return value of two simuPOP functions, namely `individual::genotype([p, [ch]])` and `population::genotype([sp])` is of a special Python type `carray`. This object reflects the underlying C/C++ array which can be modified through this list-like interface, with the exception that you can not change the size of the array. 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.

2.4.4 Population Variables

Populations are associated with python variables. These variables are usually set by various operators. For example, `stat` operator calculates many population statistics and store results in population namespace. However, you can also make use of this mechanism to pass parameters, store variables etc.

The interface functions are `population::vars()` and `population::dvars()` function. They are identical except that `vars()` returns a python dictionary and `dvars()` returns a wrapper class so that you can access this dictionary as attributes. For example, `pop.vars()['alleleFreq'][0]` is the same as `pop.dvars().alleleFreq[0]`. To have a look at all associated variables of a population, you can print `pop.vars()`, or better pass `pop.vars()` to a function `ListVars()`. A nice GUI will be used if wxPython is installed.

It is important to know that this dictionary forms a local namespace in which expressions can be evaluated. As we can see from example **??**, the same expression "`'%f ' % LD[0][1]`" can be evaluated in each population's local namespace and yield different results.

Populations are associated with Python variables. These variables are usually set by various operators but you can also set them manually. For example, `stat` operator calculates many population statistics and store the results in a population's local namespace.

Conceptually, population variables are organized as follows (looking from a simulator's point of view):

```
simu.vars(0)          simu.vars(1) ...      // replicate
  popSize               popSize             // local namespace
  alleleFreq[0]         alleleFreq[0]       // allele frequency at locus 1
  alleleFreq[1]         alleleFreq[1]       // at locus 2
  ...                   ....
  subPop[0]             subPop[0]           // subpop namespace
    popSize               popSize           // subpopulation 1 size
    allaleFreq[0]         allaleFreq[0]     // allele frequency at locus 1
    ...                   ...
  subPop[1]             subPop[1]           // variables for subpop 2
    ...                   ...
```

You can refer to these variables using `population::vars()` or `population::dvars()` function. The returned values of `vars()` and `dvars()` reflect the same dictionary, but `dvars()` uses a little Python magic so

that you can use attribute syntax to access dictionary keys. Because `a.allaleFreq[0]` is easier to read than `a['alleleFreq'][0]`, `dvars()` is more frequently used.

There are several ways to use these two functions

- `pop.vars(), pop.dvars()` return the variables of population `pop`
- `pop.vars(subPop), pop.dvars(subPop)` returns dictionary `pop.vars()['subPop'][subPop]`
- `simu.vars(rep), simu.dvars(rep)` return the variables of the `rep`'th population of simulator `simu`, i.e. `simu.population(rep).vars()`.
- `simu.vars(rep, subPop), simu.dvars(rep, subPop)` returns dictionary `simu.vars(rep)['subPop'][subPop]`

Direct access to variables `pop.vars()['subPop'][subPop]` is provided because statistics calculator `stat`, by default, calculates the same set of statistics for all subpopulations (and the whole population).

To have a look at all variables defined in this dictionary, you can use function `ListVars` defined in `simuUtil.py`. With wxPython installed, this function opens a nice window with a tree representing the variables. Without wxPython (or use parameter `useWxPython=False`), variables are displayed in an indented form. Several parameters can be used to limit your display. They are

- `level`: the level of the tree, further nested variables will not be displayed
- `name`: the name of the variable to display
- `subPop`: whether or not display variables for each subpopulation.

Example 2.12: Population variables

```
>>> from simuUtil import ListVars
>>> pop = population(size=[1000, 2000], loci=[1])
>>> InitByFreq(pop, [0.2, 0.8])
>>> ListVars(pop.vars(), useWxPython=False)
 rep :  -1
>>> Stat(pop, popSize=1, alleleFreq=[0])
>>> # subPop is True by default, use name to limit the variables to display
>>> ListVars(pop.vars(), useWxPython=False, subPop=False, name='alleleFreq')
 alleleFreq :
   [0]
     [0]        0.197166666667
     [1]        0.802833333333
>>> # print number of allele 1 at locus 0
>>> print pop.vars()['alleleNum'][0][1]
4817
>>> print pop.dvars().alleleNum[0][1]
4817
>>> print pop.dvars().alleleFreq[0]
[0.19716666666666666, 0.80283333333333329]
>>> print pop.dvars(1).alleleNum[0][1]
3204
>>>
```

Population variables is a Python dictionary, and furthermore a *Local namespace*, which means that you can use dictionary items as variables during evaluation. To evaluate in a population's local namespace, you can use function `population::evalulate()` or `population::execute()`. For example:

Example 2.13: Local namespaces of populations

```
>>> pop = population(size=[1000, 2000], loci=[1])
>>> InitByFreq(pop, [0.2, 0.8])
>>> Stat(pop, popSize=1, alleleFreq=[0])
>>> print pop.evaluate('alleleNum[0][0] + alleleNum[0][1]')
6000
>>> pop.execute('newPopSize=int(popSize*1.5)')
>>> ListVars(pop.vars(), level=1, useWxPython=False)
 newPopSize :   4500
 rep :  -1
 popSize :      3000
 numSubPop :    2
 alleleNum :
   list of length 1
 virtualPopSize :
   list of length 2
 subPopSize :
   list of length 2
 alleleFreq :
   list of length 1
 subPop
   list of length 2
>>> # this variable is 'local' to the population and is
>>> # not available in the main namespace
>>> newPopSize
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
NameError: name 'newPopSize' is not defined
>>> #
>>> simu = simulator(population(10), noMating(), rep=2)
>>> # evaluate an expression in different areas
>>> print simu.vars(1)
{'rep': 1, 'gen': 0}
>>> # a statement (no return value)
>>> simu.population(0).execute("myRep=2+rep*rep")
>>> simu.population(1).execute("myRep=2*rep")
>>> print simu.vars(0)
{'rep': 0, 'myRep': 2, 'gen': 0}
>>>
```

These two functions are rarely used, because

```
pop.evaluate('alleleNum[0][1] + 1')
```

is equivalent to

```
pop.dvar().alleleNum[0][1] + 1
```

Operators `pyEval/pyExec` are more useful in that they can be applied to different populations during evolution, and report statistics calculated by operator `stat` dynamically. The difference between these two operators are that `pyEval` evaluates a Python expression and returns its value, while `pyExec` executes a list of statements in the form of a multi-line string, and does not return any value.

Example 2.14: Use of operators `pyEval` and `pyExec`

```
>>> simu = simulator(population(100, loci=[1]),
...     randomMating(), rep=2)
>>> simu.evolve(
...     preOps = [initByFreq([0.2, 0.8])],
...     ops = [ stat(alleleFreq=[0]),
...         pyExec('myNum = alleleNum[0][0] * 2'),
...         pyEval(r'"gen %d, rep %d, num %d, myNum %d\n"' \
...             ' % (gen, rep, alleleNum[0][0], myNum)')
...         ],
...     gen=3
... )
gen 0, rep 0, num 46, myNum 92
gen 0, rep 1, num 62, myNum 124
gen 1, rep 0, num 43, myNum 86
gen 1, rep 1, num 58, myNum 116
gen 2, rep 0, num 41, myNum 82
gen 2, rep 1, num 67, myNum 134
(3, 3)
>>>
```

2.4.5 Information fields

An individuals have genotype, sex and affection status information, but other information may be needed. For example, one or more trait values may be needed to calculate quantitative traits, and one may want to keep track of all offspring of a parent. Because the need for information fields varies from simulation to simulation, simuPOP does not fix the amount of information fields, and allow users to specify these fields during the construction of populations, or add them when you need them.

Operators may require certain information fields to work properly. For example, all selectors require field `fitness` to store evaluated fitness values for each individual. `parentTagger` needs `father_idx` and `mother_idx` to store indices of the parents of each individual in the parental generation. These information fields can be added by the `infoFields` parameter of the population constructor or be added later using relevant function. If a required information field is unavailable, an error message will appear and tell you which field is needed. Some operators allow you to specify which information field(s) to use. For example, quantitative trait operator can work on specified fields so an individual can have several quantitative traits.

The information fields is usually set during population creation, using the `infoFields` option of population constructor. It can also be set or added by functions

- `pop.setInfoFields(fields, init)` set information fields of a population, removing all previous ones
- `pop.addInfoField(field, init)` add an information field to a population
- `pop.addInfoFields(fields, init)` add information fields to a population
- `simu.addInfoField(field, init)` add an information field to all populations in a simulator
- `simu.addInfoFields(fields, init)` add information fields to all populations in a simulator

When adding information fields to a simulator, information fields are added to all populations of the simulator. Note that it is illegal to add information field (or in a broader sense changing genotypic structure) to part of the populations of a simulator, because all populations in a simulator should have the same genotypic structure.

One can read/write information fields at individual level:

- `ind.info(idx), ind.info(name)` return individual information field by index or name
- `ind.setInfo(value, idx), ind.setInfo(value, name)` set individual information field by index or name
- `ind.arrInfo()` returns a carray of all information fields of an individual

or at the population level

- `pop.indInfo(idx), pop.indInfo(name)` return an information field (referred by index or name) of all individuals
- `pop.indInfo(idx, subPop), pop.indInfo(name, subPop)` return an information field (referred by index or name) of all individuals in a subpopulation `subPop`.
- `pop.setIndInfo(values, idx), pop.setInfIndo(values, name)` set information fields of all individuals with values in an array.

Both `idx` or `name` can be used in these functions. `name` is easier to use but `idx`, which can be obtained by `idx=pop.infoIdx(name)`, is faster.

Example 2.15: Use regular information field function

```
>>> pop = population(10, infoFields=['a', 'b'])
>>> aIdx = pop.infoIdx('a')
>>> bIdx = pop.infoIdx('b')
>>> for ind in pop.individuals():
...     a = ind.info(aIdx)
...     ind.setInfo(a+1, bIdx)
...
>>> print pop.indInfo(bIdx)
(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
>>>
```

Information fields can also be used to track individual information fields during evolution, using Python operators or operators `infoEval` and `infoExec`. The latter two operators can evaluate Python expressions and statements with varaibles being the information fields of individuals. Changes to these variables will change the corresponding information fields of an individul. For example, assuming that population `pop` has information field `a`, the following function (function form of operator `infoExec`) will increase the information field `a` of every individual in the population by 1.

```
InfoExec(pop, 'a += 1')
```

These statements are usually used to change the values of an information field, or derive an information field from other ones. However, variables from a poulation's local namespace can be made available, using option `usePopVars=True`.

Example 2.16: Use infoExec and infoEval operators

```
>>> pop = population(5, infoFields=['a', 'b'])
>>> InfoExec(pop, 'import random\na=random.randint(2, 10)')
>>> InfoExec(pop, 'b=a+a*2')
>>> InfoEval(pop, r"'(%.0f, %.0f) ' % (a, b)")
(7, 21) (5, 15) (9, 27) (10, 30) (6, 18) >>>
```

```
>>> # this is wrong because 'c' is not available
>>> InfoExec(pop, 'b=c+a')
Traceback (most recent call last):
  File "<embed>", line 1, in ?
NameError: name 'c' is not defined
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
  File "/usr/lib64/python2.4/site-packages/simuPOP_std.py", line 9735, in InfoExec
    infoExec(*args, **kwargs).apply(pop)
SystemError: Evalulation of statements failed
>>> # but we can also make use of population variables.
>>> pop.vars()['c'] = 6
>>> InfoExec(pop, 'b=c+a', usePopVars=True)
>>> print pop.indInfo('b')
(13.0, 11.0, 15.0, 16.0, 12.0)
>>>
```

2.4.6 Ancestral populations

By default, a population object only holds the current generation. All ancestral populations (generations) will be discarded. You can, however, keep as many ancestral generations as you wish, provided that you have enough RAM to store all these extra information.

Parameter `ancestralDepth` is used to specify the number of generations to keep. This parameter is default to `0`, meaning keeping no ancestral population. You can specify a positive number `n` to store most recent n generations; or `-1` to store all populations.

Several important usage of ancestral generations:

- `dumper()` operator and `Dump()` function has a parameter `ancestralPops`. If set to `True`, they will dump all ancestral generations.
- function `population::setAncestralDepth()` and operator `setAncestralDepth()` set the number of ancestral generations to keep for a population. A typical use of `setAncestralDepth()` is

  ```
  simu.evolve(...
    setAncestralDepth(3, at=[-3])
  )
  ```

  which saves the last three generations in populations so that pedigree based sampling schemes can be used.
- `pop.useAncestralPop(idx)` set the current generation of population `pop` to `idx` generation. `idx = 1` for the first ancestral generation, `2` for second ancestral ..., and `0` for the current generation. After this function, all functions, operators will be applied to this ancestral generation. You should always call `setAncestralPop(0)` after you examined the ancestral generations.

