simuPOP User's Guide

Release 0.8.8 (Rev: 2189)

Bo Peng

December 2004

Last modified
December 23, 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?

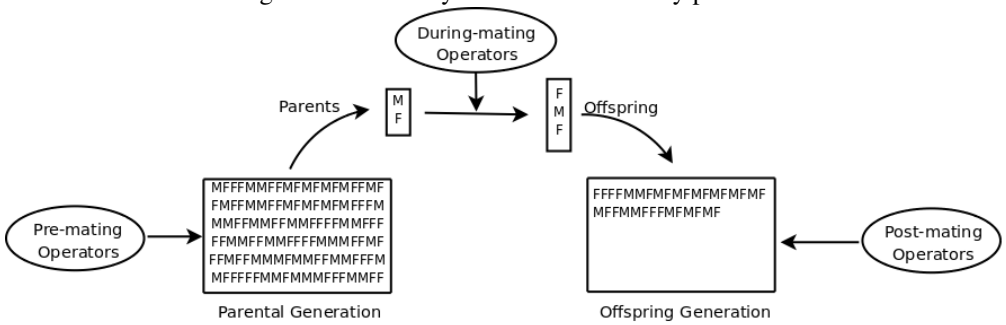
simuPOP is an individual-based forward-time population genetics simulation environment based on Python, a dynamic object-oriented programming language that has been widely used in biological studies. simuPOP provides a large number of Python objects and functions, and a mechanism to evolve populations 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
0.21    0.23    0.22
0.17    0.21    0.20
0.13    0.17    0.18
0.10    0.15    0.18
0.11    0.14    0.16
0.12    0.10    0.16
0.11    0.11    0.15
0.09    0.10    0.14
0.07    0.10    0.11
(100, 100, 100)
>>>
```

These concepts are demonstrated in Example 1.1, where a standard diploid Wright-Fisher model with recombination is simulated. The first line imports the standard simuPOP module. The second line creates a diploid population with 1000 individuals, each having one chromosome with two loci. The third line creates a simulator with three replicates of this population. Random mating will be used to generate offspring. The last statement uses the `evolve()` function to evolve the populations for 100 generations, subject to five operators.

The first operator `initByValue` is applied to all populations before evolution. This operator initializes all individuals with the same genotype `12/21`. The other operators can be applied at every generation. `recombinator` is a during-mating operator that recombines parental chromosomes with the given recombination rate `0.01` during the generation of offspring; `stat` calculates linkage disequilibrium between the first and second loci. The results of this operator are stored in a local variable space of each population. The last two operators `pyEval` and `pyOutput` are applied at the end of every 10 generations. `pyEval` is applied to all replicates to output calculated linkage disequilibrium values with a trailing tab, and the last operator outputs a newline after the last replicate. The result is a table of three columns, representing the decay of linkage disequilibrium of each replicate at 10 generation intervals. The

return value of the `evolve` function, which is the number of evolved generations for each replicate, is also printed.

1.3 Features

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

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

2. An arbitrary number of float numbers, called information fields, can be attached to individuals of a population. For example, information field father_idx and mother_idx are used to track an individual's parents, and pack_year can be used to simulate an environmental factor associated with smoking.

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

4. An operator can be native (implemented in C++) or hybrid (Python assisted). A hybrid operator calls a user-provided Python function to implement arbitrary genetic effects. For example, a hybrid mutator passes to-be-mutated alleles to a user-provided function and mutates these alleles according to the returned values.

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

6. Mating schemes and many operators can work on virtual subpopulations of a subpopulation. For example, positive assortative mating can be implemented by mating individuals with similar properties such as ancestry. The number of offspring per mating event can be fixed, or can follow a statistical distribution.

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

- The scripting interface gives simuPOP the flexibility to create arbitrarily complex evolutionary scenarios. For example, it is easy to use simuPOP to explicitly introduce a disease predisposing mutant to an evolving population, trace the allele frequency of them, and restart the simulation if they got lost due to genetic drift.

- The Python interface allows users to define customized genetic effects in Python. In contrast, other programs either do not allow customized effects or force users to modify code at a lower (e.g. C++) level.

- simuPOP is the only application that embodies the concept of virtual subpopulation that allows evolutions at a finer scale. This is required for realistic simulations of complex evolutionary scenarios.
- simuPOP allows users to examine an evolutionary process very closely because all simuPOP objects are Python objects that can be assessed using their member functions. For example, users can keep track of genotype at particular loci during evolution. In contrast, other programs work more or less like a black box where only limited types of statistics can be outputted.

1.4 Installation

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

If a binary distribution is unavailable for a specific platform, it is usually easy to compile simuPOP from source, following the standard "`python setup.py install`" procedure. Besides a C++ compiler, several supporting tools and libraries are needed. Please refer to the `INSTALL` file for further information.

Thanks to the 'glue language' nature of Python, it is easy to 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 How to read this user's guide

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

simuPOP is a comprehensive forward-time population genetics simulation environment with many unique features. If you are new to simuPOP, you can go through Chapter 2 and 3 quickly and understand what simuPOP is and what features it provide. Then, you can read Chapter 5 and learn how to apply simuPOP in real-world problems. After you play with simuPOP for a while and start to write simple scripts, you can study relevant sections in details. The *simuPOP reference manual* will become more and more useful when the complexity of your scripts grow.

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

- All classes (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(s) and virtual subpopulation(s)), `rep` (replicate), `gen` (generation), `ops` (operators), `expr` (expression), `stmts` (statements).

Chapter 2

Core 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 0.9.1svn (Revision 2188, Dec 23 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 0x1c46bce51b5c7a7e
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:551 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:446 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 2.14 demonstrates the use of ranges and iterators in simuPOP.

Example 2.5: Ranges and iterators

```
>>> pop = population(size=2, loci=[5, 6])
>>> InitByFreq(pop, [0.2, 0.3, 0.5])
>>> for ind in pop.individuals():
...     for loc in range(pop.chromBegin(1), pop.chromEnd(1)):
...         print ind.allele(loc),
...     print
...
1 0 2 0 1 0
0 2 0 2 2 2
>>>
```

2.2.4 `carray` datatype

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

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

Example 2.6: The carray datatype

```
>>> pop = population(size=2, loci=[3, 4])
>>> InitByFreq(pop, [0.3, 0.5, 0.2])
>>> ind = pop.individual(0)
>>> arr = ind.genotype()    # a carray to the underlying genotype
>>> geno = list(arr)        # a list of alleles
>>> print arr
[0, 1, 1, 1, 0, 2, 0, 1, 1, 1, 1, 1, 1, 1]
>>> print geno
[0, 1, 1, 1, 0, 2, 0, 1, 1, 1, 1, 1, 1, 1]
>>> arr.count(1)           # count
10
>>> arr.index(2)           # index
5
>>> ind.setAllele(5, 3)    # change underlying genotype using setAllele
>>> print arr              # arr is change
[0, 1, 1, 5, 0, 2, 0, 1, 1, 1, 1, 1, 1, 1]
```

```
>>> print geno              # but not geno
[0, 1, 1, 1, 0, 2, 0, 1, 1, 1, 1, 1, 1, 1]
>>> arr[2:5] = 4           # can use regular Python slice operation
>>> print ind.genotype()
[0, 1, 4, 4, 4, 2, 0, 1, 1, 1, 1, 1, 1, 1]
>>>
```

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

For efficiency considerations, simuPOP saves the same numbers of homologous sets of chromosomes even if some individuals have different numbers of homologous sets in a population. For example, in a haplodiploid population, because male individuals have only one set of chromosomes, their second homologous set of chromosomes are *unused*, which are labelled as `'_'`, as shown in Example 2.14.

Example 2.8: An example of haplodiploid population

```
>>> pop = population(size=[2,5], ploidy=Haplodiploid, loci=[3, 5])
>>> InitByFreq(pop, [0.3, 0.7])
>>> Dump(pop)
Ploidy: 2 (haplodiploid)
Chromosomes:
1: chrom1 (Autosome, 3 loci)
  loc1-1 (1), loc1-2 (2), loc1-3 (3)
2: chrom2 (Autosome, 5 loci)
  loc2-1 (1), loc2-2 (2), loc2-3 (3), loc2-4 (4), loc2-5 (5)
population size: 7 (2 subpopulations with 2, 5 individuals)
Number of ancestral populations: 0

Genotype of individuals in the present generation:
Subpopulation 0 (unnamed):
   0: MU 111 11001 | ___ _____
   1: FU 010 11111 | 110 11110
Subpopulation 1 (unnamed):
   2: MU 111 01110 | ___ _____
   3: FU 000 11110 | 110 11011
   4: MU 100 11111 | ___ _____
   5: MU 101 11111 | ___ _____
   6: FU 100 01101 | 110 11110
End of individual genotype.

