simuPOP Reference Manual

Release 0.7.11 (Rev: 1132)

Bo Peng

December 2004

Last modified
26th July 2007

Department of Epidemiology, U.T. MD Anderson Cancer Center
Email: bpeng@mdanderson.org
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.

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 operators, simulators and gene mapping methods) to construct a simulation. This provides a R/Splus 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 is written in C++ and is provided as Python modules. Besides a front-end providing an interactive shell and a scripting language, Python is used extensively to pass dynamic parameters, calculate complex statistics and write operators. Because of the openness of simuPOP and Python, users can make use of a wide variety of tools (Splus/R, Python/SciPy, MatPlotLib etc.) to perform tasks like statistical analysis, gene mapping and visualization. Depend on machine configuration, simuPOP can simulate large (think in millions) populations at reasonable speed.

This is a reference manual to all variables, functions, and operators. This should be read after you learned the structure of simuPOP and how to write a simuPOP script from the *simuPOP user's guide.*

How to cite simuPOP:

> Bo Peng and Marek Kimmel (2005) simuPOP: a forward-time population genetics simulation environment. *bioinformatics*, **21**(18): 3686-3687

CONTENTS

LISTINGS

CHAPTER

ONE

Genotypic structure

Genotypic structure refers to

- ploidy, the number of copies of basic number of chromosomes (c.f. `ploidy(), ploidyName()`)
- number of chromosomes (c.f. `numChrom()`)
- existence of sex chromosome (c.f. `sexChrom()`)
- number of loci on each chromosome (c.f. `numLoci(ch), totNumLoci()`)
- locus location on chromosome (c.f. `locusPos(loc), arrlociPos()`)
- allele names, default to allele number (c.f. `alleleName(allele)`)
- maximum allele state (c.f. `maxAllele()`)
- name of the information fields (c.f. `infoField(idx), infoFields()`)

Here information fields refer to the numbers attached to each individual, such as fitness value, parent index, age etc. Individuals may need some information fields to use certain operators. For example, `'fitness'` field is required by all selectors.

Example 1.1 creates a population and displays some of genotypic information.

Listing 1.1: Genotype structure functions

```
>>> # create a population, most parameters have default values
>>> pop = population(size=5, ploidy=2, loci=[5,10],
...     lociPos=[range(0,5),range(0,20,2)],
...     alleleNames=['A','C','T','G'],
...     subPop=[2,3], maxAllele=3)
>>> print pop.popSize()
5
>>> print pop.ploidy()
2
>>> print pop.ploidyName()
diploid
>>> print pop.numChrom()
2
>>> print pop.locusPos(2)
2.0
>>> print pop.alleleName(1)
C
>>>
```

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*.

Listing 1.2: Calling genotype structure functions from individual or simulator

```
>>> # get the fourth individual of the population
>>> ind = pop.individual(3)
>>> # access genotypic structure info
>>> print ind.ploidy()
2
>>> print ind.numChrom()
2
>>> print ind.numLoci(0)
5
>>> print ind.genoSize()
30
>>> # and from simulator level
>>> simu = simulator(pop, randomMating(), rep=3)
>>> print simu.numChrom()
2
>>>
```

You may have noticed that locus Indices start from 0. **As a matter of fact, all arrays in simuPOP start at index 0**. To avoid confusion, I will refer the first locus as locus zero, second locus as locus one; first individual in a population as individual zero, and so on. The reason why zero-based indices are used is because C++ and Python, using which simuPOP is built, are both zero-based.

Another concern is how we should refer to loci on different chromosomes. The solution is that we almost always use *absolute index* and seldom *use relative index*. For example, if there are five and seven loci on the first two chromosomes, the absolute indices of loci will be (0,1,2,3,4), (5,6,7,8,9,10,11). It may feel confusing at first but this avoids the trouble of having to use two numbers (chrom, index) to refer to a locus. If relative index is needed, functions `chromLocusPair(absIndex)` and `absLocusIndex(chrom,index)` can be used.

Listing 1.3: Conversion between absolute and relative indices

```
>>> print pop.chromLocusPair(7)
(1, 2)
>>> print pop.absLocusIndex(1,1)
6
>>>
```

1.1 Class reference

To display the following manual item in python, you can use `help(population)` or `help(individual)` or directly `help(GenoStruTrait)`. (You may notice that both `population` and `individual` classes are inherited from `GenoStruTrait` class.)

Class `GenoStruTrait`

genotypic structure related functions, can be accessed from both individuals and populations

Details

Genotypic structure refers to the number of chromosomes, positions, the number of loci on each chromosome, and allele and locus names etc. All individuals in a population share the same genotypic structure and functions provided in this class can be accessed from individual, population and simulator levels.

