simuPOP User's Guide

Release 0.7.0 (Rev: 324)

Bo Peng

December 2004

Last modified
27th August 2006

Department of Statistics, Rice University
Email: bpeng@rice.edu
URL: http://simupop.sourceforge.net
Mailing List: simupop-list@lists.sourceforge.net

Acknowledgements:

Dr. Marek Kimmel
Dr. François Balloux
Dr. William Amos
SWIG user community
Python user community
Keck Center for Computational and Structural Biology

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 can be used at two levels. The core of simuPOP is a scripting language (Python) that provides a large number of building blocks (populations, mating schemes, various genetic forces in the form of functions, operators, simulators and gene mapping methods) to construct a simulation. This provides a R/S-Plus or Matlab-like environment where users can interactively create, manipulate and evolve populations; monitor and visualize population statistics and apply gene mapping methods. The full power of simuPOP and Python (even R) can be utilized to simulate arbitrarily complex evolutionary scenarios.

simuPOP also comes with an increasing number of pre-defined simulation scenarios. If one of them happens to fit your need, all you need to do is running the script file with appropriate parameters. No knowledge of Python or simuPOP is required. To make simuPOP readily usable for time-limited users, users of simuPOP are strongly encouraged to submit their simulations to this collection.

This user's guide covers the basic usage of simuPOP, including installation, basic usage, brief introduction to built-in scripts, and how to write simuPOP scripts. Detailed information about simuPOP components, functions and operators is available in the *simuPOP Reference Manual.* All resources, including a pdf version of this guide and a discussion forum 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

CONTENTS

CHAPTER

ONE

Introduction

1.1 What is simuPOP?

`simuPOP` is a forward-time population genetics simulation environment. Unlike coalescent-based simulation programs, simuPOP evolves population(s) forward in time, subject to arbitrary number of genetic and environmental forces (mutation, recombination, migration, population size change etc.). simuPOP allows users to control every aspects of the evolutionary process and observe the details at each generation. For example, users can start with a population of identical individuals, manually introduce a mutant and observe the spread of this mutant in the population from generation to generation. Population substructure, recombination, migration, selection etc can be added to the simulation as needed.

simuPOP consists of a number of Python objects and functions, including populations that store and provide access to individual genotypes; mating schemes that determine how populations evolve to the next generation; operators that manipulate populations and calculate population statistics; simulators that coordinate the evolution process and functions that perform tasks ranging from saving/loading populations to doing gene mapping. It is user's responsibility to write a Python script to glue these pieces together and form a simulation. Since these modules are mostly independent to each other, it is easy to add additional operators to an existing simulation. There is no limit on the number of operators, and thus no limit on the complexity of a simulation.

simuPOP does not aim at any specific result or outcome. It is more like a workshop, where users use various components and tools to assemble a simulation and study its properties. Just like any such programming environments such as R/Splus and Matlab, users will have to learn how to use the environment (various Python IDE) and how to program in this language (Python and the simuPOP module). A graphic user interface of simuPOP is planned but its usefulness is in doubt (just like the R/GUI) and will not be available any time soon.

On the other hand, simuPOP also has an increasing number of built-in scripts. These script are written in simuPOP/Python language and can be used without knowing their underlying machanisim. It is strongly recommended that users of simuPOP submit their own scripts to his collection and make simuPOP more and more accessible to time-limited users.

As a summary, simuPOP is suitable for the following applications:

- Teaching tool for population genetic courses. Compared to other existing programs, the biggest advantage of simuPOP is its flexibility. There is no limit on the complexity of the simulation and students can change the script and try new things (such as viewing another statistics or adding another genetic force) at will.
- Observe the dynamics of population evolution. This is where the power of simuPOP lies and is where coalescent-based simulations frown. Coalescent, by its nature, focus only on samples, and ignore genealogy information that are irrelevant to the final sample. It is therefore impractical to trace the population properties of ancestral populations. Forward-based simulation does not have this problem, at a cost of performance.
- Generating samples that can be analyzed by other programs. This area is dominated by coalescent-based methods, but the facts that coalescent-based methods can not simulate complex (non-additive) selection or penetrance

models and supports, at least till now, only one disease susceptibility locus, make it unsutable to simulate the evolution of complex human diseases. A simuPOP script simuComplexDisease.py provides a powerful alternative.

1.2 Features

Currently, simuPOP provides the following features:

- Population with one-level subpopulation structure. (no explicit family structure) There is no limit on ploidy, number of chromosomes, number of loci and population size. (depends on available RAM). Sex chromosomes can be modeled.
- Allele can be short (<255 allelic states), long (at least 2^{32} allelic states) or binary (0 or 1). Binary alleles are stored as bits so a large number of SNP markers can be simulated.
- A population can hold arbitrary number of ancestral generations (default to none) for easy pedigree analysis.
- Population/subpopulation sizes can be changed during mating. Subpopulations can be created/changed as a result of migration.
- Several replicates of populations can be evolved simultaneously.
- Mating schemes include random mating, binomial selection etc. Number of offsprings per mating can be constant, or follow a random distribution.
- Populations can be saved and loaded in text, binary, XML, Fstat, GC formats. Methods to deal with other formats are provided.
- Simulation can be paused, saved and resumed easily.
- Easy developing/debugging using Python interactive shell, or run in batch as python scripts.
- A wide variety of operators are provided. They can act on the populations at selected generations, at different stages of a life-cycle, on different replicate or replicate group.
- Built-in operators for arbitrary migration model.
- Operators for k-allele, stepwise and generalized stepwise mutation models. Hybrid operators can be used for more complicated mutation models.
- Support uniform or non-uniform (differ-by-loci) recombinations. Male/female individuals can have different recombination rates/intensities.
- Support many single-locus selection model and multiplicative/additive multi-loci selection models. Hybrid operator is provided for arbitrary selection model.
- Built-in support for allele, genotype, heterozygote, haplotype number/frequency calculation. As well as some more complicated statistics like F_{st}. Other statistics can be calculated from these basic statistics.
- Has support for plotting through Python/SciPY, Python/MatPlotLib or RPy (use R through Python). R/Rpy is recommended.
- Operators to calculate quantitative trait, penetrance and draw samples from current population.
- Built-in ascertainment methods including case/control, affected sibpair, random sample.
- Maybe most importantly: *a complete and detailed reference manual!*

1.3 Availability

Binary libraries of `simuPOP` are provided for linux, windows, solaris and mac systems. Source code and development documentations are also available for easy porting to other platforms. Both source code and binaries can be distributed free-of-charge under GPL license. All resources, including a pdf version of this manual and a discussion forum can be found at the simuPOP homepage.

1.4 Naming Conventions

`simuPOP` follows the following naming conventions.

- Classes (objects), member functions and parameter names start with small character and use capital character for the first character of each word afterwards. For example

  ```
  population, population::subPopSize(), individual::setInfo()
  ```

- Standalone functions start with capital character. This is how you can differ an operator from its function version. For example, `initByFreq` is an operator and `InitByFreq(pop, vars)` is its function version (equivalent to `initByFreq(vars).apply(pop)`.

- Constants start with Capital characters. For example

  ```
  MigrByProportion, StatNumOfFemale
  ```

- The following words in function names are abbreviated:

  ```
  dist (distance),  info (information), migr (migration), subPop (subpopulation),
  (rep) replicate, gen (generation), grp (group(s)), ops (operators),
  expr (expression), stmts (statements)
  ```

1.5 How to read this manual

There are a lot of functions/operators in simuPOP and there is no reason you should memorize all of them. (I admit that I can not.) If you are a first time simuPOP user, my suggestion is that you read through this manual quickly only to get the big picture of how simuPOP works and what simuPOP can do. Then, if you decide to write some simulations, you should

- Read some examples under `scripts` directory. From easy to difficult, you can read `simuLDDecay.py`, `simuCDCV.py` and `simuComplexDisease.py`.

- Copy one of the scripts as a template and modify it. For whatever function/operator you need, read the relevant sections in detail.

CHAPTER

TWO

Installing simuPOP

2.1 Installing simuPOP

Compiled libraries for Linux (RHEL4 and Mandriva), windows XP, Solaris and MacOSX are provided. In most cases, you will only need to download simuPOP and follow the usual installation process of your platform. For example, if you use a windows system and have Python 2.3.3 installed, you should download `simupop-x.x.x-py23-win32.exe`. Double clike the `.exe` file to install.

Things can get complicated when you have an earlier/later versions of OS or Python and have to compile simuPOP from source. The `installation` section of simuPOP homepage will have detailed instruction on it. (A single command `python setup.py instal` will usually suffice.)

Python has a large number of modules. For simple tasks like dataset generation, simuPOP modules alone are enough. However, it is highly recommended that you install

- R and python module rpy: although other ploting modules/methods can be used, simuPOP mainly uses R for this purpose. The advantage of this method is that R is not only an excellent plotting tool, but also a widely used statistical analysis package. It also has some genetic packages that can be used to analyze simuPOP generated datasets.
- wxPython: By default, simuPOP uses Tkinter to get parameters (the parameter dialog). It will use wxPython automatically if wxPython is available. This will enable a bunch of other GUI improvements including a nicer version of `listVars()` function.

2.2 Starting simuPOP

After installation, you will have the following files and directories (use windows as an example)

- Many `simuXXX.py` files under `c:\python23\Lib\site-packages`. These are simuPOP modules.
- `c:\python23\share\simuPOP\doc`: docmentations in pdf format.
- `c:\python23\share\simuPOP\test:` all unit test cases. You can run run_tests.py to test if your simuPOP installation is correct.
- `c:\python23\share\simuPOP\scripts:` This directory has all the built-in scripts.

You should be able to load simuPOP library by running command `import simuPOP` (example **??**) from python interactive shell. From the initial output, you can see the version (and revision number) of simuPOP, type of module, random number generator, etc.

In case that you do not have administrative previledge, you may not be able to install simuPOP to the system python directory. In this case, you can install simuPOP locally and load simuPOP as shown in example **??**.