A typical use of this function is demonstrated in example 2.17. In this example, a population with two loci is created and with initial genotype 0. Two `kamMutator` with different mutation rates are applied to these two loci. Five most recent populations are kept. The allele frequencies at these generations are calculated afterward. (Note that this is not the best way to exam the changes of allele frequencies, a `stat` operator should be used.)

Example 2.17: Ancestral populations

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

```
...     ops = [
...         setAncestralDepth(5, at=[-5]),
...         kamMutator(rate=0.01, loci=[0], maxAllele=1),
...         kamMutator(rate=0.001, loci=[1], maxAllele=1)
...     ],
...     gen = 20
... )
(20,)
>>> pop = simu.population(0)
>>> # start from current generation
>>> for i in range(pop.ancestralDepth()+1):
...     pop.useAncestralPop(i)
...     Stat(pop, alleleFreq=[0, 1])
...     print '%d     %5f     %5f' % \
...         (i, pop.dvars().alleleFreq[0][1], pop.dvars().alleleFreq[1][1])
...
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'population' object has no attribute 'ancestralDepth'
>>> # restore to the current generation
>>> pop.useAncestralPop(0)
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'population' object has no attribute 'useAncestralPop'
>>>
```

2.4.7 Save and Load a Population

Internally, population can be saved to or loaded from disk files using `savePopulation(file)` member function, global `SavePopulation(pop, file)` and `LoadPopulation`. (Yes, it is `Load..` not `load..` because `savePopulation` is a member function and `LoadPopulation` is a global function.). Although files in any extension can be saved/loaded correctly, extension `.pop` is usually used. Populations are compressed in gzip format to save some disk space.

Populations can also be saved in other formats such as FSTAT so that they can be directly analyzed by other programs. These formats are not supported internally. They are handled in Python in the form of Python function or pure-Python operator. If you would like to save/load simuPOP population in your own format, you can do it by mimicking these functions in `simuUtil.py`.

Shared variables (c.f section **??**) are also saved (except for big objects like samples). Since the number of shared variables can be very large, it maybe a good idea to clear these variables before you save a population. On the other hand, you may want to save key parameters used to generate this population in the local namespace so that you will know these parameters after the population is loaded. For example, you can do

Example 2.18: Save population variables

```
pop.vars().clear()
pop.dvars().migrationRate = 0.002
pop.dvars().diseaseLoci = [4, 30]
SavePopulation(pop, 'example.pop')
```

2.4.8 View a population (GUI, wxPython required)

Introduced in version 0.6.9, `simuViewPop.py` can be used to view a population. It can be used as a standalone application, or in an interactive session. First, you can use this script as a standalone application, simply run

```
simuViewPop.py mypop.bin
```

will fire a GUI and allow you to exam population property, genotype and calculate statistics.

In a Python session, import this module will provide a function `viewPop`, apply it on a in-memory population or a filename will have the same effect. For example,

Example 2.19: Use simuViewPop to view a population

```
import simuViewPop
simuViewPop.viewPop(myPop)
simuViewPop.viewPop(filename='mypop.bin')
```

2.5 Mating Scheme

Mating schemes specify how to generate offspring from the current population. It must be provided when a simulator is created. Mating can perform the following tasks:

- Choose parent(s) to generate offspring to populate the next generation. The number of offspring per mating event can be a fixed number (default to 1), or a random number following one of geometric, Poisson or binomial distribution. Customized (hybrid) parent choosers can be used. Offspring sex can be assigned randomly, with specified or default (0.5) probability, or arranged to have certain number of males/females per mating event.
- Change population/subpopulation sizes. This is where demographic models are handled in simuPOP. There are a few methods to control population sizes. The most flexible one is through a user-provided function that returns population (subpopulation) sizes at each generation.
- During-mating operators are applied to all offspring. The most commonly used during mating operator is a recombinator that can recombine parental chromosomes and form offspring genotype.
- Apply selection if applicable. If individual fitness are given (usually returned by a selector operator), a mating scheme will choose an individual to mate, according to its relative fitness.

A few mating schemes are available, among which `randomMating()` is the most important. Non-random mating can be achieved using `pyMating` and `heteroMating`, which is explained in detailed in *simuPOP reference manual*.

2.5.1 Determine the number of offspring during mating

Parameters `numOffspring, maxNumOffspring, numOffspringFunc` and `mode` are provided for each mating scheme (each offspring generator, to be exact) to determine the number of offspring produced at each mating event.

The default value of `numOffspring` parameter makes a mating scheme produces one offspring per mating event. This is required by random mating schemes and should be used whenever possible. However, various situations require a larger family size or even changing family sizes. simuPOP provides a comprehensive way to deal with this problem.

As described in the class reference, the method to determine the number of offspring is to set the `mode` parameter:

- `MATE_NumOffspring`: Produce `numOffspring` offspring all the time.
- `MATE_PyNumOffspring`: When `numOffspringFunc` is defined, this mode is automatically used. A user provided function is called whenever a mating event happens. The return value determins the number of offpspring to use.
- `MATE_GeometricDistribution`: `numOffspring` is considered as p for a geometric distribution. The number of offspring for each mating is determined by

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

- `MATE_PoissonDistribution`: `numOffspring` is considered as p for a Poission distribution. The number of offspring for each mating is determined by

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

  Since the mean of this shifted Poisson distribution is $p+1$, you need to specify, for example, 2, if you want a mean family size 3.
- `MATE_BinomialDistribution`: `numOffspring` is considered as p for a Binomial distribution. Let N=`maxNumOffspring`, the number of offspring for each mating is determined by

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

- `MATE_UniformDistribution`: `numOffspring` is be considered as a, b for a Uniform distribution, respectively. The number of offspring for each mating is determined by

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

Note that all these distributions are adjusted to produce at least one offspring.

2.5.2 Determine offspring sex

When the last chromosome is a sex chromosome (`sexChrom=True`), offspring sex is determined by his/her genotype. If an offspring is cloned from his/her parent using a `cloneOffspringGenerator()`, offspring sex is the same as his/her parent. Otherwise, offspring is by default assigned to Male and Female with equal probability 0.5.

More advanced sex assignment mode is determined by parameters `sexMode` and `sexParam` of a mating scheme or an offspring generator (see later section). `sexMode` can be

- `MATE_RandomSex` This is the default mode where offspring can be Male or Female with equal probability.
- `MATE_ProbOfMale` In this mode, parameter `sexParam` is considered as the probability of a Male offspring.
- `MATE_NumOfMale` In this mode, parameter `sexParam` is the number of male in the family. If the number of offspring at a mating event is less than this number, all offspring will be male.
- `MATE_NumOfFemale` Similar to `MATE_NumOfMale` but parameter `sexParam` is considered as the number of female in the family.

`MATE_NumOfMale` and `MATE_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.

2.5.3 Determine subpopulation sizes of the next generation

The default behavior of simuPOP is to use the same population/subpopulation sizes as those of the parent generation. You can change this behavior by setting one of `newSubPopSize`, `newSubPopSizeExpr`, and `newSubPopSizeFunc` parameters:

- If you would like to have fixed subpopulation sizes, use `newSubPopSize=some_fixed_values`. This is useful when subpopulation sizes are changed by migration and you do want to keep constant subpopulation sizes.
- If subpopulation sizes can be easily calculated through an expression, you can use `newSubPopSizeExpr` to determine the new subpopulation sizes. For example, `newSubPopSizeExpr='[gen+10]'` uses the generation number + 10 as the new population size. More complicated expressions can be used, maybe along with `pyExec` operators, but in these cases, a specialized function and `newSubPopSizeFunc` are recommended.
- A more organized (and thus recommended) way to set new population/subpopulation sizes is through parameter `newSubPopSizeFunc`. To use this parameter, you need to define a Python function that takes two parameters: the generation number and the current subpopulation sizes, and return an array of new subpopulation sizes (return `[newsize]` instead of `newsize` when you do not have any subpopulation structure). The example of `class Mating` demonstrates the use of this parameter.

2.5.4 Demographic change functions

`newSubPopSizeFunc` can take a function with parameters `gen` and `oldSize`. A few functions are defined in `simuUtil.py` that will return such a function with given parameters. All these functions support a burnin stage and then split to equal sized subpopulations. For all these functions, you can test them by

```
func = oneOfTheDemographicFunc(parameters)
gen = range(0, yourEndGen)
r.plot(gen, [func(x)[0] for x in gen])
```

`numSubPop` is default to `1`. `split` is default to `0` or given `burnin` value. Population size change happens **after** burnin (start at `burnin+1`) and split happens at `split`.

```
ConstSize(size, split, numSubPop, bottleneckGen, bottleneckSize)
```

The population size is constant, but will split into `numSubPop` subpopulations at generation `split`. If `bottleneckGen` is specified, population size will be `bottleneckSize` at that generation.

```
LinearExpansion(initSize, endSize, end, burnin, split, numSubPop,
    bottleneckGen, bottleneckSize)
```

Linearly expand the population size from `intiSize` to `endSize` after `burnin`, split the population at generation `split`. If `bottleneckGen` is specified, population size will be `bottleneckSize` at that generation.

```
ExponentialExpansion(initSize, endSize, end, burnin, split,
    numSubPop, bottleneckGen, bottleneckSize)
```

Exponentially expand the population size from `intiSize` to `endSize` after `burnin`, split the population at generation `split`. If `bottleneckGen` is specified, population size will be `bottleneckSize` at that generation.

```
InstantExpansion(initSize, endSize, end, burnin, split,
    numSubPop, bottleneckGen, bottleneckSize)
```

Instaneously expand the population size from `intiSize` to `endSize` after `burnin`, split the population at generation `split`. If `bottleneckGen` is specified, population size will be `bottleneckSize` at that generation.

2.5.5 Parent choosers and offspring generators

To implement more complex mating schemes, some concepts need to be understood. The first one is *parent chooser*. Parent chooser determines how parent or parents are chosen from a given subpopulation. There are several predefined parent choosers such as `linearParentChooser`, `randomParentChooser`, `randomParentsChooser`, and the most powerful one is called `pyParentsChooser`.

A `pyParensChooser` accepts a Python generator function, instead of a normal Python function. When this generator function is called, it returns a *generator* object that provides an iterator interface. Each time when the `next()` member function of this object is called, this function resumes where it was stopped last time, executes and returns what the next *yield* statement returns. An example of generator is given in simuPOP user's guide.

Example 2.20: A generator function that mimicks random mating

```
>>> from random import randint
>>>
>>> def randomChooser(pop, sp):
...     males = [x for x in range(pop.subPopSize(sp)) \
...         if pop.individual(x, sp).sex() == Male \
...             and pop.individual(x, sp).info('age') > 30]
...     females = [x for x in range(pop.subPopSize(sp)) \
...         if pop.individual(x, sp).sex() == Female \
...             and pop.individual(x, sp).info('age') > 30]
...     nm = len(males)
...     nf = len(females)
...     while True:
...         yield males[randint(0, nm-1)], females[randint(0, nf-1)]
...
>>> pop = population(size=[1000, 200], loci=[1], infoFields=['age'])
>>> # this will initialize sex randomly
>>> InitByFreq(pop, [0.2, 0.8])
>>> for ind in pop.individuals():
...     ind.setInfo(randint(0, 60), 'age')
...
>>> rc1 = randomChooser(pop, 0)
>>> for i in range(5):
...     print rc1.next(),
...
(542, 802) (607, 895) (190, 438) (712, 311) (951, 337)
>>> rc2 = randomChooser(pop, 1)
>>> for i in range(5):
...     print rc2.next(),
...
(16, 139) (142, 155) (108, 193) (57, 178) (96, 99)
>>>
```

A user defined parents chooser can be very complicated, involving user defined information such as geometric locations. An example is given in `scripts/demoNonRandomMating.py`. In example 3.11, the parents chooser `randomChooser` collects indexes of males and females that are over the age of 30 and return a pair of random male and female repeatedly. That is to say, individuals with age < 30 is not involved in mating. Of course, to completely implement age-dependent mating, other factors need to be considered. For example, a `pyTagger` is likely to be used to assign age to offspring.