>>>
```

2.3.2 Autosomes, sex chromosomes, and other types of chromosomes

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

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

Example 2.9: Different chromosome types

```
>>> pop = population(size=6, ploidy=2, loci=[3, 3, 6, 4, 4, 4],
...     chromTypes=[Autosome]*2 + [ChromosomeX, ChromosomeY] + [Customized]*2)
>>> InitByFreq(pop, [0.3, 0.7])
>>> Dump(pop, structure=False) # does not display genotypic structure information
```

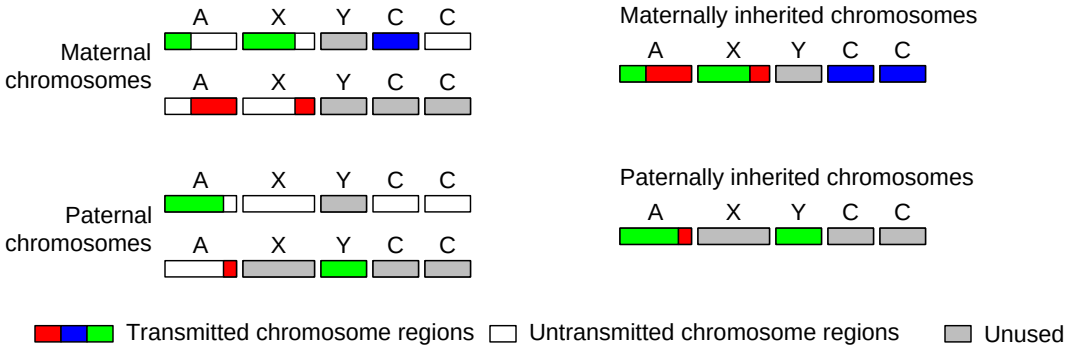
```
Genotype of individuals in the present generation:
Subpopulation 0 (unnamed):
   0: MU 110 000 111111 ____ 1111 0011 | 011 000 _______ 1101 1101 1111
   1: MU 011 011 111010 ____ 1100 0101 | 110 111 _______ 1011 1111 1110
   2: FU 010 111 110011 ____ 1010 1000 | 001 110 111111 ____ 0111 1111
   3: MU 111 111 111111 ____ 1110 0011 | 111 011 _______ 1010 1110 0001
   4: MU 111 111 111111 ____ 1011 1010 | 110 110 _______ 0000 0110 1110
   5: FU 111 001 111011 ____ 0001 1001 | 011 111 111111 ____ 1111 1001
End of individual genotype.

>>>
```

The evolution of sex chromosomes follow the following rules

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

As an advanced feature of simuPOP, chromosomes that do not follow the inheritance patterns of autosomes or sex chromosomes can be handled separately. Figure 2.1 depicts the possible chromosome structure of two diploid parents, and how offspring chromosomes are formed. It uses two customized chromosomes to model multiple copies of mitochondrial chromosomes that are passed randomly from mother to offspring. The second homologous copy of customized chromosomes are unused in this example.

Figure 2.1: Inheritance of different types of chromosomes in a diploid population



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

2.3.3 Information fields

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

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

Example 2.10: Basic usage of information fields

```
>>> pop = population(10, loci=[20], ancGen=1,
...     infoFields=['father_idx', 'mother_idx'])
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     preOps = [initByValue([0]*20+[1]*20)],
...     ops = [
...         parentsTagger(),
...         recombinator(rate=0.01)
...     ],
...     gen = 1
... )
(1,)
>>> pop = simu.extract(0)
>>> pop.indInfo('mother_idx')  # mother of all offspring
(7.0, 7.0, 4.0, 4.0, 8.0, 0.0, 2.0, 7.0, 6.0, 6.0)
>>> ind = pop.individual(0)
>>> mom = pop.ancestor(ind.intInfo('mother_idx'), 1)
>>> print ind.genotype(0)
[0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
>>> print mom.genotype(0)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> print mom.genotype(1)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>>
```

Example 2.14 demonstrates the basic usage of information fields. In this example, a population with two information fields `mother_idx` and `father_idx` are created. It can hold one ancestral generations (`ancGen=1`, see Section 2.5.5 for details) so the most recent parental generations will be kept in a population object. After initializing each individual with two chromosomes with all zero and all one alleles respectively, the population evolves one generation, suject to recombination at rate 0.01. Parents of each individual are recorded, by operator `parentsTagger`, to information fields `mother_idx` and `father_idx` of each offspring .

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

2.4 Individual

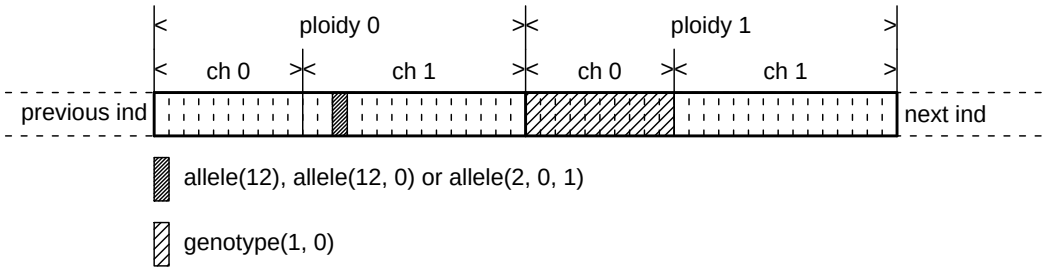
Individuals are building blocks of populations. An individual object cannot be created independently, but refences to inidividuals can be retrieved using member functions of a population object. In addition to structural information shared by all individuals in a population, the individual class provides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

Example 2.11: Access Individual properties

```
>>> pop = population([5, 4], loci=[2, 5], infoFields=['x'])
>>> # get an individual
>>> ind = pop.individual(3)
>>> ind.ploidy()            # access to genotypic structure
2
>>> ind.numChrom()
2
>>> ind.affected()
False
>>> ind.setAffected(True)   # access affection status,
>>> ind.sex()               # sex,
1
>>> ind.setInfo(4, 'x')     # and information fields
>>> ind.info('x')
4.0
>>>
```

Genotypes of an individual are stored sequentially and can be accessed locus by locus, or in batch. The alleles are arranged by position, chromosome and ploidy. That is to say, the first allele on the first chromosome of the first homologous set is followed by alleles at other loci on the same chromsome, then markers on the second and later chromosomes, followed by alleles on the second homologous set of the chromosomes for a diploid individual. A consequence of this memory layout is that alleles at the same locus of a non-haploid individual are separated by `individual::totNumLoci()` loci. The memory layout of a diploid individual with two chromosomes is illustrated in Figure 2.2.

Figure 2.2: Memory layout of individual genotype



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

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

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

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

simuPOP provies several functions to read/write individual genotype. It is worth noting that, instead of copying genotypes of an individual to a Python tuple or list, the return value of function `genotype([p, [ch]])` is a special python carray object that reflects the underlying genotypes. Modifying elements of this array will change the genotype of an individual directly. Only `count` and `index` list functions can be used, but all comparison, assignment and slice operations are allowed. If you would like to copy the content of this `carray` to a Python list, use the `list()` function. Example 2.12 demonstrates the use of these functions.

Example 2.12: Access individual genotype

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

2.5 Population

The `population` object is the most important object of simuPOP. It consists of one or more generations of individuals, grouped by subpopulations, and a local Python dictionary to hold arbitrary population information. This class provides a large number of functions to access and modify population structure, individuals and their genotypes and information fields. The following sections explain these features in detail.

2.5.1 Subpopulations

A simuPOP population consists of one or more subpopulations. Subpopulations serve as barriers of individuals in the sense that mating only happens between individuals in the same subpopulation. A number of functions are provided to merge, remove, resize subpopulations, and move individuals between subpopulations (migration). You will rarely get a chance to use them directly because such operations are usually handled by operators.

Example 2.14 demonstrates how to use subpopulation related functions. Of particular interest is the `setSubPopByIndInfo()` function. This function takes an information field as parameter and rearrange indi-

viduals according to their values at this information field. Individuals with invalid (nagative) values at this information field are removed. This is essentially how migration is implemented in simuPOP.

Example 2.13: Manipulation of subpopulations

```
>>> pop = population(size=[3, 4, 5], ploidy=1, loci=[1], infoFields=['x'])
>>> # individual 0, 1, 2, ... will have an allele 0, 1, 2, ...
>>> pop.setGenotype(range(pop.popSize()))
>>> #
>>> pop.subPopSize(1)
4
>>> # merge subpopulations
>>> pop.mergeSubPops([1, 2])
>>> # split subpopulations
>>> pop.splitSubPop(1, [2, 7])
>>> pop.subPopSizes()
(3, 2, 7)
>>> # set information field to each individual's new subpopulation ID
>>> pop.setIndInfo([0, 1, 2, -1, 0, 1, 2, -1, -1, 0, 1, 2], 'x')
>>> # this manually triggers an migration, individuals with negative values
>>> # at this information field are removed.
>>> pop.setSubPopByIndInfo('x')
>>> Dump(pop, width=2, structure=False)
Genotype of individuals in the present generation:
Subpopulation 0 (unnamed):
   0: MU  0 |  0
   1: MU  4 |  0
   2: MU  9 |  0
Subpopulation 1 (unnamed):
   3: MU  1 |  1
   4: MU  5 |  1
   5: MU 10 |  1
Subpopulation 2 (unnamed):
   6: MU  2 |  2
   7: MU  6 |  2
   8: MU 11 |  2
End of individual genotype.

>>>
```

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

Example 2.14: Use of subpopulation names

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

```
('x', 'z-1', 'z')
>>> pop.subPopByName('z')
2
>>>
```

2.5.2 Virtual subpopulations

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

VSPs are defined by virtual splitters. A splitter defines the same number of VSPs in all subpopulations, although sizes of these VSPs vary across subpopulations due to subpopulation differences. For example, a `sexSplitter()` defines two VSPs, the first with all male individuals and the second with all female individuals, and a `infoSplitter(field='x', values=[1, 2, 4])` defines three VSPs whose members have values `1`, `2` and `4` at information field `x`, respectively. If different types of VSPs are needed, a combined splitter can be used to combine VSPs defined by several splitters.

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

Example 2.15: Define virtual subpopulations in a population