Initialization

Creat a `GenoStruTrait` class, but `m_genoStruIdx` will be set later.

```
GenoStruTrait()
```

Member Functions

`x.absLocusIndex(chrom, locus)` return the absolute index of a locus on a chromosome

`x.alleleName(allele)` return the name of an allele (if previously specified)

`x.alleleNames()` return an array of allelic names

`x.arrLociPos()` return an (editable) array of loci positions of all loci

Note: Modifying loci position directly using this function is strongly discouraged.

`x.arrLociPos(chrom)` return an array of loci positions on a given chromosome

Note: Modifying loci position directly using this function is strongly discouraged.

`x.chromBegin(chrom)` return the index of the first locus on a chromosome

`x.chromEnd(chrom)` return the index of the last locus on a chromosome plus 1

`x.chromIndex()` return an array of chromosome indices

`x.chromLocusPair(locus)` return a `(chrom, locus)` pair of an absolute locus index

`x.chromMap()` return the distribution of chromosomes across multiple nodes (MPI version of simuPOP only)

`x.genoSize()` return the total number of loci times ploidy

`x.hasInfoField(name)` determine if an information field exists

`x.infoField(idx)` obtain the name of information field `idx`

`x.infoFields()` return an array of all information fields

`x.infoIdx(name)` return the index of the field `name`, return `-1` if not found

`x.infoSize()` obtain the number of information fields

`x.lociByNames(names)` return an array of locus indices by locus names

`x.lociNames()` return names of all loci

`x.lociPos()` return loci positions

`x.locusByName(name)` return the index of a locus by its locus name

`x.locusName(loc)` return the name of a locus

`x.locusPos(locus)` return the position of a locus

`x.maxAllele()` return the maximum allele value for all loci

`x.numChrom()` return the number of chromosomes

`x.numLoci(chrom)` return the number of loci on chromosome `chrom`, equals to `numLoci()[chrom]`

`x.numLoci()` return the number of loci on all chromosomes

x.ploidy() return ploidy, the number of homologous sets of chromosomes

x.ploidyName() return ploidy name, `haploid`, `diploid`, or `triploid` etc.

x.setMaxAllele(maxAllele) set the maximum allele value for all loci

Maximum allele value has to be `1` for binary modules. `maxAllele` is the maximum possible allele value, which allows `maxAllele+1` alleles `0, 1, ..., maxAllele`.

x.sexChrom() determine whether or not the last chromosome is sex chromosome

x.totNumLoci() return the total number of loci on all chromosomes

1.2 Sex chromosome

If `sexChrom()` is false, all chromosomes are assumed to be autosomes. You can also create population/individuals with a sex chromosome. Please note that we currently only model the XY chromosomes in diploid population. Consequently,

- Sex chromosome is always the last chromosome.
- Sex chromosome can only be specified for diploid population. (`ploidy()`=2)
- Sex chromosomes (XY) may differ in length. You should specify the length of the longer one as the chromosome length. If there are more loci on X than Y, the rest of the Y chromosome is unused. Mutation, recombination may still occur at this unused part of chromosome to simplify implementation and usage.
- It is assumed that male has XY and female has XX chromosomes. The sex chromosomes of male individuals will be arranged in the order of XY.

1.3 Information fields

An individual will by default have genotype, sex and affectedness information, but other information is needed for some operations. For example, the fitness value of an individual is needed for selection, one or more trait values may be needed to calculate quantitative traits; and age may be needed if age-dependent mating schemes are used. Since the need for information fields varies from simulation to simulation, simuPOP does not fix the amount of information fields, and allow users to specify these fields during the construction of populations.

Operators may require certain information field to work properly. For example, all selectors require field `fitness` to store evaluated fitness values for each individual. `parentTagger` needs `father_idx` and `mother_idx` to store index of the parents of each individual in the parental generation. You do need to add these fields to the `infoFields` parameter of the population constructor. If you forget, an error message will appear and tells you to add certain field when a field is needed by some operators.

The information fields can be access from each individual (c.f. `info(idx), info(name), setInfo(value, idx), setInfo(value, name), arrInfo()` of individuals), or from the population as a whole (c.f. `setindInfo(value), arrIndInfo(subPop)`). Some operators allows you to specify which information field(s) to use. Just to show what you can do with information fields, one can

- create a population with 5 trait values and some risk factors
- assign risk factors (environmental maybe) manually, or through some operators
- calculate each trait values using different quantitative trait operators, some gene may contribute to more than one trait values
- calualte a final trait values from these information fields.

CHAPTER

TWO

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.