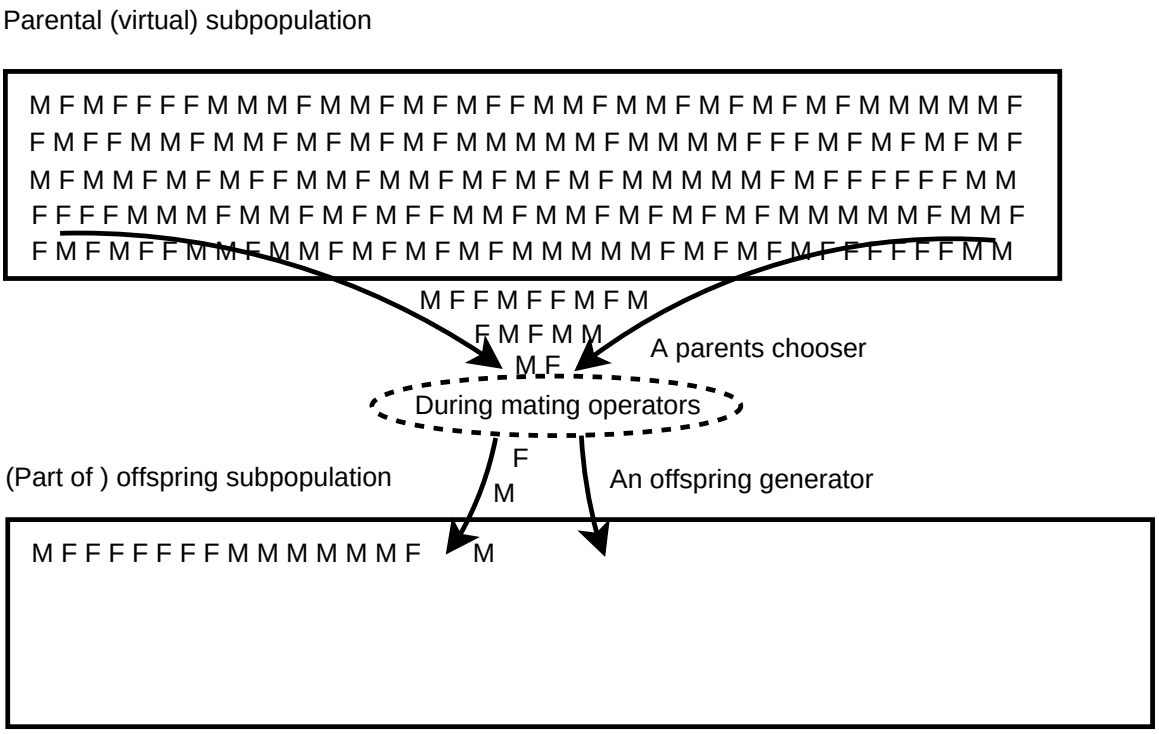
A parents chooser can yield a pair of parents, or a single parent. Obviously, a single diploid parent can not produce offspring using the usual Medelian fashion, so here comes another concept: *offspring generator*, which determines how to produce offspring from given parent or parents. Currently, there are three standard offspring generators.

These offspring generator defines only the default way to fill offspring genotype. When a during-mating operator is involved, it may override what an offspring generator does. For example, a `recombinator` recombines parental chromosomes to fill offspring genotype. In the diploid case, it will behave the same for `cloneOffspringGenerator` and `selfingOffspringGenerator`.

2.5.6 Homogeneous and hybrid mating schemes

Parent choosers and offspring generators can be combined to form homogeneous mating schemes, which work identically on all (virtual) subpopulations it is applied. The only limit is that they have to be compatible in that a parent chooser that choose one parent can not be used with an offspring generator that needs two parents. A homogenou mating scheme is illustrated in Figure

Figure 2.2: 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.

The basic usage of a `pyMating` operator is as follows

```
pyMating(randomParentChooser(),
    selfingOffspringGenerator(numOffspring=2))
```

or

```
pyMating(linearParentChooser(),
    cloneOffspringGenerator())
```

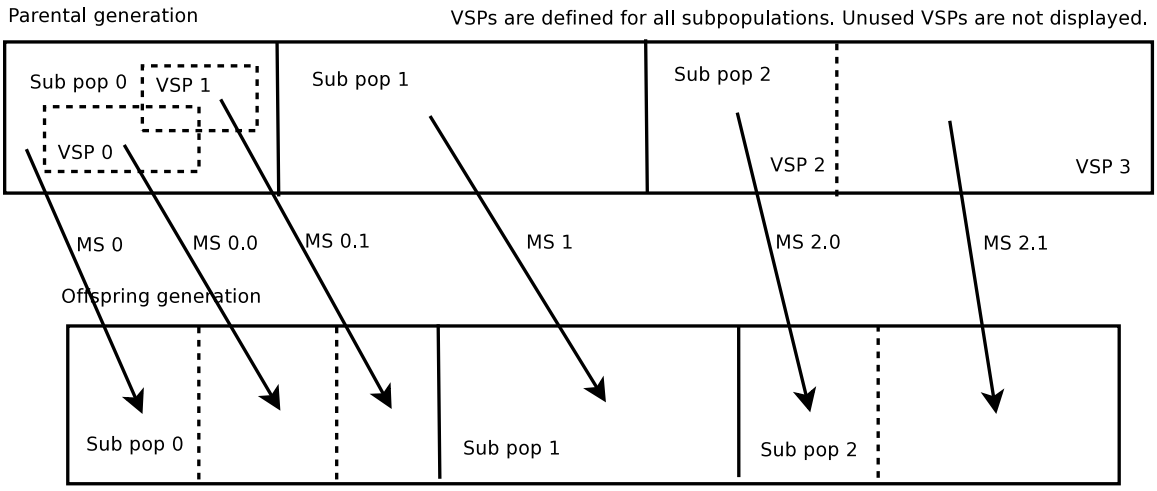
The later simply copy everyone from the parental to the offspring genertion.

2.5.7 Heterogeneous mating schemes

Non-random mating can also be introduced by mating individuals from different groups differently. Different subpopulations, or different virtual subpopulations, can have varying fecundity, represented by different numbers of offspring generated per mating event. For example, it is possible that only adults (may be defined by age > 30 and age < 50) in a subpopulation can produce offspring, where other individual will either be copied to the offspring generation or die. It is also quite common in plant genetics that a certain portion of trees go through selfing, while others go through random mating.

A `heteroMating` mating scheme accepts a list of mating schemes that works separately on different subpopulation, or virtual subpopulations. In this way, many homogenous mating schemes can be applied to different (virtual) subpopulations. This is illustrated in Figure 2.3.

Figure 2.3: Illustration of a heteogeneous mating scheme



A heterogeneous mating scheme that applies homogenous mating schemes MS0, MS0.0, MS0.1, MS1, MS2.0 and MS2.1 to subpopulation 0, the first and second virtual subpopulation in subpopulation 0, subpopulation 1, the first and second virtual subpopulation in subpopulation 2, respectively. Note that VSP 0 and 1 in subpopulation 0 overlap, and do not add up to subpopulation 0.

For example,

```
heteroMating([randomMating(numOffspring=2, subPop=0),
    randomMating(numOffspring=4, subPop=1)])
```

define a heterogeneous mating scheme that mating events in subpopulation 0 produces two offspring, while producing four in subpopulation 1.

```
pop.setVirtualSplitter(proportionSplitter([0.2, 0.8]), 0)
heteroMating([selfMating(numOffspring=2, subPop=0, virtualSubPop=0),
    randomMating(subPop=0, virtualSubPop=1)],
    shuffleOffspring=True
)
```

allows different mating schems in one subpopulation. In this example, the first subpopulation is splitted into two virtual subpopulations by proportion. Then, a selfing mating scheme is applied to the first virtual subpopulation, and a random mating scheme is applied to the second. In case that there are more than one mating schemes working on the same subpopulation, offspring are shuffled randomly by default, unless this is turned off by `shuffleOffspring=False`. Randomization of the order of offspring is usually desired because otherwise, taking this example, the first 20% of

individuals will always go through selfing, and the rest will always go through random mating. When offspring are shuffled, each individual will have probability 0.2 to be selfing, and probability 0.8 to mate randomly.

simuPOP determines if a mating scheme will be applied to a particular subpopulation using the following rules

- If neither `subPop`, nor `virtualSubPop` is specified, the mating scheme is applied to all subpoulations (as a whole, not any virtual subpopulation).
- If `subPop`, but not `virtualSubPop` is specified, the mating scheme is applied to the specified subpopulation (as a whole).
- If `subPop` and `virtualSubPop` are both specified, the mating scheme is applied to the specified virtual subpopulation.
- If `subPop` is not specified, but `virtualSubPop` is, the mating scheme is applied to spcified virtual subpopulation of all subpopulations. Note that simuPOP will report an error if a subpopulation does not define such a virtual subpopulation.

If one mating scheme is specified for each parental subpopulation, offspring subpopulation sizes are determined as usual, through parameters `newSubPopSize, newSubPopSizeFunc`, etc. However, if multiple mating schemes will be applied to the same subpopulation, they have to share the same offspring subpopulation. This problem is addressed by a weight system. That is to say, each mating scheme can be given a weight using parameter `weight`. A weight can be positive, zero (default) or negative. The number of offspring each mating scheme will produce is determined by these 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 to get a fixed size. 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 first processed, and the rest of the individuals are divided using positive weights. For example, three mating schemes with weights (-1, 2, 3) will produce 1000, $\frac{2}{5}(N-1000)$, $\frac{3}{5}(N-1000)$ individuals respectively.

2.6 Operators

Operators are objects that act on populations. They (there are exceptions) can be applied to populations directly, but most of the time they are managed and applied by a simulator. There are three kinds of operators:

- *built-in:* written in C++, fastest. They do not interact with Python shell except that some of them set variables that are accessible from Python.
- *hybrid*: written in C++ but calls python function when execution. Less efficient. For example, a hybrid mutator `pyMutator` will determine if an allele will be mutated and call a user–defined Python function to mutate it.
- *pure python*: written in python. Same speed as python. For example, a `varPlotter` can plot python variables that are set by other operators.

You do not have to know the type of an operator to use them. The interface of them are all the same. Namely, they all accept a standard set of parameters, and are used in the same fashion. Such parameters include `rep, begin, step, end` and `at`. The first two indicate that the operator only applies to one replicate, and the rest control which generation(s) the operator will be applied to. There are also parameters that redirect operator output to files. For details please refer to the reference manual.

A simuPOP life cycle (each generation) can be divided into pre-mating, during-mating and post-mating and an operator can be applied to one or more of them. For example, a `stat` operator usually applies post-mating, but if you prefer, you can change its `stage` parameter to `preMating` and apply it pre-mating.

2.6.1 Function form of an operator

Operators are usually applied to populations through a simulator. They are created and passed as parameters to the `evolve` function of a simulator. During evolution, the `evolve()` function determines if an operator can be applied to a population and apply it when appropriate. More details about operators will be described in section **??**.

You can ignore the specialties of an operator and call its `apply()` function directly. For example, you can initialize a population outside a simulator by

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

or dump the content of a population by

```
dumper().apply(pop)
```

This usage is used so often that it deserves some simplification. Equivalent functions are defined for most of the operators. For example, function `InitByFreq` is defined for operator `initByFreq` as follows

Example 2.21: Function `InitByFreq`

```
>>> def InitByFreq(pop, *args, **kwargs):
...     initByFreq(*args, **kwargs).apply(pop)
...
>>> InitByFreq(pop, [.2, .3, .4, .1])
>>>
```

The function form of an operator is listed after its class name in this reference manual.

2.7 Simulator

Simulators combine three important components of simuPOP: population, mating scheme and operators together. A simulator is usually created with an instance of population, a replicate number and a mating scheme. It makes '`rep`' replicates of this population and control the evolution process of these populations.

The most important function of a simulator is `evolve()`. It accepts arrays of operators as its parameters, among which, '`preOps`' and '`postOps`' will be applied to the populations at the beginning and end of evolution, respectively, whereas '`ops`' will be applied at every generation. Of course, a simulator will probe and respect each operator's `rep, begin, end, step, at, stage` properties and act accordingly.

2.7.1 Generation number

Several aspects of the generation number may cause confusion:

- generation starts from zero
- a generation number presents a 'to-be-evolved' generation
- the ending generation specified in `evolve()` will be executed

That is to say, a new simulator will have generation 0 (at the beginning of generation 0). If you do `evolve(..., end=0), evolve` will evolve one generation and stop at the beginning of generation 1.

It may sound strange that

```
evolve(end=2)
```

evolve the population 3 generations. Generation 0, generation 1, and generation 2. When you use `start=0, step=5, end=10` for your operator, it will be applied at generations `0,5,10` etc. At the end of the simulation, current generation number is 3! (If you are familiar with C, this is like a `for` loop index). This is why you should test if a simulation is finished correctly by

```
if(simu.gen() == endGen+1)
```

instead of `simu.gen() == endGen.` (endGen is the value for parameter `end`).

2.7.2 Operator calling sequence

In a simulation, operators are applied at different stages, pre-, during-, and post-mating (controlled by `stage` parameter), at specified generations (controlled by `begin, end, step, at` parameters), and to specified replicates (controlled by `rep` parameter). The order of applying operators usually does not matter but errors may occur if you are not careful. For example, `stat(...)` calculates the statistics of the current population. It is a pre-mating operator so you should set `stage=PostMating` and put it after all operators if you would like to measure a post-mating population. It also should be put before any operator (such as an terminator) that uses the shared variable set by `stat(...).`

If you are not sure about the calling sequence of operators, you can set the `dryrun` parameter of `evolve()` function to `True`. `evolve` will then print out the order of operators to apply. Consider that operators can be `PreMating, PostMating, PrePostMating, DuringMating` and the default value (parameter `stage`) may not be what you expect. Having a look at the calling sequence before the real evolution is always a good idea.