```
>>> import random
>>> pop = population(size=[200, 400], loci=[30], infoFields=['x'])
>>> # assign random information fields
>>> pop.setIndInfo([random.randint(0, 3) for x in range(pop.popSize())], 'x')
>>> # define a virtual splitter by information field 'x'
>>> pop.setVirtualSplitter(infoSplitter(field='x', values=[0, 1, 2, 3]))
>>> pop.numVirtualSubPop()     # Number of defined VSPs
4
>>> pop.subPopName([0, 0])     # Each VSP has a name
'unnamed - x = 0'
>>> pop.subPopSize([0, 0])     # Size of VSP 0 in subpopulation 0
46
>>> pop.subPopSize([1, 0])     # Size of VSP 0 in subpopulation 1
90
>>> # use a combined splitter that defines additional VSPs by sex
>>> InitSex(pop)
>>> pop.setSubPopName('subPop 1', 0)
>>> pop.setVirtualSplitter(combinedSplitter([
...     infoSplitter(field='x', values=[0, 1, 2, 3]),
...     sexSplitter()])
... )
>>> pop.numVirtualSubPop()     # Number of defined VSPs
6
>>> pop.subPopName([0, 4])     # VSP 4 is the first VSP defined by the sex splitter
'subPop 1 - Male'
>>> pop.subPopSize([0, 4])     # Number of male individuals
94
>>>
```

VSP provides an easy way to access groups of individuals in a subpopulation and allows finer control of an evolutionary process. For example, mating schemes can be applied to VSPs which makes it possible to apply different mating schemes to, for example, individuals with different ages. By applying migration, mutation etc to VSPs, it is easy to

implement advanced features such as sex-biased migrations, different mutation rates for individuals at different stages of a disease. Example 2.18 demonstrates how to initialize genotype and information fields to individuals in male and female VSPs.

Example 2.16: Applications of virtual subpopulations

```
>>> import random
>>> pop = population(10, loci=[2, 3], infoFields=['Sex'])
>>> InitSex(pop)
>>> pop.setVirtualSplitter(sexSplitter())
>>> # initialize male and females with different genotypes. Set initSex
>>> # to False because this operator will by default also initialize sex.
>>> InitByValue(pop, [[0]*5, [1]*5], subPop=([0, 0], [0, 1]), initSex=False)
>>> # set Sex information field to 0 for all males, and 1 for all females
>>> pop.setIndInfo([1], 'Sex', [0, 0])
>>> pop.setIndInfo([2], 'Sex', [0, 1])
>>> # Print individual genotypes, followed by values at information field Sex
>>> Dump(pop, structure=False)
Genotype of individuals in the present generation:
Subpopulation 0 (unnamed):
   0: FU 11 111 | 11 111 |  2
   1: FU 11 111 | 11 111 |  2
   2: FU 11 111 | 11 111 |  2
   3: FU 11 111 | 11 111 |  2
   4: FU 11 111 | 11 111 |  2
   5: MU 00 000 | 00 000 |  1
   6: MU 00 000 | 00 000 |  1
   7: FU 11 111 | 11 111 |  2
   8: MU 00 000 | 00 000 |  1
   9: MU 00 000 | 00 000 |  1
End of individual genotype.

>>>
```

2.5.3 Access individuals and their properties

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

If you would like to access a group of individuals, either from a whole population, a subpopulation, or from a virtual subpopulation, `population::individuals([subPop])` is easier to use. This function returns a Python iterator that can be used to iterate through individuals. An advantage of this function is that `subPop` can be a virtual subpopulation which makes it easy to iterate through individuals with certain properties (such as all male individuals).

If more then one generations are stored in a population, function `ancestor(idx, [subPop], gen)` can be used to access individual from an ancestral generation (see Section 2.5.5 for details). Because there is no group access function for ancestors, it may be more convenient to use `useAncestralGen` to make an *ancestral* generation the *current* generation, and use `population::individuals`. Note that ancestor() function can always access individuals at a certain generation, regardless which generation the current generation is. Example 2.18 demonstrates how to use all these individual-access functions.

Example 2.17: Access individuals of a population

```
>>> # create a population with two generations. The current generation has values
>>> # 0-9 at information field x, the parental generation has values 10-19.
>>> pop = population(size=[5, 5], loci=[2, 3], infoFields=['x'], ancGen=1)
>>> pop.setIndInfo(range(11, 20), 'x')
>>> pop1 = pop.clone()
```

```
>>> pop1.setIndInfo(range(10), 'x')
>>> pop.push(pop1)
>>> #
>>> ind = pop.individual(5)       # using absolute index
>>> ind.info('x')
5.0
>>> # use a for loop, and relative index
>>> for idx in range(pop.subPopSize(1)):
...     print pop.individual(idx, 1).info('x'),
...
5.0 6.0 7.0 8.0 9.0
>>> # It is usually easier to use an iterator
>>> for ind in pop.individuals(1):
...     print ind.info('x'),
...
5.0 6.0 7.0 8.0 9.0
>>> # Access individuals in VSPs
>>> pop.setVirtualSplitter(infoSplitter(cutoff=[3, 7], field='x'))
>>> for ind in pop.individuals([1, 1]):
...     print ind.info('x'),
...
5.0 6.0
>>> # Access individuals in ancetral generations
>>> pop.ancestor(5, 1).info('x')        # absolute index
16.0
>>> pop.ancestor(0, 1, 1).info('x')     # relative index
16.0
>>> # Or make ancestral generation the current generation and use 'individual'
>>> pop.useAncestralGen(1)
>>> pop.individual(5).info('x')         # absolute index
16.0
>>> pop.individual(0, 1).info('x')      # relative index
16.0
>>> # 'ancestor' can still access the 'present' (generation 0) generation
>>> pop.ancestor(5, 0).info('x')
5.0
>>>
```

Although it is easy to access individuals in a population, it is often more efficient to access genotypes and information fields in batch mode. For example, functions `genotype()` and `setGenotype()` can read/write genotype of all individuals in a population or (virtual) subpopulation, functions `indInfo()` and `setIndInfo()` can read/write certain information fields in a population or (virtual) subpopulation. The write functions work in a circular manner in the sense that provided values are reused if they are not enough to fill all genotypes or information fields. Example 2.18 demonstrates the use of such functions.

Example 2.18: Access individual properties in batch mode

```
>>> import random
>>> pop = population(size=[4, 6], loci=[2], infoFields=['x'])
>>> pop.setIndInfo([random.randint(0, 10) for x in range(10)], 'x')
>>> pop.indInfo('x')
(10.0, 8.0, 10.0, 7.0, 8.0, 2.0, 3.0, 4.0, 9.0, 3.0)
>>> pop.setGenotype([0, 1, 2, 3], 0)
>>> pop.genotype(0)
[0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
>>> pop.setVirtualSplitter(infoSplitter(cutoff=[3], field='x'))
>>> pop.setGenotype([0])    # clear all values
>>> pop.setGenotype([5, 6, 7], [1, 1])
>>> pop.indInfo('x', 1)
```

```
(8.0, 2.0, 3.0, 4.0, 9.0, 3.0)
>>> pop.genotype(1)
[5, 6, 7, 5, 0, 0, 0, 0, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6]
>>>
```

2.5.4 Information fields

Information fields are usually set during population creation, using the `infoFields` parameter of the population constructor. It can also be set or added using functions `setInfoFields, addInfoField` and `addInfoFields`. Sections 2.3.3 and 2.5.3 have demonstrated how to read and write information fields from an individual, or from a population in batch mode.

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

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

```
>>> pop = population(10)
>>> pop.setInfoFields(['a', 'b'])
>>> pop.addInfoField('c')
>>> pop.addInfoFields(['d', 'e'])
>>> pop.infoFields()
('a', 'b', 'c', 'd', 'e')
>>> #
>>> cIdx = pop.infoIdx('c')
>>> eIdx = pop.infoIdx('e')
>>> pop.setIndInfo([1], cIdx)
>>> for ind in pop.individuals():
...     ind.setInfo(ind.info(cIdx) + 1, eIdx)
...
>>> print pop.indInfo(eIdx)
(2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0)
>>>
```

2.5.5 Ancestral populations

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

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

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

Several member functions can be used to manipulate ancestral generations:

- `ancestralGens()` returns the number of ancestral generations stored in a population.
- `setAncestralDepth(depth)` resets the number of generations a population can store.

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

A typical use of ancestral generations is demonstrated in example 2.22. In this example, a population is created and is initialized with allele frequency 0.5. Its ancestral depth is set to 2 at the beginning of generation 18 so that it can hold parental generations at generation 18 and 19. The allele frequency at each generation is calculated and displayed, both during evolution using a `stat` operator, and after evolution using the function form this operator. Note that setting the ancestral depth at the end of an evolutionary process is a common practise because we are usually only interested in the last few generations.

Example 2.20: Ancestral populations

```
>>> simu = simulator(population(500, loci=[1]), randomMating())
>>> simu.evolve(
...     preOps = [initByFreq([0.5, 0.5])],
...     ops = [
...         # start recording ancestral generations at generation 18
...         setAncestralDepth(2, at=[-2]),
...         stat(alleleFreq=[0], begin=-3),
...         pyEval(r"'%.3f\n' % alleleFreq[0][0]", begin=-3)
...     ],
...     gen = 20
... )
0.528
0.519
0.522
(20,)
>>> pop = simu.population(0)
>>> # start from current generation
>>> for i in range(pop.ancestralGens(), -1, -1):
...   pop.useAncestralGen(i)
...   Stat(pop, alleleFreq=[0])
...   print '%d   %.3f' % (i, pop.dvars().alleleFreq[0][0])
...
2   0.528
1   0.519
0   0.522
>>> # restore to the current generation
>>> pop.useAncestralGen(0)
>>>
```

2.5.6 Add and remove loci

Several functions are provided to remove, add empty loci or chromosomes, and to merge loci or chromosomes from another population. They can be used to trim unneeded loci, expand exisiting population or merge two populations. Example 2.22 demonstrates how to use these populations.

Example 2.21: Add and remove loci and chromosomes