2.1 Population overview

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. More specifically

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

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

Every population has its own variable space, or *local namespaces* in simuPOP term. This namespace is a Python dictionary that is attached to each population and can be exposed to the users through `vars()` or `dvars()` function. Many functions and operators work in these namespaces and store their results in them. For example, function `Stat` set variables like `alleleFreq[loc]` and you can access them like `pop.dvars().alleleFreq[loc][allele]`.

Population has a large number of member functions, ranging from reviewing simple property to generating new population from the current one. However, you do not have to know all the member functions to use a population. As a matter of fact, you will only use a small portion of these functions unless you need to write pure python functions/operators that involves complicated manipulation of populations.

2.2 Class reference

Class `population`

a collection of individuals with the same genotypic structure

Details

simuPOP populations consists of individuals of the same genotypic structure, which refers to the number of chromosomes, number and position of loci on each chromosome etc. The most important components of a population are:

- subpopulation. A population is divided into subpopulations (unstructured population has a single subpopulation, which is the whole population itself). Subpopulation structure limits the usually random exchange of genotypes between individuals disallowing mating between individuals from different subpopulations. In the presence of subpopualtion structure, exchange of genetic information across subpopulations can only be done through migration. Note that in simuPOP there is no sub-subpopulation or family in subpopulations.

- variables. Every population has its own variable space, or *localnamespaces* in simuPOP term. This namespace is a Python dictionary that is attached to each population and can be exposed to the users through `vars()` or `dvars()` function. Many functions and operators work and store their results in these namespaces. For example, function `Stat` set variables such as `alleleFreq[loc]`, and you can access it via `pop.dvars().alleleFreq[loc][allele]`.

- ancestral generations. A population can save arbitrary number of ancestral generations. During evolution, the latest several (or all) ancestral generations are saved. Functions??? to make a certain ancestral generation *current* are provided so that one can examine and modify ancestral generations.

Other important concepts like *information fields* are explained in class `individual`???.

Note

Although a large number of member functions are provided, most of the operations are performed by *operators*. These functions will only be useful when you need to manipulate a population explicitly.

Initialization

Create a population object with given size and genotypic structure.

```
population(size=0, ploidy=2, loci=[], sexChrom=False, lociPos=[],
subPop=[], ancestralDepth=0, alleleNames=[], lociNames=[],
maxAllele=MaxAllele, infoFields=[], chromMap=[])
```

FIXME: Details of constructure is missing. ???This is techniquely the `__init__` function of the population object.

alleleNames an array of allele names. The first element should be given to invalid/unknown allele. For example, for a locus with alleles A,C,T,G, you can specify `alleleNames` as `('_','A','C','T','G')`. Note that simuPOP uses `1,2,3,4` internally and these names will only be used for display purpose.

ancestralDepth number of most recent ancestral generations to keep during evolution. Default to `0`, which means only the current generation will be available. You can set it to a positive number m to keep the latest m generations in the population, or `-1` to keep all ancestral populations. Note that keeping track of all ancestral populations may quickly exhaust your computer RAM. If you really need to do that, use `savePopulation` operator to save each generation to a file is a much better choice.

infoFields name of information fields that will be attached to each individual. For example, if you need to record the parents of each individual you will need two, if you need to record the age of individual, you need an additional one. Other possibilities include offspring IDs etc. Note that you have to plan this ahead of time since, for example, tagger will need to know what info unit to use. Default to `none`.

loci an array of numbers of loci on each chromosome. The length of parameter `loci` determines the number of chromosomes. Default to [1], meaning one chromosome with a single locus.

The last chromosome can be sex chromosome. In this case, the maximum number of loci on X and Y should be provided. I.e., if there are 3 loci on Y chromosme and 5 on X chromosome, use `5`.

lociNames an array or a matrix (separated by chromosomes) of names for each loci. Default to `"locX-X"` where `X-X` is a chromosome-loci index starting from 1.

lociPos a 1-d or 2-d array specifying positions of loci on each chromosome. You can use a nested array to specify loci position for each chromosome. For example, you can use `lociPos=[1,2,3]` when `loci=[3]` or `lociPos=[[1,2],[1.5,3,5]]` for `loci=[2,3]`. simuPOP does not assume a unit for these locations, although they are usually intepreted as centiMorgans. The default values are `1`, `2`, etc. on each chromosome.

maxAllele maximum allele number. Default to the max allowed allele states of current library (standard or long allele version) maximum allele state for the whole population. This will set a cap for all loci. For individual locus, you can specify maxAllele in mutation models, which can be smaller than global maxAllele but not larger. Note that this number is the number of allele states minus 1 since allele number starts from 0.

ploidy number of sets of chromosomes. Default to `2` (diploid).

sexChrom `true` or `false`. Diploid population only. If `true`, the last homologous chromosome will be treated as sex chromosome. (XY for male and XX for female.) If X and Y have different number of loci, number of loci of the longer one of the last (sex) chromosome should be specified in `loci`.

size population size. Can be ignored if `subPop` is specified. In that case, `size` is the sum of `subPop`. Default to `0`.

subPop an array of subpopulation sizes. Default value is `[size]` which means a single subpopulation of the whole population. If both `size` and `subPop` are are provided, `subPop` should add up to `size`.