2.7.3 Save and Load

Using function `saveSimulator`, we can save a simulator to a file. Although files with any extension can be correctly saved/loaded, extension `.sim` is usually used. Note that a mating scheme can not be saved and has to be re-specified in `LoadSimulator()`.

Example 2.22: Save and load a simulator

```
>>> simu.save("sample.sim")
>>> simu1 = LoadSimulator("sample.sim", randomMating())
>>>
```

2.8 Utilities

Chapter 3

Selected topics

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 (new in v 0.8.5), 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), load/save in text, XML, Fstat or Linkage format. In this chapter, I will discuss some practical usages of simuPOP.

3.1 Selection

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

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

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

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

This is, however, exactly

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

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

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

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

The example for `class mapSelector` is a typical example of heterozygote superiority. When $w_{11} < w_{12} > w_{22}$, the genotype frequencies will go to an equilibrium state. Theoretically, if

$$\begin{aligned} s_1 &= w_{12} - w_{11} \\ s_2 &= w_{12} - w_{22} \end{aligned}$$

the stable allele frequency of allele 1 is

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

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

3.2 Gene conversion

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

- Two (out of four) chromatids pair and a single strand cut is made in each chromatid
- Strand exchange takes place between the chromatids
- Ligation occurs yielding two completely intact DNA molecules
- Branch migration occurs, giving regions of heteroduplex DNA
- Resulution of the Holliday junction gives two DNA molecules with heteroduplex DNA. Depending upon how the holliday junction is resolved, we either observe no exchange of flanking markers, or an exchange of flanking markers. The former forms a conversion event, which can be considered as a double recombination.

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

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

   If a conversion happens, record the end of flanking region as another recombination event.

4. Copy from another copy of parental chromosome (recombination happens), until the recorded second recombination event is reached, or another recombination event happens.

5. Repeat these steps for all chromosomes.

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

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

Note that

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

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

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

3.3 Migration

Migrator is very flexible. It can accept arbitrary migration matrix, from any subset of subpopulations to any (even new) other subset of subpopulations. To facilitate the use of common theoretical migration models, several functions are defined in `simuUtil.py`.

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

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

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

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

  and if `circular=True`, returns

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

3.4 Hybrid and pure-Python operator

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 simuPOP feature called *hybrid operator*. Such operators accept a Python function and will call this function with appropriate parameter(s) when needed. For example, example 3.1 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 (`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 of being affected.

Example 3.1: An example of hybrid operators

```
#!/usr/bin/env python
'''
Demonstrate the use of hybrid operator
'''

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

def myPenetrance(geno):
    'return penetrance given genotype at spcified disease loci'
    if geno.count(1) < 3:
        return 0.
    else:
        return 1-(1-(geno[0]+geno[1])*0.25)* \
                 (1-(geno[2]+geno[3])*0.25)* \
                 (1-(geno[4]+geno[5])*0.25)

pop = population(1000, loci=[3, 4])
InitByFreq(pop, [0.3, 0.7])
PyPenetrance(pop, loci=[2, 3, 6], func=myPenetrance)
Stat(pop, numOfAffected=True)
print pop.dvars().numOfAffected
```

Example 3.1 uses the function form of operator `pyPenetrance` and `stat` and you should use the operator form in a simulator. In these functions, operators are created with the same set of parameters as their operator form, applied to the population, and are destroyed afterward. For example,

```
PyPenetrance(pop, parameters)
```

is the same as

```
pyPenetrance(parameters).apply(pop)
```

Of course, parameters `begin, end, step` etc become meaningless in the function form. Note that if you need to apply the same operator to dozens of populations, creating one operator and applying it to all populations is more efficient than using the function form, since dozens of operators will be created and destroyed for each population in the latter usage.

If hybrid operators are still not flexible enough, you can write operators in Python. Such operators will have full access to the evolving population, and can therefore perform arbitrary operations on it. A pure-python operator has been used in the previous chapter where complex statistics are calculated and printed.

Example 3.3 uses a python operator to define a frequency-dependent selection operator which has different selection pressures depending on current disease allele frequency. In this example, a population is initialized with disease allele frequency 0.3 (allele 1). Then, at each generation, a python function `freqDependSelector` is called. This function

- unpact parameters (`DSL, min, max`)
- calculate allele frequency at the disease locus
- if disease allele frequency is less than `min`, apply a multi-allele selector and give disease allele strong advantage selection;
- if disease allele frequency is greater than `max`, apply a multi-allele selector and give disease allele strong purifying selection;

The result of this operator, unseen to users, is individual `fitness` values set by one of (maybe none of) the multi-allele selector, which will be used by `randomMating()` to select individuals accordingly to population the next generation.

One tricky point of this python operator is that although selectors are `PreMating`, namely fitness will be calculated before mating, `pyOperator` is `PostMating`. To calculate `fitness` before mating, a `stage=PreMating` parameter should be used. Otherwise, the fitness will be calculated for the offspring generation, not the current generation, as shown below:

```
preMating | mating -> offspring generation | postMating, fitness calculated
preMating | mating -> ...
```

Then, because the simulator clears selection flag at the beginning of each generation, the fitness will not be used. Tricky enough, right? The good news is that

- If you are using non-optimization libraries, simulation will fail if selection flag is on at the beginning of a generation. This prevents the use of post-mating selectors.
- If you are not sure in which order the operators are applied, use the `dryrun=True` in the evolve function. `evolve()` function will do nothing but printing out when and in which order operators will be applied.

Pure-python operators are extremely flexible and even more complicated form can be used. For example, `varPlotter` in `simuRPy.py` is a class with an instance of different plotters, and a python operator is used to call one of them. Such advanced usage of pure Python operator is beyond the scope of this guide.

Example 3.2: A frequency dependent selection operator

```
#!/usr/bin/env python
'''
Demonstrate the use of pure python operator
'''

from simuPOP import *

def freqDependSelector(pop, param):
    ''' This selector will try to control disease allele
        frequency by applying advantage/purifying selection
          to DSL according to allele frequency at each DSL. '''
    # parameters are stored with population
    (DSL, min, max) = param
    # Calculate allele frequency
    Stat(pop, alleleFreq=[DSL])
    # apply harsh advantage/purifying selection to bring
    # allele frequency back to [min, max]
    if 1-pop.dvars().alleleFreq[DSL][0] < min:
        MaSelect(pop, locus=DSL, fitness=[1, 1.5, 2])
    elif 1-pop.dvars().alleleFreq[DSL][0] > max:
        MaSelect(pop, locus=DSL, fitness=[1, 0.8, 0.6])
    return True

pop = population(1000, loci=[3, 4], infoFields=['fitness'])
simu = simulator(pop, randomMating())
simu.evolve(
    preOps = [ initByFreq(alleleFreq=[0.7, 0.3]) ],
    ops = [
        pyOperator(func=freqDependSelector, param=[2, 0.2, 0.4],
            stage=PreMating),
        pyEval(r'''"%.4f\n" % (1-alleleFreq[2][0])''', step=20),
    ],
    end = 1000)
```

3.5 Python Individual operator (moved from refManual.lyx)

A Python operator accepts a function and an optional parameter. When `pyOperator` is called, it will simply pass the accepted population (or parents and offspring in the case of `stage=DuringMating`) to the function. To use this operator, in case of `stage=PostMating`, you will need to

- define a function that handle a population as you wish.

  ```
  def myOperator(pop, para):
      'do whatever you want'
      return True
  ```

  If you return `False`, this operator will work like a terminator.

- use `pyOperator` in the form of

  ```
  pyOperator(mfunc=pyOperator, param=para)
  ```

all parameters of an operator are supported except for `output` and `outputExpr` which are ignored for now.

This operator allows implementation of arbitrarily complicated operators,. To use this operator, you will have to know how to use population-related functions. The following example shows how to implement a dynamic mutator which mutate loci according to their allele frequencies.

Example 3.3: Define a python operator

```
>>> def dynaMutator(pop, param):
...     ''' this mutator mutate common loci with low mutation rate
...     and rare loci with high mutation rate, as an attempt to
...     bring allele frequency of these loci at an equal level.'''
...     # unpack parameter
...     (cutoff, mu1, mu2) = param;
...     Stat(pop, alleleFreq=range( pop.totNumLoci() ) )
...     for i in range( pop.totNumLoci() ):
...         # 1-freq of wild type = total disease allele frequency
...         if 1-pop.dvars().alleleFreq[i][1] < cutoff:
...             KamMutate(pop, maxAllele=2, rate=mu1, loci=[i])
...         else:
...             KamMutate(pop, maxAllele=2, rate=mu2, loci=[i])
...     return True
>>>
```

Example 3.4: Use of python operator

```
>>> pop = population(size=10000, ploidy=2, loci=[2, 3])
>>>
>>> simu = simulator(pop, randomMating())
>>>
>>> simu.evolve(
...     preOps = [
...         initByFreq( [.6, .4], loci=[0, 2, 4]),
...         initByFreq( [.8, .2], loci=[1, 3]) ],
...     ops = [
...         pyOperator( func=dynaMutator, param=(.5, .1, 0) ),
...         stat(alleleFreq=range(5)),
...         pyEval(r'"%f\t%f\n"%(alleleFreq[0][1], alleleFreq[1][1])', step=10)
...         ],
...     gen = 30
... )
0.399900        0.199000
0.408400        0.203750
0.418350        0.202150
(30,)
>>>
```

`pyOperator` can also be a during-mating operator. You will need to define a function

```
def Func(pop, off, dad, mom, para)
```

or

```
def shortFunc(off, para)
```

where `para` can be ignored. To use this operator, you can do

```
pyOperator(stage=DuringMating, func=Func, param=someparam, formOffGenotype=True)
```

or

```
pyOperator(stage=DuringMating, func=shortFunc, param=someparam,
formOffGenotype=False, passOffspringOnly=True)
```

If your during-mating `pyOpeartor` returns `False`, the individual will be discarded. Therefore, you can write a filter in this way. However, since the Python function will be called for each mating event, the cost of using such an operator is high, especially when population size is large.

An example of during-mating `pyOperator` can be found in `scripts/demoPyOperator.py`.

3.6 Information fields

Information fields are, in short, double values attached to each individual. Since different applications require different information fields, simuPOP takes a minimal approach in that no information field will be used (to save RAM) by default. When you apply an operator that needs a particular field, and your population does not have it, an error message will be given so that you can add appropriate fields to the `infoFields` parameter of `population()`, or use `setInfoFields(), addInfoField(), addInfoFields()` member functions to add them. Commonly used information fields are

- `fitness`: used by all selectors, and by mating schemes
- `father_idx, mother_idx`: used by taggers to track parental information
- `spouse, pedindex, oldindex`: used by ascertainment operators to obtain pedigree information.

Besides these standard information fields, you can define any fields for your use. The most frequently used functions are `individual::setInfo(value, field), individual::info(field), population::setIndInfo(values, field)` and `population::indInfo(field)`. Here field can be the name of the field, or an id returned by `population::infoIdx(field)`. Accessing information fields using indices is faster than using names.

In the following example (Example 3.5), a proportional hazard model is used to determine the age of onset of an individual with given genotype. Briefly,

- The base hazard is $h_0(t) = \beta_0 t$, the corresponding survival function is $S(s) = \exp\left(-\int_0^s h(t)\,dt\right)$. The age of onset is determined randomly by the survival function. ($F(x) = 1 - S(x)$ is used in the example.) The relevant functions are `hazard, cumHazard, cdf, ageOfOnset`. In the last function, β is the fold change of the hazard function so $h(t, \beta) = \beta\beta_0 t$.
- Date of birth is calculated as 2005 - age, where age is $U(0, 75)$.
- The proportional hazard model is
$$h(t, X) = h_0(t) \exp(\beta X)$$
where X is the number of disease alleles at the given disease susceptibility loci. The age of onset is determined by individual $h(t, X)$.
- Affection status is determined by date of birth + age of onset < 2005.

The program is pretty self-explainary so I do not comment on the code here. The resulting population has information fields `DateOfBirth`, `betaX` and `ageOfOnset`. Note that this example does not any operator or simulator, and demonstrate simuPOP's ability to manipulation populations.