```
>>> pop = population(10, loci=[3], chromNames=['chr1'])
>>> # 1 1 1,
```

```
>>> pop.setGenotype([1])
>>> # 1 1 1, 0 0 0
>>> pop.addChrom(lociPos=[0.5, 1, 2], lociNames=['rs1', 'rs2', 'rs3'],
...     chromName='chr2')
>>> pop1 = population(10, loci=[3], chromNames=['chr3'],
...     lociNames=['rs4', 'rs5', 'rs6'])
>>> # 2 2 2,
>>> pop1.setGenotype([2])
>>> # 1 1 1, 0 0 0, 2 2 2
>>> pop.addChromFrom(pop1)
>>> # 1 1 1, 0 0 0, 2 0 2 2 0
>>> pop.addLoci(chrom=[2, 2], pos=[1.5, 3.5], names=['rs7', 'rs8'])
(7, 10)
>>> # 1 1 1, 0 0 0, 2 0 2 0
>>> pop.removeLoci([8])
>>> Dump(pop)
Ploidy: 2 (diploid)
Chromosomes:
1: chr1 (Autosome, 3 loci)
  loc1-1 (1), loc1-2 (2), loc1-3 (3)
2: chr2 (Autosome, 3 loci)
  rs1 (0.5), rs2 (1), rs3 (2)
3: chr3 (Autosome, 4 loci)
  rs4 (1), rs7 (1.5), rs6 (3), rs8 (3.5)
population size: 10 (1 subpopulations with 10 individuals)
Number of ancestral populations: 0

Genotype of individuals in the present generation:
Subpopulation 0 (unnamed):
   0: MU 111 000 2020 | 111 000 2020
   1: MU 111 000 2020 | 111 000 2020
   2: MU 111 000 2020 | 111 000 2020
   3: MU 111 000 2020 | 111 000 2020
   4: MU 111 000 2020 | 111 000 2020
   5: MU 111 000 2020 | 111 000 2020
   6: MU 111 000 2020 | 111 000 2020
   7: MU 111 000 2020 | 111 000 2020
   8: MU 111 000 2020 | 111 000 2020
   9: MU 111 000 2020 | 111 000 2020
End of individual genotype.

>>>
```

2.5.7 Population extraction

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

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

from that population is used. Example 2.22 demonstrates the use of this powerful function.

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

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

>>>
```

2.5.8 Population Variables

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

simuPOP provides two functions, namely `population::vars()` and `population::dvars()` to access a population dictionary. These functions return the same dictionary object but `dvars()` returns a wrapper class so that you can access this dictionary as attributes. For example, `pop.vars()['alleleFreq'][0]` is equivalent to `pop.dvars().alleleFreq[0]`. Because dictionary `subPop[spID]` is frequently used by operators to store variables related to a particular (virtual) subpopulation, function `pop.vars(subPop)` is provided as a shortcut to `pop.vars()['subPop'][spID]`. Example 2.23 demonstrates how to set and access Population variables.

Example 2.23: Population variables

```
>>> from pprint import pprint
>>> pop = population(100, loci=[2])
>>> InitByFreq(pop, [0.3, 0.7])
>>> print pop.vars()    # No variable now
{}
>>> pop.dvars().myVar = 21
>>> print pop.vars()
{'myVar': 21}
>>> Stat(pop, popSize=1, alleleFreq=[0])
>>> # pprint prints in a less messy format
>>> pprint(pop.vars())
{'alleleFreq': [[0.32500000000000001, 0.67500000000000004]],
 'alleleNum': [[65, 135]],
 'myVar': 21,
```

```
 'numSubPop': 1,
 'popSize': 100,
 'subPop': [{'alleleFreq': [[0.32500000000000001, 0.67500000000000004]],
             'alleleNum': [[65, 135]],
             'popSize': 100}],
 'subPopSize': [100],
 'virtualPopSize': [100]}
>>> # print number of allele 1 at locus 0
>>> print pop.vars()['alleleNum'][0][1]
135
>>> # use the dvars() function to access dictionary keys as attributes
>>> print pop.dvars().alleleNum[0][1]
135
>>> print pop.dvars().alleleFreq[0]
[0.32500000000000001, 0.67500000000000004]
>>>
```

It is important to understand that this dictionary forms a **local namespace** in which Python expressions can be evaluated. This is the basis of how expression-based operators work. For example, the `pyEval` operator in example 1.1 evaluates expression "`'%.2f\t' % LD[0][1]`" in each population's local namespace when it is applied to that population. This yields different results for different population because their LD values are different. In addition to Python expressions, Python statements can also be executed in the local namespace of a population, using the `stmts` parameter of the `pyEval` or `pyExec` operator. Example 2.29 demonstrates the use of a simuPOP terminator, which terminates the evolution of a population when its expression is evaluated as `True`. Note that The `evolve()` function of this example does not specify how many generations to evolve so it will stop only after all replicates stop. The return value of this function indicates how many generations each replicate has evolved.

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

```
>>> simu = simulator(population(100, loci=[1]),
...     randomMating(), 5)
>>> simu.evolve(
...     preOps = [initByFreq([0.5, 0.5])],
...     ops = [
...         stat(alleleFreq=[0]),
...         terminateIf('alleleFreq[0][0] == 0. or alleleFreq[0][0] == 1.')
...     ]
... )
(49, 295, 885, 491, 244)
>>>
```

2.5.9 Save and load a population

simuPOP populations can be saved to and loaded from disk files using `population::save(file)` member function and global function `LoadPopulation`. (Yes, it is `Load..` not `load..` because `LoadPopulation` is a global function.). Although files in any extension can be used, extension `.pop` is recommended.

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

Example 2.25: Save and load a population

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

```
>>> pop1 = LoadPopulation('sample.pop')
>>> pop1.chromName(0)
'chrom1'
>>> pop1.dvars().name
'my population'
>>>
```

2.6 Operators

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

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

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

2.6.1 Applicable stages and generations

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

Operators that are passed to the `ops` parameter of the `simulator::evolve` function are, by default, applied to all generations during an evolutionary process. This can be changed using the `begin`, `end`, `step` and `at` parameters. As their names indicate, these parameters control the starting generation (`begin`), ending generation (`end`), generations between two applicable generations (`step`), and an explicit list of applicable generations (`at`). Other parameters will be ignored if `at` is specified. It is worth noting that,, if the simulator has an ending generation, negative generations numbers are allowed. They are counted backward from the ending generation. For example, if a simulator starts at generation `0`, and the `evolve` function has parameter `gen=10`, the simulator will stop at the *beginning* of generation `10`. Generation `-1` refers to generation `9`, and generation `-2` refers to generation `8`, and so on. Example 2.29 demonstrates how to set applicable stages and generations of an operator. In this example, a population is initialized before evolution using a `initByFreq` operator. allele frequency at locus `0` is calculated at generation `80`, `90`, but not `100` because the evolution stops at the beginning of generation `100`. A `pyEval` operator outputs generation number and allele frequency at the end of generation `80` and `90`. Another `pyEval` operator outputs similar information at generation `90` and `99`, before and after mating. Note, however, because allele frequencies are only calculated twice, the pre-mating allele frequency at generation `90` is actually calculated at generation `80`, and the allele frequencies display for generation `99` are calculated at generation `90`. At the end of the evolution, the population is saved to a file using a `savePopulation` operator.

Example 2.26: Applicable stages and generations of an operator.

```
>>> simu = simulator(population(100, loci=[20]), randomMating())
>>> simu.evolve(
...     preOps = [initByFreq([0.2, 0.8])],
...     ops = [
...         stat(alleleFreq=[0], begin=80, step=10),
...         pyEval(r"'After gen %d: allele freq: %.2f\n' % (gen, alleleFreq[0][0])",
...             begin=80, step=10),
...         pyEval(r"'Around gen %d: allele Freq: %.2f\n' % (gen, alleleFreq[0][0])",
...             at = [-10, -1], stage=PrePostMating)
...     ],
...     postOps = [savePopulation(output='sample.pop')],
...     gen=100
... )
After gen 80: allele freq: 0.14
Around gen 90: allele Freq: 0.14
After gen 90: allele freq: 0.10
Around gen 90: allele Freq: 0.10
Around gen 99: allele Freq: 0.10
Around gen 99: allele Freq: 0.10
(100,)
>>>
```

2.6.2 Applicable populations

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

Parameter `rep` is used to control which replicate(s) an operator can be applied to. This parameter can be a list of replicate numbers or a single replicate number. Negative index is allowed where `-1` refers to the last replicate. This technique has been widely used to produce table-like output where a `pyOutput` outputs a newline when it is applied to the last replicate of a simulator. Example 2.29 demonstrates how to use this `rep` parameter.

Example 2.27: Apply operators to a subset of populations

```
>>> simu = simulator(population(100, loci=[20]), randomMating(), 5)
>>> simu.evolve(
...     preOps = [initByFreq([0.2, 0.8])],
...     ops = [
...         stat(alleleFreq=[0], step=10),
...         pyEval('gen', step=10, rep=0),
...         pyEval(r"'\t%.2f' % alleleFreq[0][0]", step=10, rep=(0, 2, -1)),
...         pyOutput('\n', step=10, rep=-1)
...     ],
...     gen=30,
... )
0       0.23    0.28    0.24
10      0.27    0.45    0.47
20      0.06    0.30    0.63
(30, 30, 30, 30, 30)
>>>
```

An operator can also be applied to specified (virtual) subpopulations. For example, an `initializer` can be applied to male individuals in the first subpopulation, and everyone in the second subpopulation using parameter `subPop=[(0,0), 1]`, if a virtual subpopulation is defined by individual sex. However, not all operators support this parameter, and even if they do, their intrepretation of parameter input may vary. Please refer to *the simuPOP reference manual* for details.

2.6.3 Operator output

All operators we have seen, except for the `savePopulation` operator in Example 2.26, write their output to the standard output, namely your terminal window. However, it would be much easier for bookkeeping and further analysis if these output can be redirected to disk files. The `output` and `outputExpr` parameters are designed for this purpose.

Parameter `output` can take the following values:

- `''` (an empty string): No output.
- `'>'`: Write to standard output.
- `'filename'` or `'>filename'`: Write the output to a file named filename. If multiple operators write to the same file, or if the same operator writes to the file file several times, only the last write operation will succeed.
- `'>>filename'`: Append the output to a file named filename. The file will be opened at the beginning of `evolve` function and closed at the end. An existing file will be cleared.
- `'>>>filename'`: This is similar to the `'>>'` form but the file will not be cleared at the beginning of the `evolve` function.