Member Functions

`x.absIndIndex(ind, subPop)` return the absolute index of an individual in a subpopulation

index index of an individual in a subpopulation `subPop`

subPop subpopulation index (start from `0`)

`x.addInfoField(field, init=0)` add an information field to a population.

field new information field. If it already exists, it will be re-initialized.

init inital value for the new field.

`x.addInfoFields(fields, init=0)` add one or more information fields to a population

fields new information fields. If one or more of the fields alreay exist, they will simply be re-initialized.

init initial value for the new fields.

`x.adjustInfoPosition(order)` simuPOP::population::adjustInfoPosition

`x.ancestralDepth()` ancestral depth of the current population

Note: The returned value is the number of ancestral generations exists in the population, not necessarily equal to the number set by setAncestralDepth().

`x.ancestralGen()` currently used ancestral population (0 for the latest generation)

Current ancestral population activated by useAncestralPop. There can be several ancestral generations in a population. `0` (current), `1` (parental) etc. When useAncestralPop(gen) is used, current generation is set to one of the parental generation, which is the information returned by this function. `useAncestralPop(0)` should always be used to set a population to its usual ancestral order.

`x.arrGenotype(order)` get the whole genotypes

Return an editable array of all genotypes of the population. You need to know how these genotypes are organized to safely read/write genotype directly. Individuals will be in order before exposing their genotypes.

order if order is `true`, respect order; otherwise, do not repect population structure.

`x.arrGenotype(subPop, order)` get the whole genotypes

Return an editable array of all genotype of a subpopulation. Individuals will be in order before exposing their genotypes.

order if order is `true`, keep order; otherwise, respect `subpop` structure.

subPop index of subpopulation (start from 0)

`x.arrIndInfo(order)` get an editable array (Python list) of all information fields

order whether or not the list has the same order as individuals

`x.clone(keepAncestralPops=-1)` deep copy of a population. (In python, `pop1 = pop` will only create a reference to `pop`.)

`x.equalTo(rhs)` compare two populations

`x.evaluate(expr="", stmts="")` evaluate a python statment/expression

This function evaluates a python statment/expression and return its result as a string. Optionally run statement first.

`x.execute(stmts="")` evaluate a statement (can be multi-line string)

`x.gen()` current generation during evolution

`x.grp()` current group ID in a simulator

Group number is not meaningful for a stand-alone population.

`x.ind(ind, subPop=0)` refernce to individual`ind` in subpopulation `subPop`

This function is named `individual` in the Python interface.

ind individual index within `subPop`

subPop subpopulation index

`x.indInfo(idx, order)` get information `idx` of all individuals.

idx index in all information fields

order if true, sort returned vector in individual order

`x.indInfo(name, order)` get information `name` of all individuals.

name name of the information field

order if true, sort returned vector in individual order

`x.indInfo(idx, subPop, order)` get information `idx` of all individuals in a subpopulation `subPop`.

idx index in all information fields

order if true, sort returned vector in individual order

subPop subpopulation index

`x.indInfo(name, subPop, order)` get information `name` of all individuals in a subpopulation `subPop`.

name name of the information field

order if true, sort returned vector in individual order

subPop subpopulation index

`x.individuals()` return an iterator that can be used to iterate through all individuals

`x.individuals(subPop)` return an iterator that can be used to iterate through all individuals in subpopulation `subPop`

`x.insertAfterLoci(idx, pos, names=[])` append loci at given locations

Append loci at some given locations. In an appended location, alleles will be zero.

idx an array of locus index. The loci will be added *after* each index. If you need to append to the first locus, please use `insertBeforeLoci`. If your index is the last locus of a chromosome, the appended locus will become the last of that chromosome. If you need to append multiple loci after a locus, please repeat that locus number.

names an array of locus names. If this parameter is not given, some unique names such as "insX_X", will be given.

pos an array of locus positions. You need to make sure that the position will make the appended locus between adjacent markers.

`x.insertAfterLocus(idx, pos, name=string)` append an locus at a given location

`insertAfterLocus(idx, pos, name)` is a shortcut to `insertAfterLoci([idx], [pos], [name])`.