Example 3.5: Proportional hazard model and use of information fields

```
#!/usr/bin/env python
'''
Demonstrate the use of information fields.
'''
from simuOpt import setOptions
setOptions(alleleType='binary')
from simuPOP import *
from random import *
from math import exp

def hazard(t, beta):
    return beta*t

def cumHazard(t, beta):
    ''' cumulative hazard function'''
    return sum([hazard(x, beta) for x in range(0, t+1)])

def cdf(t, beta):
    ''' F(x) = 1-exp(-H(x)) '''
    return 1-exp(-cumHazard(t, beta))

def ageOfOnset(u, beta, beta0):
    ''' u is Unif(0,1), beta is fold change '''
    aa = 75
    for age in range(75):
        if cdf(age, beta*beta0) > u:
            aa = age
            break
    return aa

def simuDateOfBirth(pop):
    dobIdx = pop.infoIdx('DateOfBirth')
    for ind in pop.individuals():
        age = randint(0, 75)
        ind.setInfo(2005-age, dobIdx)

def simuBetaX(pop, DSL, beta):
    bxIdx = pop.infoIdx('betaX')
    for ind in pop.individuals():
        X = sum([ind.allele(i, 0) + ind.allele(i, 1) for i in DSL])
        ind.setInfo(beta*X, bxIdx)

def simuAgeOfOnset(pop, beta0):
    bxIdx = pop.infoIdx('betaX')
    aaIdx = pop.infoIdx('ageOfOnset')
    for (idx, ind) in enumerate(pop.individuals()):
        bx = ind.info(bxIdx)
        ind.setInfo(ageOfOnset(uniform(0,1), exp(bx), beta0), aaIdx)

def setAffection(pop):
    'set affected if age of onset + date of birth < 2005'
    aaIdx = pop.infoIdx('ageOfOnset')
```

```
    dobIdx = pop.infoIdx('DateOfBirth')
    for ind in pop.individuals():
        if ind.info(aaIdx) + ind.info(dobIdx) < 2005:
            ind.setAffected(True)
        else:
            ind.setAffected(False)

pop = population(1000, loci=[5, 9])
InitByFreq(pop, [.9, .1])
# suppose we load population from somewhere else, need to add information fields
pop.setInfoFields(['DateOfBirth', 'betaX', 'ageOfOnset'])
simuDateOfBirth(pop)
simuBetaX(pop, [4, 7], 1)
simuAgeOfOnset(pop, 0.0001)
setAffection(pop)
Stat(pop, numOfAffected=True)
print pop.dvars().numOfAffected
```

Information fields can also be manipulated during evolution, using one of the Python operators, or operators `infoEval` and `infoExec` (new in version 0.8.4). Please refer to simuPOP reference manual for details.

3.7 Pedigree

A pedigree records the parent(s) of each individual during evolution. It can be created manually or using tagging operators `parentTagger` (tagging one parent) and `parentsTagger` (tagging both parents). The pedigree can be analyzed to study various properties of the evolutionary process, manipulated (e.g. removing individuals without offspring), and used to re-realize the evolutionary process using `pedigreeMating`.

A pedigree file has the following format:

```
p1 p2 p3 p4 ..... # sp1 sp2 sp3
p1 p2 p3 p4 ..... # sp1 sp2 sp3
...
```

Numbers before # of each line of a pedigree file are the parent(s) of individuals, starting from generation 0. If only one parent is used to produce offspring (e.g. using the `selfMating` mating scheme), `parentTagger(output, outputExpr)` records the index of the parent of each individual (`p...`) in the parental generation. Otherwise, `parentsTagger(output, outputExpr)` records the indexes of both parents.

The generation number and the size of subpopulations are listed after the # character. The sum of subpopulation sizes should match the individuals listed before #.

A number of auxillary information pedigrees can be loaded after a pedigree is created. These information pedigree files does not have subpopulation and generation information (does not have character # and numbers after it). If there are n individuals at a generation, the corresponding line in an information pedigree file should have $m * n$ numbers where m is the number of properties for each individual. Information pedigrees can be created by other tagging operators such as `pyTagger(output, outputExpr)`.

These auxiliary information will be attached to individuals in a pedigree. They will be removed if an individual is removed from the pedigree.

3.8 Population structure and migration

You first need to understand that mating schemes populate subpopulations from their corresponding ancestral subpopulations one by one, so it can not change number of subpopulations. Split and merge of subpopulations are done by operators `splitSubPop` and `mergeSubPops` respectively. In example 3.6, these two operators are used to split and merge subpopulations, but keep total population size untouched. Note that after subpopulation merge, subpopulation 2 still exists, but with size 0. This is used to keep subpopulation id of other subpopulations unchanged.

Example 3.6: Population split and merge

```
>>> from simuPOP import *
>>> pop = population(1000, loci=[1], infoFields=['migrate_to'])
>>> simu = simulator(pop, binomialSelection())
>>> simu.evolve(
...     ops=[
...         splitSubPop(0, proportions=[0.2, 0.8], at = [3]),
...         splitSubPop(1, proportions=[0.4, 0.6], at = [5]),
...         mergeSubPops([0,2], at = [7]),
...         stat(popSize=True),
...         pyEval(r'"%s\n" % subPopSize'),
...     ],
...     gen = 10
... )
[1000]
[1000]
[1000]
[200, 800]
[200, 800]
[200, 320, 480]
[200, 320, 480]
[680, 320]
[680, 320]
[680, 320]
(10,)
>>>
```

Migration can change subpopulation size, but not total population size. In example 3.6, two migrators are used. The first migrator moves individuals from subpopulation 0 to subpopulation 1. The second migrator moves individuals around, with given proportions. For example, the migration rate

$$\begin{pmatrix} 0. & 0.2 & 0.4 \\ 0. & 0. & 0.1 \\ 0.1 & 0.1 & 0. \end{pmatrix}$$

means moving 20% of individuals from subpop 0 to 1, 40% of individuals from subpop 0 to 1, and keep 40% (automatically determined). Subpopulation sizes change accordingly.

Example 3.7: Population split and migration

```
>>> from simuPOP import *
>>> pop = population(1000, loci=[1], infoFields=['migrate_to'])
```

```
>>> simu = simulator(pop, binomialSelection())
>>> simu.evolve(
...     ops=[
...         splitSubPop(0, proportions=[0.2, 0.3, 0.5], at = [3]),
...         migrator(rate = [0.2], fromSubPop=[0], toSubPop=[1],
...             begin = 3, end = 4),
...         migrator(rate = [
...             [0, 0.2, 0.4],
...             [0, 0,   0.1],
...             [0.1, 0.1, 0]],
...             begin = 4),
...         stat(popSize=True),
...         pyEval(r'"%s\n" % subPopSize'),
...     ],
...     gen = 10
... )
[1000]
[1000]
[1000]
[161, 339, 500]
[93, 419, 488]
[88, 445, 467]
[91, 470, 439]
[67, 488, 445]
[74, 491, 435]
[88, 506, 406]
(10,)
>>>
```

But what if you need to control total population size? In this case, a demographic function is needed to specify the size of each subpopulation, at each generation. In example 3.8, function `popSize` returns exact subpopulation size at each generation, and the population will behave accordingly. It might surprise you that migration can no longer control the size of subpopulation sizes. What exactly happened is that, for example

- subpopulation size = [200, 400, 400], at the beginning of a generation
- apply migrator, subpopulation size changed to [100, 470, 430]
- pre mating operator `stat` is applied and report subpopulation sizes
- during mating, with given subpopulation sizes 200, 400, 400 of the offspring generation, the mating scheme generate 200 offspring from 100 parents in subpopulation 0, 400 offspring from 470 parents in subpopulation 1, and 400 offspring from 430 parents in subpopulation 2.
- post mating operator `stat` is applied and get the new subpopulation size.

This example also demonstrates the use of stage parameter. As a matter of fact, you can use only one `stat` operator by using `stage=PrePostMating`. If you are confused by the order of operators, use the `dryrun=True` parameter of `evolve` to check.

Example 3.8: Population split with changing population size

```
>>> from simuPOP import *
```

```
>>> pop = population(1000, loci=[1], infoFields=['migrate_to'])
>>> def popSize(gen, oldSize=[]):
...     if gen < 3:
...         return [1000]
...     elif gen < 5:
...         return [400, 500]
...     else:
...         return [300, 400, 600]
...
>>> simu = simulator(pop, binomialSelection(newSubPopSizeFunc=popSize))
>>> simu.evolve(
...     ops=[
...         splitSubPop(0, proportions=[0.3, 0.7], at = [3]),
...         migrator(rate = [0.2], fromSubPop=[0], toSubPop=[1],
...             begin = 3, end = 4),
...         splitSubPop(0, proportions=[0.3, 0.7], at = [5]),
...         migrator(rate = [
...             [0, 0.2, 0.4],
...             [0, 0,   0.1],
...             [0.1, 0.1, 0]],
...             begin = 5),
...         stat(popSize=True, stage=PreMating),
...         pyEval(r'"From %s\t" % subPopSize', stage=PreMating),
...         stat(popSize=True),
...         pyEval(r'"to: %s\n" % subPopSize'),
...     ],
...     gen = 10
... )
From [1000]     to: [1000]
From [1000]     to: [1000]
From [1000]     to: [1000]
From [247, 753] to: [400, 500]
From [322, 578] to: [400, 500]
From [108, 329, 463]    to: [300, 400, 600]
From [192, 446, 662]    to: [300, 400, 600]
From [214, 486, 600]    to: [300, 400, 600]
From [183, 472, 645]    to: [300, 400, 600]
From [179, 478, 643]    to: [300, 400, 600]
(10,)
>>>
```

You might say, OK, this looks nice, but how can I grow a population with migration acting freely? This is also easy, all you need to do is using the oldSize parameter of a demographic function in a clever way. The underlying story is that

- before mating, a mating scheme calculates current subpopulation sizes
- it calls the given demographic function with current generation number and current subpopulation sizes
- it uses the return value as the new subpopulation sizes.

Example 3.9 demonstrate an exponentially increase population with free migration between subpopulations.

Example 3.9: Population split with changing population size

```
>>> from simuPOP import *
>>> pop = population(1000, loci=[1], infoFields=['migrate_to'])
>>> def popSize(gen, oldSize=[]):
...     return [x*2 for x in oldSize]
...
>>> simu = simulator(pop, binomialSelection(newSubPopSizeFunc=popSize))
>>> simu.evolve(
...     ops=[
...         splitSubPop(0, proportions=[0.3, 0.7], at = [3]),
...         migrator(rate = [0.2], fromSubPop=[0], toSubPop=[1],
...             begin = 3, end = 4),
...         splitSubPop(0, proportions=[0.3, 0.7], at = [5]),
...         migrator(rate = [
...             [0, 0.2, 0.4],
...             [0, 0,   0.1],
...             [0.1, 0.1, 0]],
...             begin = 5),
...         stat(popSize=True, stage=PrePostMating),
...         pyEval(r'"From %s\t" % subPopSize', stage=PreMating),
...         pyEval(r'"to: %s\n" % subPopSize'),
...     ],
...     gen = 10
... )
From [1000]     to: [2000]
From [2000]     to: [4000]
From [4000]     to: [8000]
From [1897, 6103]       to: [3794, 12206]
From [2976, 13024]      to: [5952, 26048]
From [3302, 6662, 22036]        to: [6604, 13324, 44072]
From [7084, 17783, 39133]       to: [14168, 35566, 78266]
From [13348, 42615, 72037]      to: [26696, 85230, 144074]
From [25183, 96143, 134674]     to: [50366, 192286, 269348]
From [46937, 210190, 254873]    to: [93874, 420380, 509746]
(10,)
>>>
```

3.9 Non-random mating

Random-mating implies random choices of parents. Non-random mating is much more difficult to implement because there are numerous way to introduce non-randomness. One of the ways to achive non-random mating in simuPOP is to use a hybrid operator `pyMating`.

A `pyMating` mating scheme accepts a *parents chooser* and an *offspring generator*. The parents chooser is responsible for choosing one or two parents from the parental generation, and the offspring generator is responsible for generating a number of offspring from the chosen parents. There are a number of default parents choosers and offspring generators and a `pyMating` can be built with them. For example

```
pyMating(randomParentsChooser(), mendelianOffspringGenerator())
```

works exactly as a `randomMating` scheme, and

```
pyMating(randomParentChooser(), selfingOffspringGenerator(numOffspring=2))
```

works as `selfMating(numOffspring=2)`. Note that parent chooser and offspring generator should be compatible , meaning that if a parent chooser chooses one parent each time, the offspring generator should be able to produce offspring from a single parent.

The power of `pyMating` lies in its `pyParentChooser()`, which accepts a user-defined Python generator function, instead of a normal python function. Generally speaking, when a generator function is called, it returns a *generator* object that provides an iterator interface. Each time when the `next()` member function of this object is called, this function resumes where it was stopped last time, executes and returns what the next *yield* statement returns. For example, example 3.10 defines a function that calculate $f(k) = \sum_{i=1}^{k} \frac{1}{i}$ for $k = 1, ..., 10$. It does not calculate each $f(k)$ repeatedly but returns $f(1)$, $f(2)$, ... in a sequence interface.

Example 3.10: A sample generator function

```
>>> def func():
...     i = 1
...     all = 0
...     while i < 10:
...         all += 1./i
...         i += 1
...         yield all
...
>>> a = func()
>>> a.next()
1.0
>>> a.next()
1.5
>>> for i in a:
...     print '%.3f' % i,
...
1.833 2.083 2.283 2.450 2.593 2.718 2.829
>>>
```

A *parents chooser* takes two parameters, a population and a subpopulation index. It can return different generator objects for different subpopulations.