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

A question naturally arises: what if I would like to write to different output for different replicate, or at different generation? For example, the `savePopulation` operator in Example 2.26 write to file `sample.pop`. This works well if there is only one replicate but not so when the operator is applied to multiple populations. Only the last population will be saved successfully! In this case, parameter `outputExpr` should be used.

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

Example 2.29 demonstrates the use of these two parameters. In this example, a table is written to file `LD.txt` using `output='>>LD.txt'`. Similar operation to `output='R2.txt'` fails because only the last R^2 value is written to this file. The last operator writes output for each replicate to their respective output file such as `LD_0.txt`, using an expression that involves variable `rep`.

Example 2.28: Use the output and outputExpr parameters

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

```
0.20    0.20    0.20
0.17    0.12    0.17
0.11    0.11    0.15
0.09    0.06    0.13

>>> print open('R2.txt').read()    # Only the last write operation succeed.
0.23
>>> print open('LD_2.txt').read()  # Each replicate writes to a different file.
0.24    0.20    0.17    0.15    0.13
>>>
```

2.6.4 Hybrid operators

Despite the large number of built-in operators, it is obviously not possible to implement every genetics models available. For example, although simuPOP provides several penetrance models, a user may want to try a customized one. In this case, one can use a *hybrid operator*.

A *hybrid operator* is an operator that calls a user-defined function when its applied to a population. The number and meaning of input parameters and return values vary from operator to operator. For example, a hybrid mutator sends a to-be-mutated allele to a user-defined function and use its return value as a mutant allele. A hybrid selector uses the return value of a user defined function as individual fitness. Such an operator handles the routine part of the work (e.g. scan through a chromosome and determine which allele needs to be mutated), and leave the creative part to users. Such a mutator can be used to implement complicated genetic models such as an asymmetric stepwise mutation model for microsatellite markers.

For example, Example 2.29 defines a three-locus heterogeneity penetrance model [Risch, 1990] that yields positive penetrance only when at least two disease susceptibility alleles are available. The underlying mechanism of this operator is that for each individual, simuPOP will collect genotype at specified loci (parameter `loci`) and send them to function `myPenetrance` and evaluate. The return values are used as the penetrance value of the individual, which is then interpreted as the probability that this individual will become affected.

Example 2.29: Use a hybrid operator

```
>>> def myPenetrance(geno):
...     'A three-locus heterogeneity penetrance model'
...     if sum(geno) < 2:
...         return 0
...     else:
...         return sum(geno)*0.1
...
>>> simu = simulator(population(1000, loci=[20]*3), randomMating())
>>> simu.evolve(
...     preOps = [initByFreq([0.8, 0.2])],
...     ops = [
...         pyPenetrance(func=myPenetrance, loci=[10, 30, 50]),
...         stat(numOfAffected=True),
...         pyEval(r"'%d: %d\n' % (gen, numOfAffected)")
...     ],
...     gen = 5
... )
0: 83
1: 73
2: 78
3: 74
4: 73
(5,)
>>>
```

2.6.5 Python operators

If hybrid operators are still not flexible enough, you can always resort to a pure-Python operator `pyOperator`. This operator has full access to the evolving population (or parents and offspring when `stage=DuringMating`), and can therefore perform arbitrary operations.

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

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

where `param` is optional, depending on whether or not a parameter is passed to the `pyOperator()` constructor. Function `func` can perform arbitrary action to `pop` and must return `True` or `False`. The evolution of `pop` will be stopped if this function returns `False`.

Example 2.30 defines such a function. It accepts a cutoff value and two mutation rates as parameters. It then calculate the frequency of allele 1 at each locus and apply a two-allele model at high mutation rate if the frequency is lower than the cutoff and a low mutation rate otherwise. The `KamMutate` function is the function form of a mutator `kamMutator` (see Section 2.6.7 for details).

Example 2.30: A frequency dependent mutation operator

```
def dynaMutator(pop, param):
    '''This mutator mutates commom loci with low mutation rate and rare
    loci with high mutation rate, as an attempt to raise allele frequency
    of rare loci to an higher level.'''
    # unpack parameter
    (cutoff, mu1, mu2) = param;
    Stat(pop, alleleFreq=range(pop.totNumLoci()))
    for i in range(pop.totNumLoci()):
        # Get the frequency of allele 1 (disease allele)
        if pop.dvars().alleleFreq[i][1] < cutoff:
            KamMutate(pop, maxAllele=1, rate=mu1, loci=[i])
        else:
            KamMutate(pop, maxAllele=1, rate=mu2, loci=[i])
    return True
```

Example 2.41 demonstrates how to use this operator. It first initializes the population using two `initByFreq` operators that initialize loci with different allele frequencies. It applies a `pyOperator` with function `dynaMutator` and a tuple of parameters. Allele frequencies at all loci are printed at generation `0`, `10`, `20`, and `30`. Note that this `pyOperator` is applied at `stage=PreMating` (the default stage is post mating) so allele frequencies have to be recalculated to be used by post-mating operator `pyEval`.

Example 2.31: Use a pyOperator during evolution

```
>>> simu = simulator(population(size=10000, loci=[2, 3]),
...     randomMating())
>>> simu.evolve(
...     preOps = [
...         initByFreq([.99, .01], loci=[0, 2, 4]),
...         initByFreq([.8, .2], loci=[1, 3])],
...     ops = [
...         pyOperator(func=dynaMutator, param=(.2, 1e-2, 1e-5), stage=PreMating),
...         stat(alleleFreq=range(5), step=10),
...         pyEval(r"' '.join(['%.2f' % alleleFreq[x][1] for x in range(5)]) + '\n'",
...             step=10),
...     ],
...     gen = 31
... )
0.02 0.21 0.02 0.21 0.02
0.11 0.21 0.11 0.20 0.11
0.17 0.21 0.17 0.20 0.18
```

```
0.20 0.22 0.20 0.21 0.21
(31,)
>>>
```

An `pyOperator` can also be applied during-mating. They can be used to filter out unwanted offspring (by returnning `False` in a user-defined function), modify offspring, calculate statistics, or pass additional information from parents to offspring. Depending on parameter `param` and `offspringOnly`, such an operator accepts a function in the form of

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

Example 2.32 demonstrates the use of a during-mating Python operator. This operator rejects an offspring if it has allele 1 at the first locus of the first homologous chromosome, and results in an offspring population without such individuals.

Example 2.32: Use a during-mating pyOperator

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

`pyOperator` is the most powerful operator in simuPOP and has been widely used, for example, to calculate statistics and is not supported by the `stat()` operator, to examine population property during evolution, or prepare populations for a special mating scheme. However, because `pyOperator` works in the Python interpreter, it is expected that it runs slower than operators that are implemented at the C/C++ level. If performance becomes an issue, you can reimplement part or all the operator in C++. Section 2.8.8 describes how to do this.

2.6.6 Define your own operators

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

`pyOperator` is a Python class so you can derive your own operator from this operator. The tricky part is that the constructor of the derived operator needs to call the `__init__` function of `pyOperator` will proper functions. This technique has been used by simuPOP in a number of occasions. For example, the `varPlotter` operator defined in `simuRPy.py` is derived from `pyOperator`. This class encapsulates several different plot class that uses `rpy` to plot python expressions. One of the plotters is passed to the func parameter of `pyOperator::__init__` so that it can be called when this operator is applied.

Example 2.41 rewrites the `dynaMutator` defined in Example 2.30 into a derived operator. The parameters are now passed to the constructor of `dynaMutator` and are saved as member variables. A member function `mutate` is defined and is passed to the constructor of `pyOperator`. Other than making `dynaMutator` look like a real

simuPOP operator, this example does not show a lot of advantage over defining a function. However, when the operator gets complicated (as in the case for `varPlotter`), the object oriented implementation will prevail.

Example 2.33: Define a new Python operator

```
>>> class dynaMutator(pyOperator):
...     '''This mutator mutates commom loci with low mutation rate and rare
...     loci with high mutation rate, as an attempt to raise allele frequency
...     of rare loci to an higher level.'''
...     def __init__(self, cutoff, mu1, mu2, *args, **kwargs):
...         self.cutoff = cutoff
...         self.mu1 = mu1
...         self.mu2 = mu2
...         pyOperator.__init__(self, func=self.mutate, *args, **kwargs)
...     #
...     def mutate(self, pop):
...         Stat(pop, alleleFreq=range(pop.totNumLoci()))
...         for i in range(pop.totNumLoci()):
...             # Get the frequency of allele 1 (disease allele)
...             if pop.dvars().alleleFreq[i][1] < self.cutoff:
...                 KamMutate(pop, maxAllele=1, rate=self.mu1, loci=[i])
...             else:
...                 KamMutate(pop, maxAllele=1, rate=self.mu2, loci=[i])
...         return True
...
>>> simu = simulator(population(size=10000, loci=[2, 3]),
...     randomMating())
>>> simu.evolve(
...     preOps = [
...         initByFreq([.99, .01], loci=[0, 2, 4]),
...         initByFreq([.8, .2], loci=[1, 3])],
...     ops = [
...         dynaMutator(cutoff=.2, mu1=1e-2, mu2=1e-5, stage=PreMating),
...         stat(alleleFreq=range(5), step=10),
...         pyEval(r"' '.join(['%.2f' % alleleFreq[x][1] for x in range(5)]) + '\n'",
...             step=10),
...     ],
...     gen = 31
... )
0.02 0.20 0.02 0.21 0.02
0.11 0.20 0.11 0.21 0.11
0.18 0.20 0.19 0.21 0.18
0.21 0.21 0.22 0.20 0.21
(31,)
>>>
```

New during-mating operators can be defined similarly. They are usually used to define customized genotype transmitters. Section 2.8.4 will describe this feature in detail.