`x.insertBeforeLoci(idx, pos, names=[])` insert loci at given locations

Insert loci at some given locations. In an inserted location, alleles will be zero.

idx an array of locus index. The loci will be inserted *before* each index. If you need to append to the last locus, please use `insertAfterLoci`. If your index is the first locus of a chromosome, the inserted locus will become the first of that chromosome. If you need to insert multiple loci before a locus, please repeat that locus number.

names an array of locus names. If this parameter is not given, some unique names such as "insX_X", will be given.

pos an array of locus positions. You need to make sure that the position will make the inserted locus between adjacent markers.

`x.insertBeforeLocus(idx, pos, name=string)` insert an locus at given location.

`insertBeforeLocus(idx, pos, name)` is a shortcut to `insertBeforeLoci([idx], [pos], [name])`

`x.mergePopulation(pop, newSubPopSizes=[], keepAncestralPops=-1)` merge populations by individuals

Merge individuals from `pop` to the current population. Two populations should have the same genotypic structures. By default, subpopulations of the merged populations are kept. I.e., if you merge two populations with one subpopulation, the resulting population will have two subpopulations. All ancestral generations are also merged.

keepAncestralPops ancestral populations to merge, default to all (`-1`)

newSubPopSizes subpopulation sizes can be specified. The overall size should be the combined size of the two populations. Because this parameter will be used for all ancestral generations, it may fail if ancestral generations have different sizes. To avoid this problem, you may run `mergePopulation` without this parameter, and then adjust subpopulation sizes generation by generation.

Note: Population variables are not copied to `pop`.

`x.mergePopulationByLoci(pop, newNumLoci=[], newLociPos=[], byChromosome=False)`
merge populations by loci

Two populations should have the same number of individuals. This also holds for any ancestral generations. By default, chromosomes of `pop` are added to the current population. That is to say, chromosomes from `pop` is added, as new chromosomes to this populaton. You can change this arrangement in two ways

- specify new chromosome structure using parameter `newLoci`. loci from new and old population are still in their original order, but chromosome number and position can be changed in this way.
- specify `byChromosome=true` so that chromosomes will be merged one by one. In this case, loci position of two popualtions are important because loci will be arranged in the order of loci position; and identical loci position of two loci in two populations will lead to error.

byChromosome merge chromosome by chromosome, loci are ordered by loci position default to false.

newLociPos the new loci position if number of loci on each chromosomes are changed with `newNumLoci`. On each chromosome, loci position should in order.

newNumLoci the new number of loci for the combined genotypic structure.

Note:

- Information fields are not merged.
- All ancestral generations will be merged because all individuals in a population have to have the same genotypic structure.

`x.mergeSubPops(subPops=[], removeEmptySubPops=False)` merge given subpopulations

Merge subpopulations, the first subpopulation ID (the first one in array `subPops`) will be used as the ID of the new subpopulation. That is to say, all subpopulations will take the ID of the first one.

`x.newPopByIndID(keepAncestralPops=-1, id=[], removeEmptySubPops=False)` Form a new population according to the parameter information. Information can be given directly as

- `keepAncestralPops=-1`: keep all
- `keepAncestralPops=0`: only current
- `keepAncestralPops=1`: keep one ...

`x.newPopWithPartialLoci(remove=[], keep=[])` obtain a new population with selected loci

Copy current population to a new one with selected loci and remove specified loci. (No change on the current population.)

`x.numSubPop()` number of subpopulations in a population

`x.popSize()` obtain total population size

`x.pushAndDiscard(rhs, force=False)` Absorb `rhs` population as the current generation of a population.

This is mainly used by a simulator to push offspring generation `rhs` to the current population, while the current population is pushed back as an ancestral population (if `ancestralDepath() != 0`). Because `rhs` population is swapped in, `rhs` will be empty after this operation.

`x.rearrangeLoci(newNumLoci, newLociPos)` rearrange loci on chromosomes, e.g. combine two chromosomes into one

This is used by `mergeByLoci`.

`x.removeEmptySubPops()` remove empty subpopulations by adjusting subpopulation IDs

x.removeIndividuals(inds=[], subPop=-1, removeEmptySubPops=False) remove individuals

If a valid `subPop` is given, remove individuals from this subpopulation.

x.removeLoci(remove=[], keep=[]) remove some loci from the current population. Loci that will be removed or kept can be specified.

x.removeSubPops(subPops=[], shiftSubPopID=True, removeEmptySubPops=False) remove subpopulations and adjust subpopulation IDs so that there will be no *'empty'* subpopulation left

Remove specified subpopulations (and all individuals within). If `shiftSubPopID` is false, `subPopID` will be kept intactly.