Example 3.11: A generator function that mimicks random mating

```
>>> from random import randint
>>>
>>> def randomChooser(pop, sp):
...     males = [x for x in range(pop.subPopSize(sp)) \
...         if pop.individual(x, sp).sex() == Male \
...             and pop.individual(x, sp).info('age') > 30]
...     females = [x for x in range(pop.subPopSize(sp)) \
...         if pop.individual(x, sp).sex() == Female \
...             and pop.individual(x, sp).info('age') > 30]
...     nm = len(males)
...     nf = len(females)
...     while True:
...         yield males[randint(0, nm-1)], females[randint(0, nf-1)]
...
>>> pop = population(size=[1000, 200], loci=[1], infoFields=['age'])
>>> # this will initialize sex randomly
>>> InitByFreq(pop, [0.2, 0.8])
```

```
>>> for ind in pop.individuals():
...     ind.setInfo(randint(0, 60), 'age')
...
>>> rc1 = randomChooser(pop, 0)
>>> for i in range(5):
...     print rc1.next(),
...
(542, 802) (607, 895) (190, 438) (712, 311) (951, 337)
>>> rc2 = randomChooser(pop, 1)
>>> for i in range(5):
...     print rc2.next(),
...
(16, 139) (142, 155) (108, 193) (57, 178) (96, 99)
>>>
```

A user defined parents chooser can be very complicated, involving user defined information such as geometric locations. An example is given in `cookbook/Mating_pyMating_cpp.py`. In example 3.11, the parents chooser `randomChooser` collects indexes of males and females and simply return a pair of random male and female repeatedly. This is exactly what `randomMating` does if selection is not considered. It becomes obvious now that whereas a python function can return random male/famale pair, the generator interface is much more efficient because the identification of two sex groups is done only once. Example 3.12 demonstrates how to use this user-defined parent chooser.

Example 3.12: pyMating with a user-defined parent chooser

```
>>> simu = simulator(pop,
...     pyMating(pyParentsChooser(randomChooser),
...         mendelianOffspringGenerator()))
>>> simu.step()
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'simulator' object has no attribute 'step'
>>>
```

Because arbitrary information can be stored with an individual through information fields, `pyMating` can be very complicated. For example, one can choose individuals according their age, and/or geographic information. For populations with well-defined structure, virtual subpopulations can be used. Basically, one needs to specify a virtual subpopulation splitter to a subpopulation. Then, different mating schemes can be applied to different virtual subpopulations. A simple example is given in Example 3.13 where the first subpopulation is divided into two parts. The first 20% of individuals undergo selfing, and the rest of the subpopulation undergoes usual sexed random mating. Note that two mating schemes produce different number of offspring per mating event, and the family sizes are recorded in a shared variable `famSizes` when `DBG_MATING` is turned on.

Example 3.13: A heterogeneous mating scheme

```
>>> TurnOnDebug(DBG_MATING)
>>> pop = population(100, loci=[2])
>>> pop.setVirtualSplitter(proportionSplitter([0.2, 0.8]))
>>> simu = simulator(pop, heteroMating(
...     [selfMating(numOffspring=5, subPop=0, virtualSubPop=0),
...     randomMating(numOffspring=20, subPop=0, virtualSubPop=1)]))
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
  File "/usr/lib64/python2.4/site-packages/simuPOP_std.py", line 5216, in __init__
```

```
    _simuPOP_std.selfMating_swiginit(self,_simuPOP_std.new_selfMating(*args, **kwargs))
TypeError: 'virtualSubPop' is an invalid keyword argument for this function
>>>
>>> simu.step()
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'simulator' object has no attribute 'step'
>>> print simu.dvars(0).famSizes
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'dw' object has no attribute 'famSizes'
>>> TurnOffDebug(DBG_MATING)
Debug code DBG_MATING is turned off. cf. ListDebugCode(), TurnOnDebug().
>>>
```

3.10 Sex chromosomes

Supports for sex chromosomes are done in simuPOP in the following ways:

- If `sexChrom=True` is specified in `population()`, the last chromosome is assumed to be the sex chromosome. For female, it is `XX`, for male, it is `XY`, in that order.
- During mating, sex of offspring is determined by sex chromosome. (It is otherwise determined randomly with probability 0.5).
- Recombination can not happen between `X` and `Y` chromosomes. That is to say, offspring can get recombined `X` from his/her mother, but untouched `X` or `Y` from father.

As of version 0.7.5, no other operator recognize sex chromosome. Most notably, `stat` counts allele frequencies etc regardless sex chromosome and can not count allele frequency for `X` or `Y` separately.

3.11 Pedigree tracking

simuPOP provides the following functions to manipulate pedigrees

- If you set `ancestralDepth` of a population to a positive number (default 0), `ancestralDepth` number of ancestral generations will be saved to the population, which makes a total of `ancestralDepth + 1` generations.
- You can use `population::useAncestralPop(idx)` to use current (0), parental (1), grand-parental (2) generations etc. Just remember to call `population::useAncestralPop(0)` to set current generation back.
- You can set `ancestralDepth` dynamically using operator `setAncestralDepth`. Usually, this operator is called, for example, as `setAncestralDepth(at=[-2])`, to allow last several generations to be saved at the end of evolution.
- No parental information is saved by default we usually do not know the parents of an offspring. This can be changed by using the `father_idx` and `mother_idx` information fields, and an appropriate tagger such as `parentTagger()`, which is a during mating operator that will record the parents' indices in the parental generation to offspring's information fields.

- `randomMating()` only produce one offspring per mating event. This makes full siblings very unlikely. You usually need to change this at the last several generations.

You can see that generating multi-generation populations are quite different from the usual evolutionary process where random mating is used, and one offspring is generated for each mating event. In practice (see `scripts/simuComplexDisease.py`), if we need to prepare a population for pedigree sampling, we can run a simulator like this

Example 3.14: One-stage simulation for pedigree tracking

```
pop = population(...., ancestralDepth=2,
    infoFields=['father_idx', 'mother_idx'])
simu = simulator(pop, randomMating(numOffspring=2))
simu.evolve(
    preOps=[...],
    ops = operators,
    end = 1000
)
```

The problem with this approach is that two generations are saved at all generations, and all mating events produce two offspring. The former is not a big deal but the latter will reduce effective population size of the resulting population. To avoid these problems, a two-stage simulation can be done

Example 3.15: Two-stage simulation for pedigree tracking

```
pop = population(....)
simu = simulator(pop, randomMating())
simu.evolve(
    preOps=[...],
    ops = operators,
    end = 1000 - 2
)
simu.setAncestralDepth(2)
simu.addInfoFields(['father_idx', 'mother_idx'])
simu.setMatingScheme(randomMating(numOffspring=2))
operators.append(parentsTagger())
simu.evolve(ops=operators, end=2)
```

That is to say, we separate the simulation into two parts. The first part is geared toward performance and maximum effective population size (use true random mating), and the second part is tweaked for the final multi-generation population. Note that `setAncestralDepth` and `addInfoFields` should be done at the simulator level so that every replicates in the simulator have the same new information fields. `simu.population(0).addInfoFields(['father_idx', 'mother_idx'])` will compromise the integrity of the simulator and is disallowed. (Integrity refers to the fact that all populations in a simulator should have the same genotypic structure as the simulator).

Now, at the end of the simulation, you get a population with multiple generations, with parental information. But it is still not easy to obtain pedigrees. As a matter of fact, since individuals can belong to multiple pedigrees, it is not even easy to define a pedigree. simuPOP provides a few pedigree ascertainment operators

- `AffectedSibpairSample`: sample affected sibpairs, along with their parents from a population. Affection status should have been set by other means such as a penetrance operator.
- `LargePedigreeSample`: sample grand parents, their children, and the spouse and children of them. Affection status is ignored, although the minimal number of affected individuals in each family can be specified.
- `NuclearFamilySample`: sample two-generation pedigrees.

If you need to sample more complicated pedigrees, you should first use `sample::findOffspringAndSpouse` to locate each individual's offspring and spouse, then use `useAncestralPop()` to go through the generations and set `pedIndex` for the pedigree you choose, and then use `setSubPopID()`, `newPopByIndID()` to exclude and remove unneeded individuals. `sample::resetParentalIndex()` should also be used to reset the `father_idx` and `mother_idx` fields. Sound complicated? It is complicated! I hope that I can get some better idea and make this process a bit easier, but this is where simuPOP is at right now.

Finally, you can save the sample populations in a pedigree-aware format like Linkage or Merlin/QTDT format. simuPOP can do this easily for you.

3.12 Save and load to other formats

simuPOP data structure is open in that many functions are provided to access every aspect of the population. This makes it easy to save and load populations in other formats. As an example, I will explain `SaveTDT` function in detail here, which is available in `simuUtil.py`.

Although all file formats have different characteristics, `simuPOP` tries to provide a uniform interface to them. Common parameters are

- pop: population to save, can be a file name, or a file object (loaded simuPOP population)
- output and outputExpr: output is the base filename, and outputExpr should be evaluated from pop's local namespace.
- loci: loci to output, default is [], meaning output all loci
- fields: information fields to output.
- combine: a python function, if given, used to combine two alleles at the same locus. For example

  ```
  def comb(geno):
      return geno[0]+geno[1]+1
  ```

  returns 1 for genotype(0, 0), 2 for genotype (0, 1) and so on.
- shift: default to 1. simuPOP uses 0 based allele and many formats use 1 based allele. Setting shift=1 output (1,2) for genotype (0,1).

The `Merlin/QTDT` format uses several files to store genotype and phenotype information. Namely a `.dat` file for phenotype, `.map` file for chromosome structure, and `.ped` for pedigree. The population given must have `pedindex, father_idx` and `mother_idx` information fields to indicate family id and parents of each individual. These information fields will be available if the sample is obtained from `affectedSibpairSample` or `largePedigreeSample` operators.

The first part of the function is the usual housekeeping part (see example 3.16). It loads population if `pop` is a name, evaluate `outputExpr` if needed, and open the files to write. This part is likely to be similar for all such functions.

Example 3.16: Function SaveQTDT, part one

```
def SaveQTDT(pop, output='', outputExpr='', loci=[],
        fields=[], combine=None, shift=1, **kwargs):
    """ save population in Merlin/QTDT format. The population must have
        pedindex, father_idx and mother_idx information fields.

        pop: population to be saved. If pop is a filename, it will be loaded.
```

```
        output: base filename.
        outputExpr: expression for base filename, will be evaluated in pop's
            local namespace.

        loci: loci to output

        fields: information fields to output

        combine: an optional function to combine two alleles of a diploid
            individual.

        shift: if combine is not given, output two alleles directly, adding
            this value (default to 1).
    """
    if type(pop) == type(''):
        pop = LoadPopulation(pop)
    if output != '':
        file = output
    elif outputExpr != '':
        file = eval(outputExpr, globals(), pop.vars())
    else:
        raise exceptions.ValueError, "Please specify output or outputExpr"
    # open data file and pedigree file to write.
    try:
        datOut = open(file + ".dat", "w")
        mapOut = open(file + ".map", "w")
        pedOut = open(file + ".ped", "w")
    except exceptions.IOError:
        raise exceptions.IOError, "Can not open file " + file + " to write."
    if loci == []:
        loci = range(0, pop.totNumLoci())
```

Part two of the code (example 3.17) output data file. There are three kinds of phenotype, affection status, trait and markers. We determine if a user wants to output affection from the `fields` parameter. We remove `affection` from `fields` because `affection` is not a real information field (that can be retrieved by `info()` function). You can learn how to use the `locusName` function from this part.

Example 3.17: Function SaveQTDT, part two

```
    # write dat file
    #
    if 'affection' in fields:
        outputAffectation = True
        fields.remove('affection')
        print >> datOut, 'A\taffection'
    else:
        outputAffectation = False
    for f in fields:
        print >> datOut, 'T\t%s' % f
    for marker in loci:
        print >> datOut, 'M\t%s' % pop.locusName(marker)
    datOut.close()
```

Part three (example 3.18) of the function output a map file. We need to know the chromosome number (+1 to use 1 based index), locus name and locus position, all of which can be retrieved from simple `simuPOP` functions. Note that if locus name, position are not given explicitly when a population is created, they all have default values.

Example 3.18: Function SaveQTDT, part three

```
    # write map file
    print >> mapOut, 'CHROMOSOME MARKER POSITION'
    for marker in loci:
        print >> mapOut, '%d\t%s\t%f' % (pop.chromLocusPair(marker)[0] + 1,
            pop.locusName(marker), pop.locusPos(marker))
    mapOut.close()
```

The next part (example 3.19) prepares pedigree output. It determines the code to output for sex and affection status. These are likely to be different from format to format so we define explicitly here. The `writeInd` output the line for one individual, given family id, id, father and mother. For QTDT format, two alleles of a genotype are outputted separately so the `combine` parameter is ignored.