2.6.7 Function form of an operator

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

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

Similarly, you can apply the hybrid penetrance model defined in Example 2.29 to a population by

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

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

Example 2.34: The function form of operator `initByFreq`

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

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

```
InitByFreq(pop, [.3, .2, .5])
```

and

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

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

2.7 Mating Schemes

Mating schemes are responsible for populating an offspring generation from the parental generation. There are currently three types of mating schemes

- A **homogeneous mating scheme** is the most flexible and most frequently used mating scheme and is the center topic of this section. A homogeneous mating is responsible for choosing parent(s) from a (virtual) subpopulation and population all or part of the offspring generation.
- A **heterogeneous mating scheme** applies several homogeneous mating scheme to different (virtual) subpopulations. Because the division of virtual subpopulations can be arbitrary, this mating scheme can be used to simulate mating in heterogeneous populations such as populations with age structure.
- A **pedigree mating scheme** that follows a recorded evolutionary scenario. The selection of parents and the production of offspring are controlled by a pedigree. This mating scheme does not support virtual subpopulation.

This section describes some standard features of mating schemes and most pre-defined mating schemes. The next section will demonstrate how to build complex nonramdom mating schemes from scratch.

2.7.1 Control the size of the offspring generation

A mating scheme goes through each subpopulation and populates the subpopulations of an offspring generation sequentially. The number of offspring in each subpopulation is determined by the mating scheme, following the following rules:

- A simuPOP mating scheme, by default, produces an offspring generation that has the same subpopulation sizes as the parental generation. This does not gurantee a constant population size because some operators, such as a migrator, can change population or subpopulation sizes.
- If fixed subpopulation sizes are given to parameter `subPopSize`. A mating scheme will generation an offspring generation with specified sizes even if an operator has changed parental population sizes.
- A demographic function can be specified to parameter `subPopSize`. This function should take two parameters: the generation number and the current subpopulation sizes, and return an array of new subpopulation sizes (return `[newsize]` instead of `newsize` if you do not have any subpopulation structure).

The following examples demonstrate these cases. Example 2.35 uses a default `randomMating()` scheme that keeps parental subpopulation sizes. Because migration between two subpopulations are asymmetric, the size of the first subpopulation increases at each generation, although the overall population size keeps constant.

Example 2.35: Free change of subpopulation sizes

```
>>> simu = simulator(
...     population(size=[500, 1000], infoFields=['migrate_to']),
...     randomMating())
>>> simu.evolve(
...     preOps = [initSex()],
...     ops = [
...         migrator(rate=[[0.8, 0.2], [0.4, 0.6]]),
...         stat(popSize=True),
...         pyEval(r'"%s\n" % subPopSize')
...     ],
...     gen = 3
... )
[774, 726]
[912, 588]
[956, 544]
(3,)
>>>
```

Example 2.36 uses the same migrator to move individuals between two subpopulations. Because a constant subpopulation size is specified, the offspring generation always has 500 and 1000 individuals in its two subpopulations. Note that operators `stat` and `pyEval` are applied both before and after mating. It is clear that subpopulation sizes changes before mating as a result of migration, although the pre-mating population sizes vary because of uncertainties of migration.

Example 2.36: Force constant subpopulation sizes

```
>>> simu = simulator(
...     population(size=[500, 1000], infoFields=['migrate_to']),
...     randomMating(subPopSize=[500, 1000]))
>>> simu.evolve(
...     preOps = [initSex()],
...     ops = [
...         migrator(rate=[[0.8, 0.2], [0.4, 0.6]]),
...         stat(popSize=True, stage=PrePostMating),
...         pyEval(r'"%s\n" % subPopSize', stage=PrePostMating)
...     ],
...     gen = 3
... )
[782, 718]
[500, 1000]
[783, 717]
[500, 1000]
[817, 683]
[500, 1000]
(3,)
>>>
```

Example 2.40 uses a demographic function to control the subpopulation size of the offspring generation. This example implements a linear population expansion model but arbitrarily complex demographic model can be implemented similarly.

Example 2.37: Use a demographic function to control population size

```
>>> def demo(gen, oldSize=[]):
...     return [500 + gen*10, 1000 + gen*10]
```

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

All these examples have fixed number of subpopulations. Section 3.1 will introduce how to split and merge subpopulations dynamically.

2.7.2 Determine the number of offspring during mating

simuPOP by default produces only one offspring per mating event. Because more parents are involved in the production of offspring, this setting leads to larger effective population sizes than mating schemes that produce more offspring at each mating event. However, various situations require a larger family size or even varying family sizes. In these cases, parameter `numOffspring` can be used to control the number of offspring that are produced at each mating event. This parameter takes the following types of inputs

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

More specifically,

- `numOffspring=(GeometricDistribution, p)`: The number of offspring for each mating event follows distribution:
$$\Pr(k) = p(1-p)^{k-1} \quad \text{for } k \geq 1$$
This distribution has mean $1/p$ and variance $(1-p)/p^2$.
- `numOffspring=(PoissonDistribution, p)`: The number of offspring for each mating event follows a shifted Poisson distribution:
$$\Pr(k) = p^{k-1}\frac{e^{-p}}{(k-1)!} \quad \text{for} k \geq 1$$
Because the mean of this shifted Poisson distribution is $p+1$, you need to specify, for example, 2, if you want a mean family size of 3. The variance of this distribution is p.

- `numOffspring=(BinomialDistribution, p, n)`: The number of offspring for each mating event follows a shifted Binomial distribution:

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

  The mean of this distribution is $(n-1)p+1$ with variance $(n-1)\,p\,(1-p)$.

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

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

  The mean of this distribution is $(a+b)/2$.

Example 2.38 demonstrates how to use parameter `numOffspring`. In this example, a function `checkNumOffspring` is defined. It takes a mating scheme as its input parameter and use it to evolve a population with 30 individuals. After evolving a population for one generation, parental indexes are used to identify siblings, and then the number of offspring per mating event.

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

```
>>> def checkNumOffspring(ms):
...     '''Check the number of offspring for each family using
...        information field father_idx
...     '''
...     simu = simulator(
...         population(size=[30], infoFields=['father_idx', 'mother_idx']),
...         matingScheme=ms)
...     simu.evolve(
...         preOps = [initSex()],
...         ops=[parentsTagger()],
...         gen=1)
...     # get the parents of each offspring
...     parents = [(x, y) for x, y in zip(simu.population(0).indInfo('mother_idx'),
...         simu.population(0).indInfo('father_idx'))]
...     # Individuals with identical parents are considered as siblings.
...     famSize = []
...     lastParent = (-1, -1)
...     for parent in parents:
...         if parent == lastParent:
...             famSize[-1] += 1
...         else:
...             lastParent = parent
...             famSize.append(1)
...     return famSize
...
>>> # Case 1: produce the given number of offspring
>>> checkNumOffspring(randomMating(numOffspring=2))
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
>>> # Case 2: Use a Python function
>>> import random
>>> def func(gen):
...     return random.randint(5, 8)
...
>>> checkNumOffspring(randomMating(numOffspring=func))
[8, 8, 5, 5, 4]
>>> # Case 3: A geometric distribution
>>> checkNumOffspring(randomMating(numOffspring=(GeometricDistribution, 0.3)))
```

```
[1, 6, 1, 4, 2, 6, 1, 9]
>>> # Case 4: A Possition distribution
>>> checkNumOffspring(randomMating(numOffspring=(PoissonDistribution, 3)))
[7, 4, 5, 5, 4, 3, 2]
>>> # Case 5: A binomial distribution
>>> checkNumOffspring(randomMating(numOffspring=(BinomialDistribution, 0.1, 10)))
[4, 1, 1, 1, 2, 2, 2, 2, 1, 2, 3, 1, 3, 2, 1, 1, 1]
>>> # Case 6: A uniform distribution
>>> checkNumOffspring(randomMating(numOffspring=(UniformDistribution, 2, 6)))
[6, 6, 2, 6, 4, 2, 4]
>>>
```

2.7.3 Determine offspring sex

Because sex can influence how genotypes are transmitted (e.g. sex chromosomes, haplodiploid population), simuPOP determines offspring sex before it passes an offspring to a *genotype transmitter* (during-mating operator) to transmit genotype from parents to offspring. The default `sexMode` in almost all mating schemes is `RandomSex`, in which case simuPOP assign `Male` or `Female` to offspring with equal probability.

Other sex determination methods are also available:

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

`NumOfMale` and `NumOfFemale` are useful in theoretical studies where the sex ratio of a population needs to be controlled strictly, or in special mating schemes, usually for animal populations, where only a certain number of male or female individuals are allowed in a family. It worth noting that a genotype transmitter can override specified offspring sex. This is the case for `cloneGenoTransmitter` where an offspring inherits both genotype and sex from his/her parent.

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

Example 2.39: Determine the sex of offspring

```
>>> def checkSexMode(ms):
...     '''Check the assignment of sex to offspring'''
...     simu = simulator(
...         population(size=[40]),
...         matingScheme=ms)
...     simu.evolve(preOps = [initSex()], ops=[], gen=1)
...     # return individual sex as a string
...     return ''.join([ind.sexChar() for ind in simu.population(0).individuals()])
...
>>> # Case 1: NoSex (all male, randomMating will not continue)
>>> checkSexMode(randomMating(sexMode=NoSex))
'MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM'
>>> # Case 2: RandomSex (Male/Female with probability 0.5)
>>> checkSexMode(randomMating(sexMode=RandomSex))
'MMMFMFMFFFFMFMMMFMFMMFFFFMFMFMFMMMMMFFFM'
```

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

2.7.4 Monogamous mating

Monogamous mating (monogamy) in simuPOP refers to mating schemes in which each parent mates only once. In an asexual setting, this implies parents are chosen without replacement. In sexual mating schemes, this means that parents are chosen without replacement, they have only one spouse during their life time so that all siblings have the same parents (no half-sibling).

simuPOP provides a diploid sexual momogamous mating scheme `monogamousMating`. However, without careful planning, this mating scheme can easily stop working due to the lack of parents. For example, if a population has 40 males and 55 females, only 40 successful mating events can happen and result in 40 offspring in the offspring generation. `monogamousMating` will exit if the offspsring genertion is larger than 40.