Example 1 Import simuPOP module

```
>>> from simuPOP import *
simuPOP : Copyright (c) 2004-2006 Bo Peng
Version snapshot (Revision 400, Aug 24 2006) for Python 2.3.4
[GCC 3.4.3 20041212 (Red Hat 3.4.3-9.EL4)]
Random Number Generator is set to mt19937 with random seed 0x3e85d1a873fa9600
This is the short allele version with 256 maximum allelic states.
You are running in standard mode with strict boundary check etc.
For more information, please visit http://simupop.sourceforge.net,
or email simupop-list@lists.sourceforge.net (subscription required).
>>>
```

Example 2 Import locally installed simuPOP module

```
>>> import sys
>>> sys.path.append('/path/to/simuPOP')
>>> from simuPOP import *
>>>
```

2.3 simuPOP libraries

simuPOP is composed of six libraries: stdandard short, long and binary alleles, plus their optimized counterparts. The short libraries use 1 byte 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 like the infinite allele model. In those cases, you should use the long allele version of the modules. Long allele libraries use at least 4 bytes for each allele and can have 2^{32} 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. Due to the additional cost of accessing alleles as bits, binary modules will be around 10% slower than other libraries.

Standard libraries have detailed debug and run-time validation mechanisum to make sure the simulations run correctly. Whenever something unusual is detected, simuPOP would terminate with detailed error messages. The cost of such run-time cheching varies from simulation to simulation but can be high under some extreme circumstances. Because of this, optimized versions for all libraries 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 if needed and you are confidence that your simulation is running as expected.

Example 3 set options through simuOpt

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

You can control the choice of modules in the following ways:

- Set environment variable `SIMUALLELETYPE` to be 'short', 'long' or 'binary', or `SIMUOPTIMIZED` to use the optimized modules. The default module is the standard short module.
- Before you load simuPOP, set options using `simuOpt.setOptions(optimized, alleleType, quiet, debug)`. `alleleType` can be `short, long` or `binary. quiet` means suppress initial output, and `debug` should be a list of debug options specified by listDebugCode().
- If you are running a simuPOP script that conforms to simuPOP convension, you should be able to use optimized library using command line option `--optimized`.

CHAPTER

THREE

SimuPOP components

The core of simuPOP is a scripting language based on the Python programming language/environment. Like any other python module, you can start a python session, import simuPOP module, create and evolve populations interactively. Or, you can create a python script and run it as a batch file.

In this chapter, I will briefly explain each component and demonstrate them with an simple example. Detailed info about each components is given in the simuPOP reference manual.

3.1 Important simuPOP concepts

simuPOP consists of the following components. It is important that you know what they are and how they are related. In this chapter, I will briefly explain each component and explain each of them in detail.

individual individuals are building blocks of populations. Each individual has its own genotype (chromosomes and loci), sex, disease status and some other auxiliary information.

genotypic structure refers to the number of chromosomes, number and location of loci on each chromosome, name of alleles, maximum number of alleles. Individuals in the same population must have the same genotypic structure.

population collection of individuals of the same type (genotypic structure) with subpopulation structure. A population object is associated with some variables that store population statistics calculated by operators.

variables are associated with populations. They are dynamically generated by operators and can be accessed from Python namespace. This is how users obtain population statistics during evolution.

Mating scheme how individuals are chosen and mated during evolution.

operator operators are objects that manipulate populations. They can apply genetic forces like mutation, recombination, migration to populations, calculate population statistics, plot dynamics of variables, save populations or terminate simulation conditionally. Operators can be

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

simulator simulator manage several replicates of *populations* and evolve them in a way specified by a *mating scheme* generation by generation, subject to arbitrary number/kinds of *operators*. There is nothing stops you from evolving an population manually but simulator simplifies this process a lot.

expression and statement are python expression and statement. They are widely used in `simuPOP` to specify dynamic parameters, calculate statistics etc.

3.2 A simple example

Let us demonstrate these concepts through a simple example. The following is a log file of an interactive Python session. User input text after the `>>>` prompt and Python will intepret and run your command interactively.

Example 4 A simple example

```
>>> from simuPOP import *
>>> from simuRPy import *
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[2]),
...     randomMating(),
...     rep = 3)
>>> simu.evolve(
...     preOps = [initByValue([1,2,2,1])],
...     ops = [
...         recombinator(rate=0.1),
...         stat(LD=[0,1]),
...         varPlotter('LD[0][1]', numRep=3,
...                    ylim=[0,.25], xlab='generation',
...                    ylab='D', title='LD Decay'),
...         pyEval(r"'%3d   ' % gen", rep=0, step=25),
...         pyEval(r"'%f    ' % LD[0][1]", step=25),
...         pyEval(r"'\n'", rep=REP_LAST, step=25)
...     ],
...     end=100
... )
  0   0.201256    0.196476    0.200503
 25   0.021431    0.030546    0.028778
 50   0.003723    0.013669    0.004178
 75   0.006802    0.002449    0.010503
100   0.006348    0.011034    0.022168
True
>>> r.dev_print(file='log/LDdecay.eps')
{'X11': 2}
>>>
```

This example demonstrates the dynamics of linkage disequilibrium when recombination is in effect.

- The `import` line import simuPOP module (output suppressed). `simuRPy` defines a pure-python operator `varPlotter` that plot given variable using R.
- `simulator` creates a simulator from a population created by the `population` function. The population is diploid (`ploidy=2`), has 1000 individuals (`size=1000`) each has two loci on the first chromosome (`loci=[2]`). The simulator has three copies of this population (`rep=3`) and will evolve through random mating (`randomMating()`).
- `simu.evolve` evolves these populations subject to the following operators.