Example 3.19: Function SaveQTDT, part four

```
    # write ped file
    def sexCode(ind):
        if ind.sex() == Male:
            return 1
        else:
            return 2
    # disease status: in linkage affected is 2, unaffected is 1
    def affectedCode(ind):
        if ind.affected():
            return 'a'
        else:
            return 'u'
    #
    pldy = pop.ploidy()
    def writeInd(ind, famID, id, fa, mo):
        print >> pedOut, '%d %d %d %d %d' % (famID, id, fa, mo, sexCode(ind)),
        if outputAffectation:
            print >> pedOut, affectedCode(ind),
        for f in fields:
            print >> pedOut, '%.3f' % ind.info(f),
        for marker in loci:
            for p in range(pldy):
                print >> pedOut, "%d" % (ind.allele(marker, p) + shift),
        print >> pedOut
```

The last part of the code (example 3.20) look most complicated. It first get the `pedindex` information field of the whole population, and figure out how many pedigrees to output. Then, it go from ancestral generation 2, 1, 0 and look for individuals within each pedigree. A map is used to map absolute index to within pedigree index. Of course, this part would be easier if you do not need to handle pedigree, for example, when outputting case control samples.

Example 3.20: Function SaveQTDT, part five

```
    # number of pedigrees
    # get unique pedgree id numbers
    from sets import Set
    peds = Set(pop.indInfo('pedindex', False))
    # do not count peds -1
    peds.discard(-1)
    #
    newPedIdx = 1
    #
    for ped in peds:
        id = 1
        # -1 means no parents
        pastmap = {-1:0}
        # go from generation 2, 1, 0 (for example)
        for anc in range(pop.ancestralDepth(), -1, -1):
            newmap = {-1:0}
            pop.useAncestralPop(anc)
            # find all individual in this pedigree
            for i in range(pop.popSize()):
                ind = pop.individual(i)
                if ind.info('pedindex') == ped:
                    dad = int(ind.info('father_idx'))
                    mom = int(ind.info('mother_idx'))
                    if dad == mom and dad != -1:
                        print ("Something wrong with pedigree %d, father and mother
                            "idx are the same: %s") % (ped, dad)
                    writeInd(ind, newPedIdx, id, pastmap.setdefault(dad, 0), \
                        pastmap.setdefault(mom, 0))
                    newmap[i] = id
                    id += 1
            pastmap = newmap
        newPedIdx += 1
    pedOut.close()
```

3.13 Gene mapping

Once you output your sample into a format that can be processed by other applications, you can handle them in whatever way you want. If you are interested in processing the data in simuPOP (actually, in python), you can use python to call these programs.

Example 3.21: Example of gene mapping

```
def VC_merlin(file, merlin='merlin'):
    ''' run variance component method
        file: file.ped, file.dat, file.map and file,mdl are expected.
            file can contain directory name.
    '''
    cmd = 'merlin -d %s.dat -p %s.ped -m %s.map --pair --vc' % (file, file, file)
    resline = re.compile('\s+([\d.+-]+|na)\s+([\d.+-]+|na)%\s+([\d.+-]+|na)\s+([\d.+
    print "Running", cmd
    fout = os.popen(cmd)
    pvalues = []
    for line in fout.readlines():
        try:
            # currently we only record pvalue
            (pos, h2, chisq, lod, pvalue) = resline.match(line).groups()
            try:
                pvalues.append(float(pvalue))
            except:
                # na?
                pvalues.append(-1)
        except AttributeError:
            pass
    fout.close()
    return pvalues
```

An example is given in example 3.21. In this function, `merlin` [Abecasis et al., 2002] is called to process file produced by the `SaveQTDT` function. The output is fed into a pipe (`popen`) and be filtered by the python `re` (regex) module. Only the p-values are obtained and returned.

Chapter 4

A real example

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

4.0.1 Simulation scenario

Reich and Lander [2001] proposed a population genetics framework to model the evolution of allelic spectra (the number and population frequency of alleles at a locus). The model is based on the fact that human population grew quickly from around 10,000 to 6 billion in 18,000 -150,000 years. His analysis showed that at the founder population, both common and rare diseases have simple spectra. After the sudden expansion of population size, the allelic spectra of simple diseases become complex; while those of complex diseases remained simple.

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

Demographic model

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

Mutation model

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

Selection on a common and a rare disease

Two diseases are simulated: a common disease with initial allele frequency of $f_0 = 0.2$; and a rare disease with initial allele frequency of $f_0 = 0.001$. The diseases are unlinked in the sense that their corresponding loci reside on separated

chromosomes. The allelic spectra of both diseases are set to be $[.9, .02, .02, .02, .02, .02]$. I.e., one allele accounts for 90% of the disease cases.

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

These parameters, translated to python, are shown in 4.1

Example 4.1: Set parameters

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

# allele spectrum
C_f = [1-C_f0] + [x*C_f0 for x in [0.9, 0.02, 0.02, 0.02, 0.02, 0.02]]
R_f = [1-R_f0] + [x*R_f0 for x in [0.9, 0.02, 0.02, 0.02, 0.02, 0.02]]
```

4.0.2 Create a simulator

Several parameters are needed to create a population:

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

You can then create a population with:

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

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

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

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

With a little adjustment of how population size is given to `population()`, and use demographic function as a parameter to allow other demographic models to be used, we end up with example 4.2. Note that because we use loci with more than 255 allele states, the long allele module is used.

Example 4.2: Create a simulator

```
from simuOpt import setOptions
setOptions(alleleType='long')
from simuPOP import *

# instantaneous population growth
def ins_exp(gen, oldSize=[]):
    if gen < burnin:
        return [initSize]
    else:
        return [finalSize]

def simulate(incScenario):
    simu = simulator(
# create a simulator
        population(subPop=incScenario(0), loci=[1,1],
            infoFields=['fitness']),
# inital population
        randomMating(newSubPopSizeFunc=incScenario)
# random mating
    )

simulate(ins_exp)
```

4.0.3 Initialization

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

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

4.0.4 Mutation and selection

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

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

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

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

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

  and

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

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

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

With these operators, the simulator can be started. It first initialize a population with given allelic spectra, and then evolve it, subject to mutation and selection, specific to each locus. The program is listed in example 4.3:

Example 4.3: Run the simulator

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

simulate(ins_exp)
```

4.0.5 Output statistics

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

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

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

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

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

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

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

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

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

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

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

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

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

Then, you can use this function in a python operator

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

The biggest difference between `pyEval` and `pyOperator` is that `pyOperator` is no longer evaluated in the population's local namespace. You will have to get the vars explicitly using the `pop.dvars()` function. (This also implies that you can do whatever you want to the population.). In this example, the function form of the `stat` operator is used to explicitly calculate allele frequency. The results are also explicitly printed using the `print` command. The explicities lead to longer, but clearer program. This becomes obvious when you need to calculate and print many statistics.

The following program (listing 4.4) uses the `pyOperator` solution. In this program, user can input two demographic models as command line parameter. Two other operators are used

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

Example 4.4: The whole program

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

'''
simulation for Reich(2001):
     On the allelic spectrum of human disease

'''

import simuOpt
simuOpt.setOptions(alleleType='long', optimized=False)
from simuPOP import *

import sys

initSize =  10000            # initial population size
finalSize = 1000000          # final population size
burnin = 500                 # evolve with constant population size
endGen = 1000                # last generation
mu = 3.2e-5                  # mutation rate
C_f0 = 0.2                   # initial allelic frequency of *c*ommon disease
R_f0 = 0.001                 # initial allelic frequency of *r*are disease
max_allele = 255             # allele range 1-255 (1 for wildtype)
C_s = 0.0001                 # selection on common disease
R_s = 0.9                    # selection on rare disease

C_f = [1-C_f0] + [x*C_f0 for x in [0.9, 0.02, 0.02, 0.02, 0.02, 0.02]]
R_f = [1-R_f0] + [x*R_f0 for x in [0.9, 0.02, 0.02, 0.02, 0.02, 0.02]]

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

```
# linear growth after burn-in
def lin_exp(gen, oldSize=[]):
    if gen < burnin:
        return [initSize]
    elif gen % 10 != 0:
        return oldSize
    else:
        incSize = (finalSize-initSize)/(endGen-burnin)
        return [oldSize[0]+10*incSize]

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


def simulate(incScenario):
    simu = simulator(                       # create a simulator
        population(subPop=incScenario(0), loci=[1,1],
            infoFields=['fitness']),        # inital population
        randomMating(newSubPopSizeFunc=incScenario)
    )
    simu.evolve(                            # start evolution
        preOps=[                            # operators that will be applied before evol
            # initialize locus 0 (for common disease)
            initByFreq(atLoci=[0], alleleFreq=C_f),
            # initialize locus 1 (for rare disease)
            initByFreq(atLoci=[1], alleleFreq=R_f),
        ],
        ops=[                               # operators that will be applied at each gen
            # mutate: k-alleles mutation model
            kamMutator(rate=mu, maxAllele=max_allele),
            # selection on common and rare disease,
            mlSelector([                # multiple loci - multiplicative model
                maSelector(locus=0, fitness=[1,1,1-C_s], wildtype=[0]),
                maSelector(locus=1, fitness=[1,1,1-R_s], wildtype=[0])
            ], mode=SEL_Multiplicative),
            # report generation and popsize and total disease allele frequency.
            pyOperator(func=ne, step=5),
            # monitor time
            ticToc(step=100),
            # pause at any user key input (for presentation purpose)
            pause(stopOnKeyStroke=1)
        ],
```

```
        end=endGen
    )

if __name__ == '__main__':
    if len(sys.argv) != 2:
        print 'Please specify demographic model to use.'
        print 'Choose from lin_exp and ins_exp'
        sys.exit(0)
    if sys.argv[1] == 'lin_exp':
        simulate(lin_exp)
    elif sys.argv[1] == 'ins_exp':
        simulate(ins_exp)
    else:
        print 'Wrong demographic model'
        sys.exit(1)
```

4.0.6 Option handling

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

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

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

  to view help information.

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

```
> simuLDDecay.py -s 500 -e 10 --recRate 0.1 --numRep 5 --noDialog
```

- use the optimized module by

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

save the parameters to a config file

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

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

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

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

to load parameters from the config file.

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

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

1. First line:

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

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

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

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

3. Define an option data structure.

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

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

4. Main simulation functions

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

You will notice that `simuOpt` does all the housekeeping things for you, including parameter reading, conversion, validation, print usage, save configuration file. Since most of the parts are pretty standard, you can actually copy any of the scripts under the `scripts` directory as a template for your new script. The following example 4.5 shows the beginning and the execution part of the complete `reich.py` script, which can be found under the doc directory. For a complete reference of the Options structure, please refer to the reference manual.

Example 4.5: Option handling

```
options = [
    {'arg': 'h',
     'longarg': 'help',
     'default': False,
     'description': 'Print this usage message.',
     'allowedTypes': [types.NoneType, type(True)],
     'jump': -1                    # if -h is specified, ignore any other parameters.
    },
    {'longarg': 'initSize=',
     'default': 10000,
     'label': 'Initial population size',
     'allowedTypes': [types.IntType, types.LongType],
```

```
     'description': '''Initial population size. This size will be maintained
               till the end of burnin stage''',
     'validate': simuOpt.valueGT(0)
    },
    {'longarg': 'finalSize=',
     'default': 1000000,
     'label': 'Final population size',
     'allowedTypes': [types.IntType, types.LongType],
     'description': 'Ending population size (after expansion.',
     'validate': simuOpt.valueGT(0)
    },
    {'longarg': 'burnin=',
     'default': 500,
     'label': 'Length of burn-in stage',
     'allowedTypes': [types.IntType],
     'description': 'Number of generations of the burn in stage.',
     'validate': simuOpt.valueGT(0)
    },
    {'longarg': 'endGen=',
     'default': 1000,
     'label': 'Last generation',
     'allowedTypes': [types.IntType],
     'description': 'Ending generation, should be greater than burnin.',
     'validate': simuOpt.valueGT(0)
    },
    {'longarg': 'growth=',
     'default': 'instant',
     'label': 'Population growth model',
     'description': '''How population is grown from initSize to finalSize.
               Choose between instant, linear and exponential''',
     'chooseOneOf': ['linear', 'instant'],
    },
    {'longarg': 'name=',
     'default': 'cdcv',
     'allowedTypes': [types.StringType],
     'label': 'Name of the simulation',
     'description': 'Base name for configuration (.cfg) log file (.log) and figures (.ep
    },
]

def getOptions(details=__doc__):
    # get all parameters, __doc__ is used for help info
    allParam = simuOpt.getParam(options,
        'This program simulates the evolution of a common and a rare direse\n' +
        'and observe the evolution of allelic spectra\n', details)
    #
    # when user click cancel ...
    if len(allParam) == 0:
        sys.exit(1)
    # -h or --help
    if allParam[0]:
        print simuOpt.usage(options, __doc__)
        sys.exit(0)
    # automatically save configurations
```

```
    name = allParam[-1]
    if not os.path.isdir(name):
        os.makedirs(name)
    simuOpt.saveConfig(options, os.path.join(name, name+'.cfg'), allParam)
    # return the rest of the parameters
    return allParam[1:-1]

#
# IGNORED
#

if __name__ == '__main__':
    # get parameters
    (initSize, finalSize, burnin, endGen, growth) = getOptions()
    #
    from simuPOP import *
    #
    if initSize > finalSize:
        print 'Initial size should be greater than final size'
        sys.exit(1)
    if burnin > endGen:
        print 'Burnin gen should be less than ending gen'
        sys.exit(1)
    if growth == 'linear':
        simulate(lin_exp)
    else:
        simulate(ins_exp)
```

Chapter 5

Introduction to bundled scripts

Several scripts are bundled with simuPOP, under the `/usr/share/simuPOP/scripts` directory under a *nix system and `c:\python25\share\simuPOP\scripts` under windows. These scripts all use `simuOpt` module to organize help messages so you can get detailed information about the scripts and the parameter used by clicking on help button of the parameter dialog, or use commands like '`simuComplexDisease.py -h`' to get the help messages.