Example 2.40 demonstrates one senario of using a monogamous mating scheme where sex of parents and offspring are strictly specified so that parents will not be exhausted. The sex initializer `initSex` assign exactly 10 males and 10 females to the initial population. Because of the use of `numOffspring=2, sexMode=(NumOfMale, 1)`, each mating event will produce two offspring with one male and one female. Unlike a random mating scheme that only about 80% of parents are involved in the production of an offspring population with the same size, this mating scheme makes use of all parents.

Example 2.40: Sexual monogamous mating

```
>>> simu = simulator(population(20, infoFields=['father_idx', 'mother_idx']),
...     monogamousMating(numOffspring=2, sexMode=(NumOfMale, 1)))
>>> simu.evolve(
...     preOps = [initSex(sex=(Male, Female))],
...     ops = [parentsTagger()],
...     gen = 5
... )
(5,)
>>> pop = simu.extract(0)
>>> [ind.sex() for ind in pop.individuals()]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
>>> [ind.intInfo('father_idx') for ind in pop.individuals()]
[2, 2, 8, 8, 18, 18, 4, 4, 16, 16, 6, 6, 14, 14, 10, 10, 0, 0, 12, 12]
>>> [ind.intInfo('mother_idx') for ind in pop.individuals()]
[19, 19, 17, 17, 9, 9, 11, 11, 5, 5, 3, 3, 15, 15, 7, 7, 1, 1, 13, 13]
>>> # count the number of distinct parents
>>> len(set(pop.indInfo('father_idx')))
10
>>> len(set(pop.indInfo('mother_idx')))
10
>>>
```

2.7.5 Polygamous mating

2.7.6 Asexual random mating

2.7.7 Mating with alpha individuals

2.7.8 Mating in haplodiploid populations

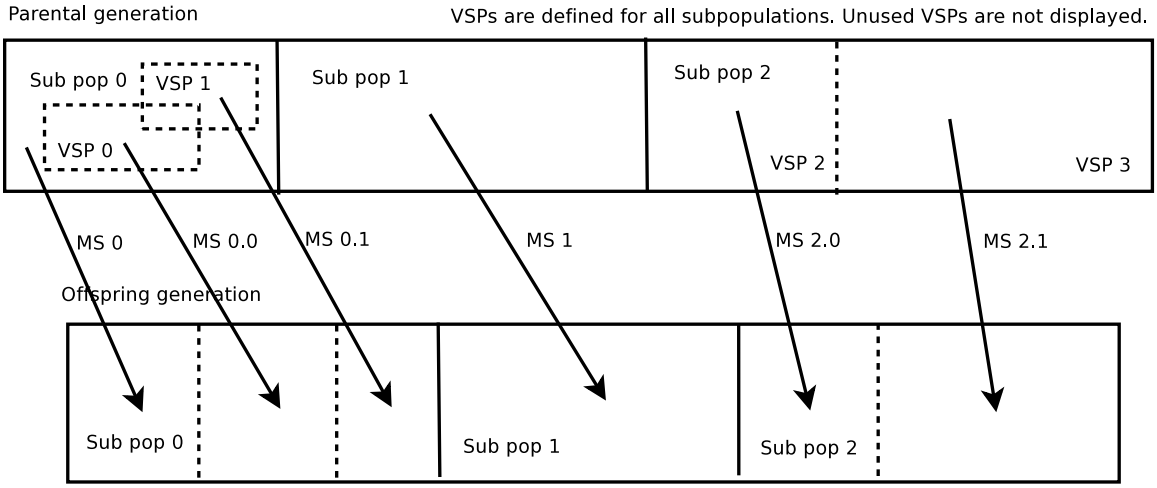
2.7.9 Selfing

2.7.10 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.

2.8 Non-random and customized mating schemes

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

2.8.1 The structure of a homogeneous mating scheme

For simplicity, I will refer to a homogeneous mating scheme as a mating scheme, and use parental and offspring generation directly, although a homogeneous mating scheme can be applied to virtual subpopulations and populate only a fraction of the offspring generation.

A mating scheme populate an offspring generation as follows:

1. Create an empty offspring population (generation) with appropriate size. Parental and offspring generation can differ in size but they must have the same number of subpopulations.
2. For each subpopulation, repeatedly choose a parent or a pair of parents from the parental generation. This is done by a simuPOP object called a **parent chooser**.
3. One or more offspring are produced from the chosen parent(s) and are placed in the offspring population. This is done by a simuPOP **offspring generator**.
4. A offspring generator uses one or more during-mating operators to transmit parental genotype to offspring. These operators are call **genotype transmitters**.
5. After the offspring generation is populated, it will replace the parental generation and becomes the present generation of a population.

A simuPOP mating scheme is responsible for all these steps. Each mating scheme uses a particular set of parent chooser, offspring generator, and genotype transmitter. For example, a `selfingMating` uses a `randomParentChooser` to choose a parent randomly from a population, possibly according to individual fitness, it uses a standard `offspringGenerator` that is capable of producing one or more offspring. This offspring generator uses a `selfingOffspringGenerator` to transmit genotype.

simuPOP provides a number of parent choosers, offspring generators and genotype transmitters. It also provides a number of pre-defined mating schemes that make use of these objects. If none of the pre-defined mating schemes fits your need, you can define a new mating scheme by constructing one from stocked or user defined parent choosers, offspring generator, and genotype transmitters. Because both parent choosers and genotype transmitters can be constructed at the Python level, very complicated nonrandom mating schemes can be defined.

Before we dive into the details of constructing a mating scheme from scratch, let us have a look at the definition of the most commonly used mating scheme `randomMating` (Example 2.41). The following sections basically explain each parameter of this mating scheme in detail and show you which part can be customized.

Example 2.41: Define a random mating scheme

```
def randomMating(numOffspring = 1.,
        sexMode = MATE_RandomSex, ops = [], subPopSize = [],
        subPop = (), weight = 0):
    'A basic sexually random mating scheme.'
    return pyMating(
        chooser = randomParentsChooser(replacement=True),
        generator = offspringGenerator([mendelianGenoTransmitter()], 2,
            ops, numOffspring, sexParam, sexMode),
        subPopSize = subPopSize,
        subPop = subPop,
        weight = weight)
```

2.8.2 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 2.42: 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(),
...
(271, 560) (619, 771) (417, 594) (387, 987) (545, 384)
>>> rc2 = randomChooser(pop, 1)
>>> for i in range(5):
...     print rc2.next(),
...
(9, 148) (172, 178) (18, 154) (46, 171) (68, 178)
>>>
```

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 2.44, 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 chro-

mosomes to fill offspring genotype. In the diploid case, it will behave the same for `cloneOffspringGenerator` and `selfingOffspringGenerator`.

2.8.3 Genotype transmitter

Some during-mating operators transmit genotype from parents to offspring, and some do not. Unless you construct a mating scheme from scratch, each stocked mating scheme has a default during-mating operator that is responsible for transmitting genotype from parents to offspring. If another operator with the same flag is passed, the default one will not be used. The most common example is a `recombinator`, which will replace the default `mendelianGenoTransmitter` for a `randomMating` scheme if it is used.

2.8.4 Customized genotype transmitter

Although some operators can be applied before, after and during mating, most during-mating operators can only be applied by a mating scheme. During-mating operators have the following properties

used 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.

`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, offspringOnly=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`.

2.8.5 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 2.43 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 2.43: 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 2.44: 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(),
...
(271, 560) (619, 771) (417, 594) (387, 987) (545, 384)
>>> rc2 = randomChooser(pop, 1)
>>> for i in range(5):
...     print rc2.next(),
...
(9, 148) (172, 178) (18, 154) (46, 171) (68, 178)
>>>
```

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 2.44, 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 2.45 demonstrates how to use this user-defined parent chooser.

Example 2.45: 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 2.46 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 2.46: A heterogeneous mating scheme