- `preop=[initByValue]`: operators in parameter `preop` (accept a list of operators) will be applied to the populations at the beginning of evolution. `initByValue` is an `initializer` that set the same genotype to all individuals. In this case, everyone will have genotype `12/21` (`1 2` on the first chromosome and `2 1` on the second copy of the chromosome) so linkage disequilibrium is 0.25 (maximum possible value).
- operators in `ops` parameter will be applied to all populations at each generation. (Not exactly, operators can be inactive at certain generations.)
- `recombinator` is a *during-mating operator* that recombine chromosomes with probability 0.1 (an unrealistically high value) during mating.
- `stat` is a *post-mating* operator. Parameter `LD=[0,1]` tells the operator to calculate the linkage disequilibrium between locus 0 and 1 (note the 0 index of loci). When this operator is applied to a population, it will calcuate the LD for the population and store the result in the population's local variable namespace. For this specific case, variables LD, LD_prime and R2 will be set.
- `varPlotter` is a pure python operator that plot variable `LD[0][1]` for each replicate of the populations. Title, labels on the x, y axis, and a wealth of other options can be set.
- `pyEval` accepts any python expression, evaluate it in each replicates' local namespace and return the result. In this example, `pyEval` get the value of `gen` (generation number), `LD[0][1]` and print them. Note the we use `rep` parameter to let operators apply to first (`rep=0`), last (`rep=REP_LAST`) or all (no `rep`) replicates and result in a table. We also use step=25 to apply these operators at 25 generations interval.
- `end=100`: evolve 100 generations (actually 0 - 100, 101 generations).
- `r.dev_print`: is a direct call to the rpy module. This line saves the figure to a file ld/LDdecay.eps. Note that `'.'` in R function names need to be replaced by `'_'`.

The output is a table of LD values of each replicate at 0, 25, 50, 57 and 100 generations, as well as a figure at generation 100 (figure).

All simuPOP scripts will have similar steps. You can add more operators to the `ops` list to build more complicated simulations. Obvious choices are `mutator`, `migrator`, or some proper visualizer to plot the dynamics of variables.

3.3 Genotype structure

Genotypic structure refers to the number of copies of basic number of chromosomes, number of chromosomes, existence of sex chromosome, number of loci on each chromosome, locus location on chromosome and allele names. It presents the common genetic configureation for all the individuals in a population. Note that loci are usually accessed by their absolute index regardless of chromosome structure.

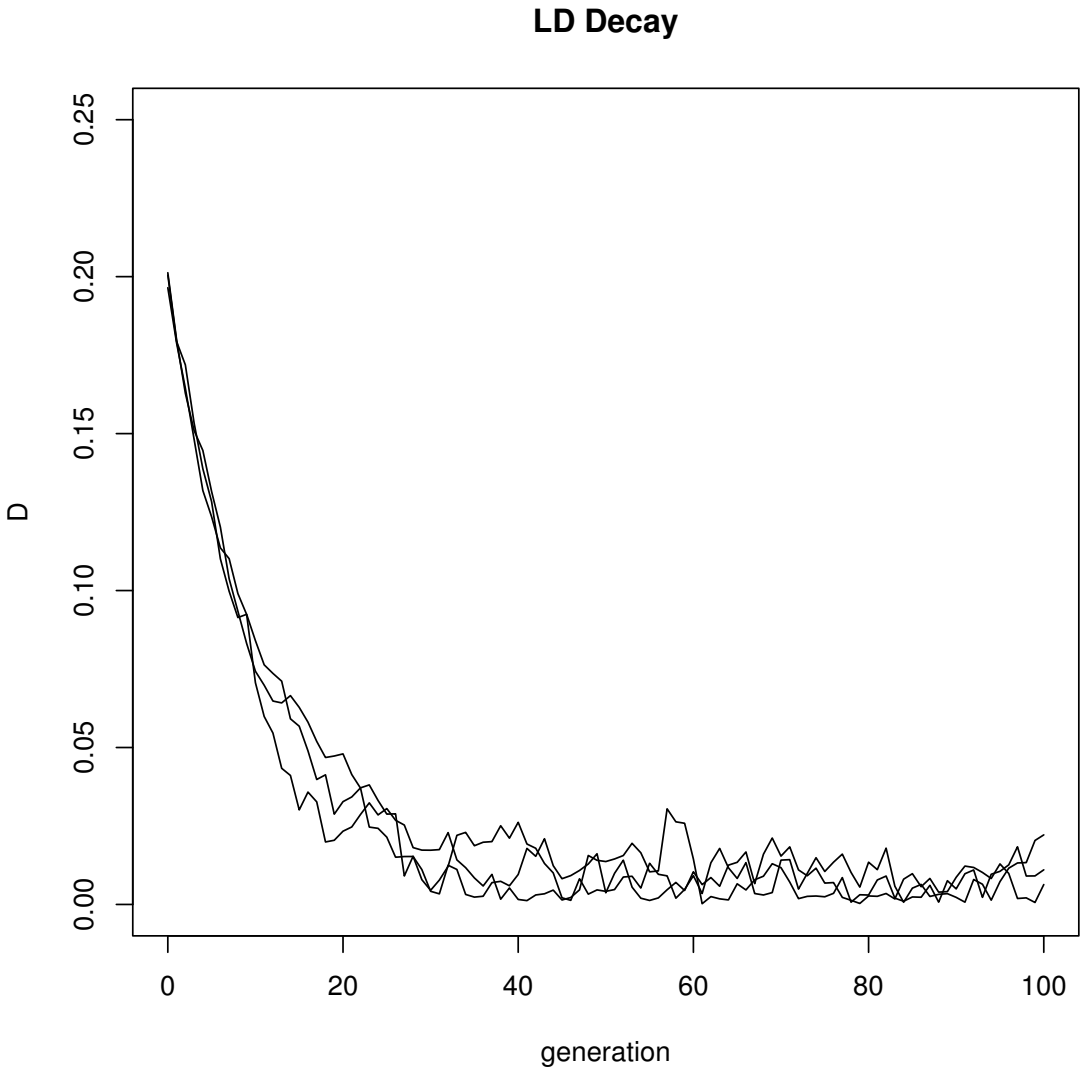
Individuals in the same population share the same genotypic structure. Consequently, *genotypic information can be retrieved from individual, population and simulator* (consists of populations with the same genotypic structure) *level*. Although there are a number of genotype structure related function (see reference manual for details), you seldom need to use them directly.