x.reorderSubPops(order=[], rank=[], removeEmptySubPops=False) reorder subpopulations by order or by rank

order new order of the subpopulations. For examples, 3 2 0 1 means `subpop3`, `subpop2`, `subpop0`, `subpop1` will be the new layout.

rank you may also specify a new rank for each subpopulation. For example, 3,2,0,1 means the original subpopulations will have new IDs 3,2,0,1, respectively. To achive order 3,2,0,1, the rank should be 1 0 2 3.

x.rep() current replicate in a simulator

Replication number is not meaningful for a stand-alone population.

x.resize(newSubPopSizes, propagate=False) resize population

Resize population by giving new subpopulation sizes.

newSubPopSizes an array of new subpopulation sizes. If there is only one subpopulation, use `[newPopSize]`.

propagate if `propagate` is `true`, copy individuals to new comers. I.e., 1, 2, 3 ==> 1, 2, 3, 1, 2, 3, 1

Note: This function only resizes the current generation.

x.savePopulation(filename, format="auto", compress=True) simuPOP::population::savePopulation

filename save to filename

format format to save. Can be one of the following: 'text', 'bin', or 'xml'. The default format is 'text' but the output is not supposed to be read. 'bin' has smaller size than the other two and should be used for large populations. 'xml' is the most readable format and should be used when you would like to convert simuPOP populations to other formats.

x.setAncestralDepth(depth) set ancestral depth.

depth `0` for none, `-1` for unlimited, a positive number sets the number of ancestral generations to save.

x.setIndInfo(values, idx) set individual information for the given information field (index),

idx index to the information field.

values an array that has the same length as population size.

x.setIndInfo(values, name) set individual information for the given information field (name)

setIndInfo using field name, `x.setIndInfo(values, name)` is equivalent to the `idx` version `x.setIndInfo(values, x.infoIdx(name))`.

x.setIndSubPopID(id) set subpopulation ID with given ID

Set subpopulation ID of each individual with given ID. Individuals can be rearranged afterwards using `setSubPopByIndID`.

id an array of the same length of population size, resprenting subpopulation ID of each individual.

x.setIndSubPopIDWithID() set subpopulation ID of each individual with their current subpopulation ID???

x.setInfoFields(fields, init=0) set information fields for an existing population. The existing fields will be removed.

fields an array of fields

init initial value for the new fields.

x.setSubPopByIndID(id=[]) adjust subpopulation according to individual subpopulation ID.

Rearrange individuals to their new subpopulations according to their subpopulation ID (or the new given `ID`). Order within each subpopulation is not respected.

id new subpopulation ID, if given, current individual subpopulation ID will be ignored.

Note: Individual with negative info will be removed!

x.setSubPopStru(newSubPopSizes, allowPopSizeChange=False) set population/subpopulation structure given subpopulation sizes

allowPopSizeChange if this parameter is `true`, population will be resized.

subPopSize an array of subpopulation sizes. The population may or may not change according to parameter `allowPopSizeChange` if the sum of `subPopSize` does not match `popSize`.

x.splitSubPop(which, sizes, subPopID=[]) split a subpopulation into subpopulations of given sizes

The sum of given sizes should be equal to the size of the split subpopulation. Subpopulation IDs can be specified. The subpopulation IDs of non-split subpopulations will be kept. For example, if subpopulation 1 of 0 1 2 3 is split into three parts, the new subpop id will be 0 (1 4 5) 2 3.

Note: `subpop` with negative ID will be removed. So, you can shrink one `subpop` by splitting and setting one of the new `subpop` with negative ID.

x.splitSubPopByProportion(which, proportions, subPopID=[]) split a subpopulation into subpopulations of given proportions

The sum of given proportions should add up to one. Subpopulation IDs can be specified.

Note: `subpop` with negative ID will be removed. So, you can shrink one `subpop` by splitting and setting one of the new `subpop` with negative ID.

x.subPopBegin(subPop) index of the first individual of a subpopulation

subPop subpopulation index

x.subPopEnd(subPop) return the value of the index of the last individual of a subpopulation plus 1

subPop subpopulation index

Note: As with all ...End functions, the returning index is out of the range so that the actual range is [xxxBegin, xxxEnd). This agrees with all STL conventions and Python range.

x.subPopIndPair(ind) return the `(sp, idx)` pair from an absolute index of an individual

x.subPopSize(subPop) return size of a subpopulation `subPop`

subPop index of subpopulation (start from 0)

x.subPopSizes() return an array of all subpopulation sizes

x.swap(rhs) swap the content of two populations

x.turnOffSelection() Turn off selection for all subpopulations.