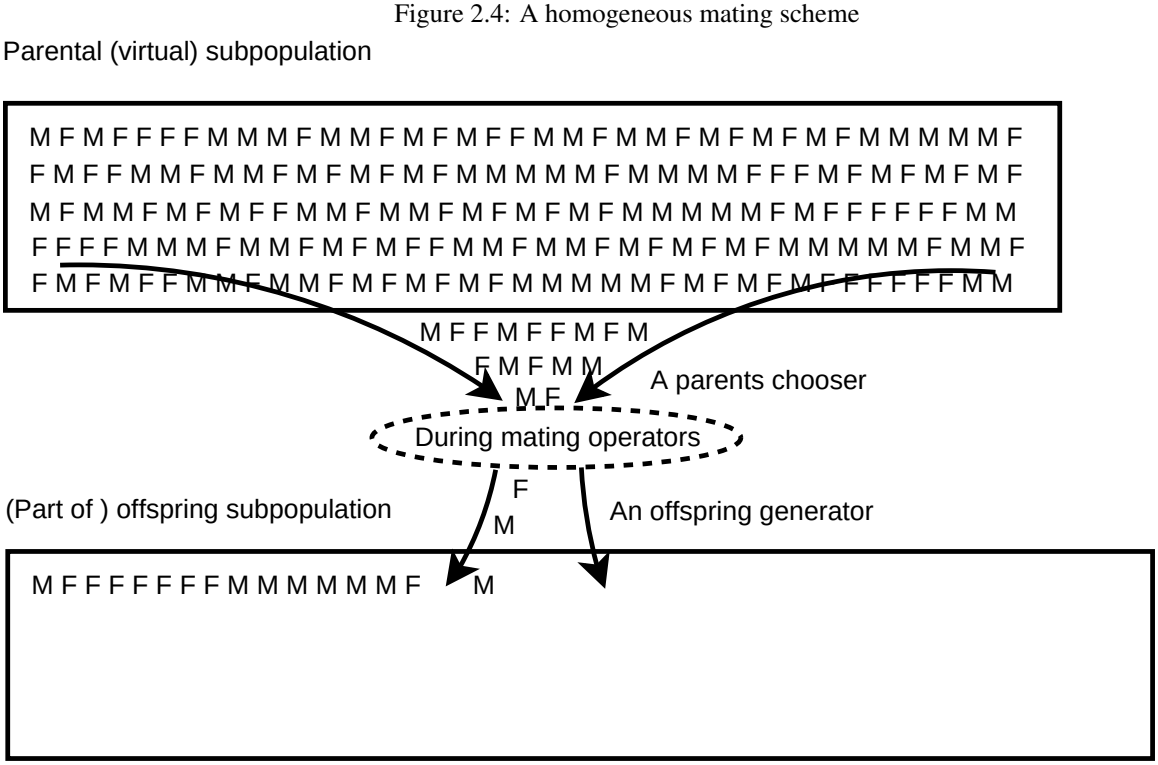
```
>>> TurnOnDebug(DBG_MATING)
>>> pop = population(100, loci=[2])
>>> pop.setVirtualSplitter(proportionSplitter([0.2, 0.8]))
>>> simu = simulator(pop, heteroMating(
...     [selfMating(numOffspring=5, subPop=0, virtualSubPop=0),
...     randomMating(numOffspring=20, subPop=0, virtualSubPop=1)]))
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
TypeError: selfMating() got an unexpected keyword argument 'virtualSubPop'
>>>
>>> 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().
>>>
```

2.8.6 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 2.4: 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.

2.8.7 Python parents chooser

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

2.8.8 Using C++ to implement a parents chooser

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

2.8.9 The pedgiree mating scheme

This section is under development.

2.9 Simulator

Simulators combine three important components of simuPOP: population, mating scheme and operators together. A simulator is 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.

2.10 Pedigrees

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 2.47: 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 2.48: 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.

Chapter 3

simuPOP Operators

simuPOP is large, consisting of more than 80 operators and various functions that covers all important aspects of genetic studies. These includes mutation (*k*-allele, stepwise, generalized stepwise), migration (arbitrary, can create new subpopulation), recombination (uniform or nonuniform), gene conversion (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 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 2.36, 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 3.1: 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 2.36, 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 3.2: 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]
[101, 430, 469]
[84, 462, 454]
[64, 484, 452]
[62, 486, 452]
[65, 485, 450]
[75, 496, 429]
(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 3.3, 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 3.3: 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 [238, 762] to: [400, 500]
From [316, 584] to: [400, 500]
From [88, 344, 468]     to: [300, 400, 600]
From [174, 468, 658]    to: [300, 400, 600]
From [190, 468, 642]    to: [300, 400, 600]
From [179, 463, 658]    to: [300, 400, 600]
From [180, 468, 652]    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 3.4 demonstrate an exponentially increase population with free migration between subpopulations.

Example 3.4: 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 [1930, 6070]       to: [3860, 12140]
From [3070, 12930]      to: [6140, 25860]
From [3343, 6829, 21828]        to: [6686, 13658, 43656]
From [7042, 17982, 38976]       to: [14084, 35964, 77952]
From [13289, 42955, 71756]      to: [26578, 85910, 143512]
From [24843, 96994, 134163]     to: [49686, 193988, 268326]
From [46906, 211561, 253533]    to: [93812, 423122, 507066]
(10,)
>>>
```

3.1.1 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`).

3.1.2 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.

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

Example 3.5: List the order at which operators are applied

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

```
Start evolution
  Replicate 0
    Pre-mating operators
      - <simuPOP::pyEval > at generation(s) -10 -1
    Start mating
    Apply post-mating operators
      - <simuPOP::statistics> begin at 80 at interval 10
      - <simuPOP::pyEval > begin at 80 at interval 10
      - <simuPOP::pyEval > at generation(s) -10 -1
Apply post-evolution operators:
  Replicate 0
      - <simuPOP::save population> at all generations
(0,)
>>>
```

3.1.3 Terminate a simulator

3.1.4 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 3.6: Save and load a simulator

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

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

3.3 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.

3.4 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

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

  and if `circular=True`, returns

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

3.5 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 3.7), 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 3.7: 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.

3.6 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.

3.7 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.

3.8 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 3.8). 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 3.8: 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 3.9) 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 3.9: 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 3.10) 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 3.10: 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 3.11) 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 3.11: 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 3.12) 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 3.12: 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()
```

3.9 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 3.13: 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.+-]+|na)\s+
    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 3.13. 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

Utility Modules

4.1 simuOpt

4.2 simuUtil

4.2.1 ascertainment operators

4.3 simuRPy

Chapter 5

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.

5.1 Simulation scenario

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

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

5.2 Demographic model

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

5.3 Mutation model

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

5.4 Selection on a common and a rare disease

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

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

These parameters, translated to python, are shown in 5.1

Example 5.1: Set parameters

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

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

5.5 Create a simulator

Several parameters are needed to create a population:

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

You can then create a population with:

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

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

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

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

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

Example 5.2: Create a simulator

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

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

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

simulate(ins_exp)
```

5.6 Initialization

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

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

5.7 Mutation and selection

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

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

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

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

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

  and

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

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

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

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

Example 5.3: Run the simulator

```
def simulate(incScenario):
    simu = simulator(                       # create a simulator
        population(subPop=incScenario(0), loci=[1,1],
            infoFields=['fitness']),        # inital population
        randomMating(newSubPopSizeFunc=incScenario)
    )
    simu.evolve(                            # start evolution
        preOps=[                            # operators that will be applied before evolution
            # 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)
```

5.8 Output statistics

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

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

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

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

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

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

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

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

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

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

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

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

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

Then, you can use this function in a python operator

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

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

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

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

Example 5.4: The whole program

```
#!/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 evolution
            # 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)
```

5.9 Option handling

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

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

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

  to view help information.

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

```
> simuLDDecay.py -s 500 -e 10 --recRate 0.1 --numRep 5 --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 5.5 shows the beginning and the execution part of the complete `reich.py` script, which can be found under the doc directory. For a complete reference of the Options structure, please refer to the reference manual.

Example 5.5: Option handling

```
options = [
    {'arg': 'h',
     'longarg': 'help',
     'default': False,
     'description': 'Print this usage message.',
     'allowedTypes': [types.NoneType, type(True)],
     'jump': -1                    # if -h is specified, ignore any other parameters.
    },
    {'longarg': 'initSize=',
     'default': 10000,
     'label': 'Initial population size',
     'allowedTypes': [types.IntType, types.LongType],
     'description': '''Initial population size. This size will be maintained
               till the end of burnin stage''',
     'validate': simuOpt.valueGT(0)
    },
    {'longarg': 'finalSize=',
     'default': 1000000,
     'label': 'Final population size',
     'allowedTypes': [types.IntType, types.LongType],
     'description': 'Ending population size (after expansion.',
     'validate': simuOpt.valueGT(0)
    },
    {'longarg': 'burnin=',
     'default': 500,
     'label': 'Length of burn-in stage',
```

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

def getOptions(details=__doc__):
    # get all parameters, __doc__ is used for help info
    allParam = simuOpt.getParam(options,
        'This program simulates the evolution of a common and a rare direse\n' +
        'and observe the evolution of allelic spectra\n', details)
    #
    # when user click cancel ...
    if len(allParam) == 0:
        sys.exit(1)
    # -h or --help
    if allParam[0]:
        print simuOpt.usage(options, __doc__)
        sys.exit(0)
    # automatically save configurations
    name = allParam[-1]
    if not os.path.isdir(name):
        os.makedirs(name)
    simuOpt.saveConfig(options, os.path.join(name, name+'.cfg'), allParam)
    # return the rest of the parameters
    return allParam[1:-1]

#
# IGNORED
#

if __name__ == '__main__':
    # get parameters
    (initSize, finalSize, burnin, endGen, growth) = getOptions()
    #
    from simuPOP import *
    #
    if initSize > finalSize:
        print 'Initial size should be greater than final size'
```

```
        sys.exit(1)
    if burnin > endGen:
        print 'Burnin gen should be less than ending gen'
        sys.exit(1)
    if growth == 'linear':
        simulate(lin_exp)
    else:
        simulate(ins_exp)
```

Chapter 6

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.

6.1 Examples and teaching scripts

6.1.1 `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.

6.1.2 `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.

6.2 Utility scripts

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

6.2.1 `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

6.2.2 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 6.1 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 6.1: 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)
```

6.2.3 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.

6.3 General simulation scripts

6.3.1 `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].

6.3.2 `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].

6.3.3 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.

6.4 Simulations of the evolution of complex human diseases

6.4.1 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.

6.4.2 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.

6.4.3 `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. 66, 86

C I Amos. Robust variance-components approach for assessing genetic linkage in pedigrees. *Am J Hum Genet*, 54: 535–543, 1994. 86

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. 85

Eric S Lander. The new genomics: Global views of biology. *Science*, 274(5287):536–539, 1996. 84

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. 84

Bo Peng and Marek Kimmel. Simulations provide support for the common disease common variant hypothesis. *Genetics*, 175:1–14, 2007. 69, 84

Jonathan K Pritchard. Are rare variants responsible for susceptibility to complex diseases. *Am J Hum Genet*, 69: 124–137, 2001. 84

David E Reich and Eric S Lander. On the allelic spectrum of human disease. *Trends Genet*, 17(9):502–510, 2001. 69, 73, 84

Neil Risch. Linkage strategies for genetically complex traits. i. multilocus models. *Am J Hum Genet*, 46:222–228, 1990. 29

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. 86

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. 86

Index