3.4 Population

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

simuPOP uses one-level population structure. That is to say, there is no sub-subpopulation or families in subpopulations. Mating is within subpopulations only. Exchange of genetic information across subpopulations can only be done through migration. Population and subpopulation sizes can be changed, as a result of mating or migration.

Figure 3.1: LD decay example: saved figure at generation 100



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

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

- access genotype structure
- access invididuals and their genotypes
- manipulate subpopulations
- access ancestral generations
- manipulate genotype

- sample (subset) from the population
- access population variables
- save/load populations in various formats

Again, you usually do not need to use these functions explicitly unless you need to write pure python functions/operators that involves complicated manipulation of populations.

3.5 Individuals

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

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

3.6 Population Variables

Populations are associated with python variables. These variables are usually set by various operators. For example, `stat` operator calculates many population statistics and store results in population namespace. However, you can also make use of this mechanisum to pass parameters, store variables etc. The interface functions are `population::vars()` and `population::dvars()` function. They are identical except that `vars()` returns a python dictionary and `dvars()` returns a wrapper class so that you can access this dictionary as attributes. For example, `pop.vars()['alleleFreq'][0] is the same as pop.dvars().alleleFreq[0]`. To have a look at all associated variables with a population, you can print pop.vars(), or better pass `pop.vars()` to a function `ListVars()`. A nice GUI will be used if wxPython is installed.

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

3.7 Mating Scheme

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

- change population/subpopulation sizes. This is where demographic models are handled in simuPOP. There are a few methods to control population sizes. The most flexible one is through a user-provided function that returns population (subpopulation) sizes at each generation.
- Randomly choose parent(s) to generate offsprings to fill the next generation. The number of offspring per mating event can be controlled. This can be a fixed number (default to 1), or a random number following one of geometric, poisson or binomial distribution.

- During-mating operators are applied to all offsprings. The most commonly used during mating operator is a recombinator that can recombine parental chromosomes and form offspring genotype.
- Apply selection if applicable. If individual fitness are given (usually returned by a selector operator), a mating scheme will choose an individual to mate, according to its relative fitness.

A few mating schemes are available, among which `randomMating()` is the most important.

3.8 Operators

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

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

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

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

3.9 Simulator

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

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

CHAPTER

FOUR

Programming simuPOP

4.1 conventions of simuPOP scripts

Although you can treat simuPOP as a regular python module and use it in whatever way you have got used to, all bundled simuPOP scripts follow the same set of conventions. If you follow the style guid, your simuPOP script should be able to

- use a dialog to input parameters (when tkinter or wxPython is installed)
- use -h or –help to view help information and description of options
- use `--noDialog` to suppress parameter dialog and input parameters through short or long command line arguments, a configuration file, or input when being prompted. A default value will be used if you press enter directly.
- be able to save currently used parameters into a configuration file (`--saveConfig`) and reuse it through (`-c` or `--config` option)

Let us look at one example `simuLDDecay.py` closely. This is one of the scripts located in `c:\python\share\simuPOP\scripts\`. 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

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

  to load parameters from the config file.

4.2 Structure of simuPOP scripts

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.

4. Optional auxillary functions

5. Optional main evolution function

   ```
   def simulation(....)
   ```

6. Executable part:

   ```
   if __name__ == ’__main__’:
     allParam = simuOpt.getParam(options,
       ”’ A short description ”’, __doc__)
     # if user press cancel,
     if len(allParam) == 0:
       sys.exit(1)
     # -h or --help
     if allParam[0]:
       print simuOpt.usage(options, __doc__)
       sys.exit(0)
     # sace configuration, something like
     if allParam[-2] != None:
        simuOpt.saveConfig(options, allParam[-2]+’.cfg’, allParam)
     # get the parameters, something like
     N = allParam[1]
     # run the simulation
     simulation(N)
   ```

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. For a complete reference of the Options structure, please refer to the reference manual.

4.3 Using 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), 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. Each of these operators accept a number of parameters that allow it to be applied at any given stage of a life-cycle, at any generation and so on.