If you really want to apply another selector, run turnOffSelection to eliminate the effect of the previous one.

x.useAncestralPop(idx) use an ancestral generation. `0` for the latest generation.

idx Index of the ancestral generation. `0` for current, `1` for parental, etc. idx can not exceed ancestral depth (see setAncestralDepth).

x.vars(subPop=-1) return variables of a population. If `subPop` is given, return a dictionary for specified subpopulation.

Examples

```
>>> # a Wright-Fisher population
>>> WF = population(size=100, ploidy=1, loci=[1])
>>>
>>> # a diploid population of size 10
>>> # there are two chromosomes with 5 and 7 loci respectively
>>> pop = population(size=10, ploidy=2, loci=[5, 7], subPop=[2, 8])
>>>
>>> # a population with SNP markers (with names A,C,T,G
>>> # range() are python functions
>>> pop = population(size=5, ploidy=2, loci=[5,10],
...     lociPos=[range(0,5),range(0,20,2)],
...     alleleNames=['A','C','T','G'],
...     subPop=[2,3], maxAllele=3)
>>>
```

2.3 Creating a population

A population can be created through

- call `population` function to create an instance of population from `population` class.
- call `LoadPopulation`, `LoadFstat` etc to load a population from a saved file.
- generated as a subset of an existing population by operators like `randomSample, caseControlSample` or equivalent functions `RandomSample, CaseControlSample`.
- Obtained from an existing simulator through `simulator::getPopulation()`

Help contents of all functions of population class can be displayed by `help(population)`. Help on a memeber function can be viewed by `help(population.func)`. In Python, constructor is named `__init__` and you can use class name to create an instance of the class. Therefore, to display parameters of `population` function, you need to run

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

Some notes about the parameters:

- `size, subPop:` size can be ignored if subPop is specified. If both parameters are provided, `subPop` should add up to `size`.
- `loci:` number of loci on each chromosome. The length of parameter `loci` determines number of chromosomes. The last chromosome can be sex chromosome. In this case, please specify the maximum number of loci on X and Y. I.e., if there are 3 loci on Y chromosme and 5 on X chromosome, use 5.
- `sexChrom:` true or false. Diploid population only. If true, the last homologous chromosomes will be treated as sex chromosomes. (XY for male and XX for female.) If X and Y have different number of loci, you should use the longer one as loci number of the last (sex) chromosome.
- `lociPos:` a 1-d or 2-d array specifying positions of loci on each chromosome. For example, you can use `lociPos=[1,2,3]` when `loci=[3]` or `lociPos=[[1,2],[1.5,3,5]]` for `loci=[2,3]`. simuPOP does not assume a unit for these locations, although they are usually intepreted as base pairs or centiMorgans, depending on types of simulation being performed. Currently, loci location is used only for specifying recombination intensity. The actual recombination rate is intensity times loci distance between adjacent loci.
- `ancestralDepth:` number of most recent ancestral generations to keep during evolution. Default to 0. You can set it to a positive number `m` to keep the latest `m` generations in the population, or -1 to keep all ancestral populations. Note that keeping track of all ancestral populations may quickly exhaust your computer RAM. If you really need to do that, use `savePopulation` operator to save each generation to a file is a much better choice.
- `alleleNames:` Names of the alleles. They are used only for output.
- `maxAllele`: maximum allele state for the whole population. This will set a cap for all loci. For individual locus, you can specify `maxAllele` in mutation models, which can be smaller than global `maxAllele` but not larger. Note that this number is thee number of allele states minus 1 since allele number starts from 0.

Example 2.1 shows a few examples of using the population function to create populations.

Listing 2.1: Use of population function

```
>>> # a Wright-Fisher population
>>> WF = population(size=100, ploidy=1, loci=[1])
>>>
>>> # a diploid population of size 10
>>> # there are two chromosomes with 5 and 7 loci respectively
>>> pop = population(size=10, ploidy=2, loci=[5, 7], subPop=[2, 8])
>>>
>>> # a population with SNP markers (with names A,C,T,G
>>> # range() are python functions
>>> pop = population(size=5, ploidy=2, loci=[5,10],
...     lociPos=[range(0,5),range(0,20,2)],
...     alleleNames=['A','C','T','G'],
...     subPop=[2,3], maxAllele=3)
>>>
```

2.4 Copying a population

Like many other python operations,

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

will create a reference `pop1` to population `pop`. Modifying `pop1` will modify `pop` as well. If you would like to have an independent copy, use

```
pop1 = pop.clone()
```

This scenario also apply to simulator (see later sections), if `simu` is a simulator with several replicates,

```
pop = simu.population(idx)
```

will get a reference to one of the replicates. You can, although not recomended, modify simulator through this `pop` reference. Note that this pop reference will become invalid when the simulator is distroyed so the following calling sequence will crash python.

```
pop = simu.population(idx)
simu = simulator(....)
pop.savePopulation(....)
```

If you would like to get a real copy, use

```
pop = simu.getPopulation(idx)
```

2.5 Interaction with Operators and Functions

Operators are objects that can be applied to populations. They have special attributes like at which generations to be active, at what stage of a evolutionary life cycle to be applied. Usually, an operator is created and passed as a parameter to a simulator. When `simulator::evolve` (or `step, apply`) is called, the simulator will call the `apply()` function of these operators at appropriate times.