In this chapter, I will briefly explain what these scripts do, from a more methodology side of view. Be warned, though, that these scripts are less actively maintained than simuPOP core and I mostly rely on user bug report to identify problems in these scripts.

5.1 Examples and teaching scripts

5.1.1 `simuLDDecay.py`

This is the simplest script under the `scripts` directory, showing the decay of linkage disequilibrium under recombination. It is intended to be a template for many more such simulations for teaching a population genetics course.

5.1.2 `demoPyOperator.py`

This script demonstrate the use of a during-mating pure-Python operator. Since such operator will be called very frequently (at each mating event), the performance of such operators tend to be bad. Since most of the task performed by such an operator can be achieved by other means (for example a post-mating operator), it is rarely used.

5.2 Utility scripts

These scripts are not necessarily written in simuPOP. It is written to facilitate the use of simuPOP.

5.2.1 `simuViewPop.py`

`simuViewPop.py` is a `wxPython` application written to view simuPOP populations. You will need to have `wxPython` installed to use it. There are two ways to use this script:

- Import this script and call `viewPop(pop)` to view population `pop`

- Run from command line

```
$ simuViewPop.py /path/to/population.txt
```

This script shows four tabs to show the information of a population

- basic information
- a table view of all genotype
- calculation of statistics, with a tree-view of local name space
- save to other formats

5.2.2 simuCluster.py

`simuCluster.py` helps you manage a large number of simulations on a cluster system. You only need to maintain a single job-description file and `simuCluster.py` will help you submit them. The command line options are

```
$ simuCluster.py -l simulation.lst -a  -r -f key=val jobs
```

where

- `-l (--list) list:` a list file (actually a python file) that specifies variale `script` and `joblist`
- `-a (--all):` use all jobs defined in the list file
- `-r (--run):` run the jobs, by default, this script will only list the jobs and generate job file.
- `-p (--repeat) n:` execute command `n` times.
- `-f (--force):` force the execution even if the generated job scripts have `$` character.
- `key=val:` additional subsitution key/value pair that will be used to replace `$key` in the job scripts. Commonly used, or machine-specific, `key=val` pairs can be defined in a configuration file `$HOME/.simuCluster` with content like:

```
command = 'bsub -J $name <'
queue = 'batch'
job_dir = '/scratch/jobs'
```

- `job:` a list of jobs, a simple form of regular expression can be used. Namely, `job1_3` means `job1`, `job2` and `job3`.

The list file can be any python script, that defines variables `script` and `joblist` after execution, where `script` is a simple script with variables `$name` or `${name}`. and `joblist` is a string with lines of comma (can be other charater if you define a variable `separator`) separated fields, that will be used to replace `$0` (also `$name`, the name of a job), `$1, $2, ...` etc.

Then, what `simuCluster.py` will do is process this list file, replace `$name, $var, $1, $2 ...` etc with environmental variables, command line paramters, configuration file and joblist and generate job scripts. If `-r` is given, the job will be submitted. Example 5.1 gives a sample job list file. Command

```
$ python scripts/simuCluster.py -l joblist.lst -a
```

will generate files `job1.pbs`, ... and if `-r` option is given, these files will be submitted using `qsub job1`, unless you specify another `command` variable.

Example 5.1: A sample job list file

```
# list file for some simulations, should be processed by
# scripts/simuCluster.py

script = r"""
#!/bin/bash
#PBS -S /bin/bash
#PBS -N $name
#PBS -q $queue
#PBS -l walltime=$time:00:00
#PBS -o $job_dir
#PBS -e $job_dir
#
PYTHONPATH=/home/user/PythonModules/lib64/python2.3/site-packages
export PYTHONPATH
cd $job_dir
[ -d $job_dir ] || mkdir -p $job_dir
/bin/rm -rf $job_dir/$name
/bin/mkdir -p $job_dir/$name
python /home/bpeng/simuPOP/scripts/simuComplexDisease.py --noDialog --optimized \
      --simuName=$name --numChrom=5 --DSL='[5, 15, 25, 25, 45]' \
      --splitGen=8000 --numSubPop=1 \
      --fitness=$1 --alleleDistInSubPop=even \
      --recRate="0.0005" --curAlleleFreq='[0.2]*5' --numLoci="10" --DSLLoc="(0.5)"
\
      --initSize="10000" --endingSize="200000" --burninGen="5000" --markerType="SNP"
      --growthModel='exponential' --mixingGen="10000" --endingGen="10000" --savePop=
      --minMutAge=0 --maxMutAge=0 --migrRate="0." --migrModel='stepping stone'\
      --selMultiLocusModel="additive" --mutaRate=$2" --saveFormat='txt'
"""

joblist = ''

idx = 0
for fit in [0.001, 0.0005]:
    for mut in [0.0001, 0.00001]:
        joblist += 'jobs_%d: %s*5: %f\n' % (idx, [1, 1+fit/2., 1+fit], mut)
```

5.2.3 simuUtil.py

`simuUtil.py` is a standard part of simuPOP and is installed along with `simuPOP.py` (other utility scripts are installed under `scripts` directory). These function include

1. extra python operators, the two potentially useful ones are

   - **tab**
   - **endl**

   These two operators output, as their names suggest, `'\t'` and `'\n'`.

2. Pre-defined demographic functions:

- `constSize`
- `LinearExpansion`
- `ExponentialExpansion`
- `InstantExpansion`

These functions return a demographic function with given event times.

3. Pre-defined migration rate functions

   - `MigrIslandRates`
   - `MigrSteppingStoneRates`

   These functions return a migration matrix of given migration model and parameter.

4. Save and load from other formats

   - `SaveFstat (saveFstat), LoadFstat`
   - `LoadGCData`
   - `SaveLinkage (saveLinkage), LoadLinkage`
   - `SaveQTDT`
   - `SaveCSV`

   These functions save and load simuPOP populations in various formats.

5. Gene mapping functions

   - `TDT_gh, LOD_gh`
   - `ChiSq_test`
   - `LOD_merlin, VC_merlin`
   - `Sibpair_TDT_gh, Sibpair_LOD_gh`
   - `Sibpair_LOD_merlin, QtraitSibs_Reg_Merlin, QtriatSibs_VC_merlin`
   - `LargePeds_Reg_merlin, LargePeds_VC_merlin`

   These functions call GENEHUNTER or MERLIN to map disease genes. Various parameters like penetrance, quantitative trait functions, sample size are needed. These functions are tested only under Linux and are subject to frequent changes.

In general, these utility functions are provided as it is and you may need to read the source code to make it work should errors occur. Unit test will be added later when these functions are more or less stablized/standardized.

5.3 General simulation scripts

5.3.1 `simuCDCV.py`

This script is used to simulate the evolution of allelic spectra (number and allele freqencies of alleles at a locus) for monogenic or polygenic, rare or common diseases. The goal of the simulations is to validate the common disease common variant hypothesis[Lander, 1996]. I used this script to verify two theoretical models proposed by Pritchard [2001] and Reich and Lander [2001]. The results are published in Peng and Kimmel [2007].

5.3.2 simuRecHotSpots.py

I wrote this script to simulate the evolution of a chromosome, subject to recombination of uniform recombination rate. Using this script, I would like to see how many recombination hotspots can be observed if there is no physical recombination hotspots, i.e. actual variation of recombination rate on the chromosome. The population is saved in LDhat format to be analyzed by LDhat [Myers et al., 2005].

5.3.3 simuNeutralSNPs.py

This script is adapted from `simuRecHotSplots.py`, the main purpose is to observe the evolution of allele frequency under more complicated scenarios than classical population genetics theory can handle.

5.4 Simulations of the evolution of complex human diseases

5.4.1 simuForward.py

This script presents my first attempt to simulate the evolution of complex human diseases in a forward-time manner and generate samples for gene mapping purposes. The script goes like this:

- initialize a small (likely 10K) founder population with a few haplotypes
- burn-in this founder population for a few thousands generations to break down linkage disequilibrium
- after this stage, the population starts to expand. It can be split into several subpopulations (simulate human subpopulations), with and/or without migration and be merged back to a single population.
- At the beginning of population expansion, several disease mutants are introduced to the population. Positive or negative selection is applied to individuals with disease mutants. We hope to harvest a final population with certain disease allele frequency.

This process is problematic in that

- The disease allele can get lost
- We can not control the disease allele frequency at the last generation

To solve the first problem, I re-introduce disease mutants if they get lost. I also apply, optionally, strong positive selection pressure during an disease-introduction stage to artificially boost the disease allele frequency, until it reach a designed range of allele frequencies. If the disease allele still get lost after the disease introduction stage, the simulation will be restarted. By manipulating parameters like designed allele frequency and population size, the impact of genetic drift can be moderate and give me a final population with designed disease allele frequency. This simulation scenario roughly follows that of Calafell et al Calafell et al. [2001].

To save simulation time, population at the end of the burnin stage is reused if simulation gets restarted.

5.4.2 simuComplexDisease.py

The previous simulation scenario is not satisfactory in that

- The age of mutant is fixed, but they should be somehow random
- Mutants can get lost and the simulation needs to be restarted repeatedly. This problem can be severe if we simulate mutants under purifying selection.

- We still can not control the final disease allele frequency well. The variation of disease allele frequencies in the final generation makes fair comparison between gene mapping methods difficult.

Therefore, I propose a simulation method, which is still under review, that

- simulate, backward in time, the trajectory of disease allele frequencies. The age of mutant is determined by trajectory length, and is random.
- Then, the script simulate forward in time using a controlled random mating scheme that follow the pre-simulated disease allele trajectories. The resulting population will have exact designed allele frequency.

An obvious advantage of this approach is that the simulation does not have to be restarted, and the disease allele frequency at the last generation can be controlled exactly.

5.4.3 `analComplexDisease.py`

I use `simuComplexDiseas.py` to simulate many population under various genetic and demographic models. The resulting populations are analyzed by this script. The analyses involved are

- merlin variance component method [Abecasis et al., 2002, Amos, 1994]
- merlin regression [Sham et al., 2002]
- TDT [Spielman et al., 1993]
- Linkage, and
- Case control association study.

Bibliography

G R Abecasis, S S Cherny, W O Cookson, and L R Cardon. Merlin-repid analysis of dense genetic maps using sparse gene flow trees merlin-rapid analysis of dense genetic maps using merlin-rapid analysis of dense genetic maps using. *Nat Genet*, 30:97–101, 2002. 55, 74

C I Amos. Robust variance-components approach for assessing genetic linkage in pedigrees. *Am J Hum Genet*, 54: 535–543, 1994. 74

F. Calafell, E L Grigorenko, A A Chikanian, and K K Kidd. Haplotype evolution and linkage disequilibrium: A simulation study. *Hum Hered*, 51:85–96, 2001. 73

Eric S Lander. The new genomics: Global views of biology. *Science*, 274(5287):536–539, 1996. 72

S Myers, L Bottolo, C Freeman, G McVean, and P Donnelly. A fine-scale map of recombination rates and hotspots across the human genome. *Science*, 310(5746):247–8, 2005. 73

Bo Peng and Marek Kimmel. Simulations provide support for the common disease common variant hypothesis. *Genetics*, 175:1–14, 2007. 57, 72

Jonathan K Pritchard. Are rare variants responsible for susceptibility to complex diseases. *Am J Hum Genet*, 69: 124–137, 2001. 72

David E Reich and Eric S Lander. On the allelic spectrum of human disease. *Trends Genet*, 17(9):502–510, 2001. 57, 61, 72

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

P C Sham, S Purcell, S S Cherny, and G R Abecasis. Powerful regression-based quantitative-trait linkage analysis of general pedigrees. *Am J Hum Genet*, 71:238–253, 2002. 74

R S Spielman, R E McGinnis, and W J Ewens. Transmission test for linkage disequilibrium: the insulin gene region and insulin-dependent diabetes mellitus (iddm). *Am J Hum Genet*, 52:506–516, 1993. 74

Index