4.3.1 Use of hybrid opertor

Example 5 An example of hybrid operators

```
def myPenetrance(geno):
'return penetrance given genotype at spcified disease loci'
if geno.count(1) < 3:
return 0.
else:
return 1-(1-(geno[0]+geno[1])*0.25)* \
(1-(geno[2]+geno[3])*0.25)*(1-(geno[4]+geno[5])*0.25)
# now, use this function in a hybrid operator as follows
pyPenetrance( atLoci=[10,20,30], func=myPenetrance)
```

Despite the large number of built-in operators, it is obviously not possible to implement every genetics models available. For example, although simuPOP provides several penetrance models, a user may want to try a customized one. In this case, one can use a simuPOP feature called *hybrid operator*. Such operators accept a Python function and will call this function with appropriate parameter(s) when needed. For example, Algorithm **??** defines a three-locus heterogeneity penetrance model (**?**) 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 (`atLoci`) and send them to function `myPenetrance` and evaluate. The return value will be used as the penetrance value of the individual.

4.3.2 Use of pure python operator

If hybrid operators are still not flexible enough, it is possible to write operators in Python. Such operators will have full access to the evolving population, and can therefore perform arbitrary operations to it. For example, one can define a frequency-dependent selection operator (see Algorithm **??**) that have different selection pressures depending on current disease allele frequency. Of course, to use such operators, one should have a deeper understanding of simuPOP.

Example 6 A frequency dependent selection operator

```
def freqDependSelector(pop):
”’ This selector will try to control disease allele
frequency by applying advantage/purifying selection
to DSL according to allele frequency at each DSL. ”’
# parameters are stored with population
DSL = pop.dvars().DSL
# Calculate allele frequency
Stat(pop, alleleFreq=[DSL])
# apply advantage/purifying selection accordingly
if 1-pop.dvars().alleleFreq[DSL][1] < pop.dvars().minAlleleFreq:
MaSelector(pop, locus=DSL, fitness=[1,1.5,2])
elif 1-pop.dvars().alleleFreq[DSL][1] > pop.dvars().maxAlleleFreq:
MaSelector(pop, locus=DSL, fitness=[1,0.9,0.8])
return True
#
# Then, use this operator like
pyOperator(func=freqDependSelector, begin=100, end=200)
```

CHAPTER

FIVE

Some Real Examples

5.1 Decay of Linkage Disequilibrium

```
>>> #
>>> # this is an example of observing decay of LD
>>> from simuUtil import *
>>> from simuRPy import *
>>>
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[2]),
...     randomMating(), rep=4 )
>>>
>>> # see the change of allele/genotype/heplotype numbers as
>>> # the result of genetic drift.
>>> init = initByValue([1,2,2,1])
>>> count = stat(LD=[0,1])
>>> recombine = recombinator( rate=0.1 )
>>> simu.evolve([
...    recombine, count,
...    pyEval(r'"%.4f\t" % LD[0][1]'),
...    endl(rep=REP_LAST),
...    #varPlotter(expr='LD[0][1]', title='Linkage disequilibrium',
...    #  numRep = 4, ytitle='LD', saveAs='LD')
...    ], preOps=[init],
...    end=10
... )
0.2000 0.1995 0.1954 0.1992
0.1767 0.1792 0.1730 0.1836
0.1621 0.1635 0.1617 0.1631
0.1481 0.1450 0.1431 0.1363
0.1388 0.1343 0.1249 0.1283
0.1238 0.1140 0.1037 0.1138
0.1184 0.0942 0.0904 0.1015
0.1101 0.0791 0.0870 0.0981
0.1006 0.0830 0.0802 0.1010
0.0969 0.0676 0.0772 0.0861
0.0866 0.0589 0.0735 0.0830
True
>>>
>>>
```

Hopefully, the program is not too difficult to understand.

5.2 Recombinator, Mutator, Migrator ...

Here is an example when all genetic forces are in effect:

```
>>> #
>>>
>>> numSubPop = 100     # number of archipelagos
>>> numFamilies = 10    # real simulation uses 1000
>>> numOffspring = 4   # kind of family size
>>> numReplicate = 1
>>> loci = [20]*20      # 400 loci on 20 chromosomes
>>> endGen = 10         # should be at leat 1000
>>> maxAllele = 30
>>> mutationRate = 0.001
>>> recombinationRate = 0.02
>>>
>>> popSize = numFamilies*numOffspring*numSubPop
>>> subPopSize = [numFamilies*numOffspring]*numSubPop
>>>
>>> # intializer
>>> init = initByFreq( alleleFreq=[1./maxAllele]*maxAllele )
>>>
>>> # migration: island model
>>> #   by proportion, .1 to all others
>>> #
>>> migrRate = .1
>>> # rate[i->i] will be ignored so we can do the following
>>> migrRates = [[migrRate/(numSubPop-1)]*numSubPop]*numSubPop
>>> migrMode  = MigrByProbability
>>> #
>>> migrate =  migrator(migrRates, mode=migrMode)
>>>
>>> # mutation
>>> mutate = kamMutator(rate=mutationRate, maxAllele=maxAllele)
>>>
>>> # recombination
>>> recombine = recombinator( rate = recombinationRate )
>>>
>>> # create a simulator
>>> simu = simulator(
...     population(size=popSize, ploidy=2, loci=loci,
...       subPop=subPopSize),
...     randomMating(numOffspring = numOffspring,
...                          newSubPopSize=subPopSize) )
>>> #
>>> # evolve
>>> simu.evolve([
...     migrate,
...     recombine,
...     mutate,
...     pyEval(r"gen", rep=0),  # report progress
...     endl(rep=REP_LAST)
...     ],
...     preOps=[init],
...     end=endGen)
0
1
2
```

```
3
4
5
6
7
8
9
10
True
>>>
>>>
>>>
```

5.3 Complex Migration Scheme

The following is a demonstration of dynamic population number/size change. Based on the same idea, we can simulate very complicated models like the 'out of africa' model. Here is what this model does:

- There are 6 cities along a line.
- Migration happens only between adjacent cities at a rate of 0.1 (0.05 each if there are two adjacent cities).
- Population size at each city will grow by a factor of 1.2 each time. But when the subpopulation size exceeds 1000, starvation :-) will cut the subpop size by half.
- Initially, everyone is in the 3th city.

The following script describe the rules almost literally:

```
>>> # this is an example of complex population size change.
>>> # for endl and tab
>>> from simuUtil import *
>>>
>>> #number of cities
>>> nc = 6
>>>
>>> # how to change subpop size?
>>> def changeSPSize(gen, oldSize):
...   size = [0]*len(oldSize)
...   for i in range(0, len(size)):
...     size[i] = oldSize[i]*1.2
...   if size[i] > 1000:
...     size[i] /= 2
...   return size
...
>>> # migration between subpopulaitons
>>> import numpy
>>> rates = numpy.array([[0.]*nc]*nc)
>>> for i in range(1,nc-1):
...   rates[i][i+1]=0.05
...   rates[i][i-1]=0.05
...
>>> #
>>> rates[0][1] = 0.1
>>> rates[nc-1][nc-2] = 0.1
>>>
```

```
>>> # print rates
>>> print rates
[[ 0.    0.1   0.    0.    0.    0.  ]
 [ 0.05  0.    0.05  0.    0.    0.  ]
 [ 0.    0.05  0.    0.05  0.    0.  ]
 [ 0.    0.    0.05  0.    0.05  0.  ]
 [ 0.    0.    0.    0.05  0.    0.05]
 [ 0.    0.    0.    0.    0.1   0.  ]]
>>> migr = migrator( rate=rates, mode=MigrByProbability)
>>>
>>> # initially, we need to set everyone to middle subpop
>>> initMigr = migrator(rate=[[1]], mode=MigrByProportion,
...        fromSubPop=[0], toSubPop=[nc/2])
>>>
>>> pop = population(size=500)
>>>
>>> # the new popsize relies on a variable newSPSize
>>> # which is calculated from subPopSize bu newSize operator
>>> simu = simulator(pop,
...     randomMating(newSubPopSizeFunc=changeSPSize) )
>>>
>>> # evolve!
>>> simu.evolve(
...   [migr, stat(popSize=True),
...    pyEval('list(subPopSize)'), endl()],
...   preOps = [ initMigr ], end=10
...   )
[0, 0, 32, 517, 50]
[0, 0, 68, 559, 88, 2]
[0, 7, 105, 613, 126, 8]
[0, 12, 147, 690, 166, 14]
[0, 19, 192, 772, 225, 25]
[0, 36, 240, 853, 306, 44]
[2, 57, 314, 949, 394, 56]
[8, 90, 392, 1069, 478, 87]
[13, 127, 505, 1183, 602, 117]
[18, 165, 627, 1352, 727, 165]
[24, 212, 745, 1576, 886, 219]
True
>>>
>>>
```

and you can see the change of population number/sizes clearly.

It should not be difficult to add recombinator, selectors to this model. Tracing the spreading of genetic diseases should also be possible, but this is out of the scope of this user's guide.

5.4 Association Mapping with Genomic Control

This example demonstrates how to generate SNP datasets and analyze them using genomic control method. (**??**)

There are several other applications that can generate SNP datasets (e.g. SNPsim **?**). These methods are coalescent based and can simulate datasets under certain mutation and recombination models. It would be easy to generate datasets using these applications but simuPOP has the following advantages:

- simuPOP can keep track of details of ancestral generations so it is possible to perform various analysis multiple times. For example, you can trace the formation of haplotype blocks or test the power of association method as

a function of generation.

- simuPOP can simulate selection and many other complicated scenarios. It is easy to add more genetic forces and observe their impact on your study.

5.4.1 Genotypic structure and Initial Population (incomplete)

For SNP datasets, we can simulate loci with two (1/2) or four (A/C/T/G) allelic states. Since we will have at most 2 alleleic states at each locus and it does not matter exactly what two states a lcous has, the first one makes more sense. If you would like to simulate four allelic states, you will have to use the `states` option of mutators so that alleles will mutate back and force in these states.

This example will initialize the population with genotype of a single individual. Linkage disequlibrium is at its highest at first and will break down with time. Note that we need to make sure initial individuals are heterozygous at disease susceptibility locus so LD will exist between this locus and others.

5.4.2 Mutation model

Coalescent based applications usually use 'infiite-site model' to perform mutation. In such simulations, once a mutation happens on the coalescent tree, it will definitely be passed to the final generation. This makes infinite-site model very appealing both in theory and in practics. However, in a forward-based simulation, a mutation may get lost very quickly so what is 'infinite-site' becomes unclear. There is also no sensible choice how to implement this model: 'mutation will not happen at a site that has been mutated before' does not make sense in biolody!

To avoid these troubles, I choose a Juke-Cantor model **?** (essentially a K-allele model) with two allelic states. I.e., allele 1 and 2 will mutate to each other with equal probability.

5.4.3 Recombination

Uniform recombination with rate 0.0001 will be used. Although non-uniform recombination can be applied easily. (Use the array form of parameter `rate`.)

5.5 Does rapid population growth lead to common disease/common variant in human population?

Reich and Lander's 2001 paper "On the allelic spectrum of human disease" (Trends in Genetics, 17(0):502-510) proposed a population genetics framework to model the evolution of allelic spectra. 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 will use simuPOP to simulate this evolution process and observe the allelic spectra of both diseases.

5.5.1 Population expansion

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. Other growth patterns are also simulated.

Actually human population took 720 ~ 6000 generations to reach a population of size 6 billion. There is no way to simulate such a huge population – the biggest population my workstation (equipped with 2G RAM) can handle is around 50 million. However, mating of really human population is far from random which implies a much smaller effected population size. As it turns out, a final population size of 1 million is enough to demonstrate the model.

5.5.2 Mutation model

The maximum number of alleles at each locus is set to be 255, a number that is hopefully big enough to mimic the infinite allele model. Allele 1 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, the same for common and complex disease, and regardless of current allelic state. This rate is not $P(A \to a)$. Instead, it is 'probability to mutate' regardless of current allelic state. Consequently,

$$P(A \to a) = P(A \to a \mid A)\, P(A)\, P(\text{mutate}) = \mu p$$

since $P(A \to a) = 1$ and $P(A)$ is the allele frequency of allele A. For rare disease, $p \sim 1$ so $\mu \sim P(A \to a)$.

Note that I can also use $\mu = 3.2 \times 10^{-6}$ as suggested in the paper, at a cost of longer simulation time.

5.5.3 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., $(1-s_c) \times (1-s_r) = 0.09$.

The choice of s_c and s_r seems to be appropriate because allele frequencies of disease alleles for both diseases remain largely unchanged during the first 500 generations. This suggests that both diseases are in mutation-selection equilibrium.

5.5.4 Implementation

The python script is short and well commented. It translates the above specifications word by word into Python/simuPOP. The plots are drawn by R. All I did was assign variables to R workspace by `r.assign` function and execute a big trunk of R code through `r('')` function. This part of source code is omited since it is irrelevant to simuPOP.

```
#!/usr/bin/env python
# simulation for Reich's paper:
#   On the allelic spectrum of human disease
#
import simuOpt
# optimized should be false during testing
simuOpt.setOptions(optimized=True)
from simuPOP import *
from simuRPy import *    # use R for plotting
```

```
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

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]]

## five different population change scenarios
#
# constant small population size
def c_small(gen, oldSize=[]):
  return [initSize]

# constant large population size
def c_large(gen, oldSize=[]):
  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]

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

# expoenential growth after burn-in
def exp_exp(gen, oldSize=[]):
  if gen < burnin:
    return [initSize]
  elif gen%10 != 0:
    return oldSize
  else:
    incRate = (math.log(finalSize)-math.log(initSize))/((endGen-burnin)/10)
    return  [int(initSize*math.exp((gen-burnin)/10.*incRate))]

def simulate(incSenario):
  simu = simulator(                                  # create a simulator
    population(subPop=incSenario(0), loci=[1,1]),    # inital population
    randomMating(newSubPopSizeFunc=incSenario)       # random mating
  )
  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
      # report population size, for monitoring purpose only
      # count allele frequencies at both loci
      stat(popSize=True, alleleFreq=[0,1]),
      # report generation and popsize
      pyEval(r"'%d\t%d\n' % (gen, popSize)", step=5),
      # 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=[1]),
        maSelector(locus=1, fitness=[1,1,1-R_s], wildtype=[1])
      ], mode=SEL_Multiplicative),
      # visualization of allelic frequencies/spectra
      # use a freqPlotter defined in simuRPy.py
      pyEval(
        # run once when this operator is created
        preStmts='''p=freqPlotter(max=max_allele, y1max=C_f0*1.5, y2max=R_f0*1.5,
          x1lab="common disease alleles", x2lab="rare disease alleles",
          save=10, name=psName)''',
        # plot allele frequencies every 10 generations
        stmts='p.plot(gen, popSize, alleleFreq[0], alleleFreq[1])',
          step=10),
      # monitor time
      ticToc(step=5),
      # pause at any user key input (for presentation purpose)
      pause(stopOnKeyStroke=1)
    ],
    end=endGen
  )

#simulate(c_small)
#simulate(c_large)
simulate(lin_exp)
#simulate(ins_exp)
#simulate(exp_exp)
```

5.5.5 Results

The simulation results match Reich's paper well. The allelic spectra of common disease remain largely unchanged during simulation while rare disease spectra become complex over time.

INDEX