Listing 2.2: Population and operators

```
>>> simu = simulator(pop, randomMating(), rep=3)
>>> simu.evolve(
...   preOps = [ initByFreq([.8, .2])],
...   ops = [
...     stat(alleleFreq=[0,1], Fst=[1], step=10),
...     kamMutator(rate=0.001, rep=1),
...     kamMutator(rate=0.0001, rep=2)
...   ],
...   end=10
... )
True
>>>
```

For example, operators `initByFreq, stat` and two copies of `kamMutator` are created in example 2.2. During evolution, `simu` will apply `initByFreq` once to each replicate of the simulator; apply the first `kamMutator` to the first replicate and the second `kamMutator` to the second replicate at every generation; apply `stat` to count allele frequency and calculate F_{st} every 10 generations. More details about operators will be described later.

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

```
initByFreq( [0.3, .2, .5] ).apply(pop)
```

or dump the content of a population by

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

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

Listing 2.3: Function InitByFreq

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

Note that

1. The following two calling sequences have different consequences:

   ```
   pop = population(10)
   simu = simulator(pop, randomMating(), rep=3)
   simu.evolve( preOps = [initByFreq([.8,.2]) ] )
   ```

   initialize three replicates of the population independently, (`initByFreq` object is applied to three populations.) but

   ```
   pop = population(10)
   InitByFreq(pop, [.8, .2])
   simu = simulator(pop, randomMating(), rep=3)
   ```

   initialize a population once and create a simulator with three copies of the initialized population.

2. If you are going to call such a function many times, it is more efficient to do

   ```
   init = initByFreq([.8,.2])
   for i in range(0,1000):
     init.apply(pop[i])
   ```

   than

   ```
   for i in range(0,1000):
     InitByFreq(pop[i], [.8,.2])
   ```

   The difference is that the second method creates and destroys an `initByFreq` object each time it calls the function.

Because `initByFreq` and `dumper` will be frequently used in this manual, I will briefly describe them here.

- `initByFreq` takes an array of probabilities (must add up to one). When applying to a population, each allele will be assigned `0,1,2,3,...` etc according to the provided probabilities. `InitByFreq(pop,...)` is its function form.
- `dumper` will simply display the population. The output format for each individual is: individual index, tag, sex, affected status, alleles on copy `1,2,...` of all chromosomes, separated by `|`. Its function form is named `Dump(pop)`.

Example 2.4 demonstrates the use of these two operators:

Listing 2.4: Operator dumper and initByFreq

```
>>> pop = population(size=5, ploidy=2, loci=[5,10],
...     lociPos=[range(0,5),range(0,20,2)],
...     alleleNames=['A','C','T','G'],
...     subPop=[2,3], maxAllele=3)
>>> # .apply form
>>> initByFreq([.2, .3, .4, .1]).apply(pop)
True
>>> # function form
>>> Dump(pop)
Ploidy:                2
Number of chrom:       2
Number of loci:        5 10
Maximum allele state:  3
Loci positions:
               0 1 2 3 4
               0 2 4 6 8 10 12 14 16 18
Loci names:
               loc1-1 loc1-2 loc1-3 loc1-4 loc1-5
               loc2-1 loc2-2 loc2-3 loc2-4 loc2-5 loc2-6 loc2-7 loc2-8 loc2-9 loc2-10
population size:       5
Number of subPop:      2
Subpop sizes:          2 3
Number of ancestral populations:       0
individual info:
sub population 0:
   0: MU TTCTA AACAATCTTC | TTGAT TGCTGATACG
   1: FU TGAAG CTTCTCCGTG | ATCAA ACTCCTCCAC
sub population 1:
   2: MU CTATC GTTTATATAA | TCGTA TAACTCTCCT
   3: FU TTTTC TACTATTGCT | TGCTA TCAGTCATCT
   4: MU GGTTA TGTCCACAAT | CCCTT TATCCTTCTC
End of individual info.


No ancenstral population recorded.
>>>
```

2.6 Population Structure

subpopulation structure can be accessed through the following member functions:

- `pop.popSize()`, total population size
- `pop.numSubPop()`, number of subpopulations
- `pop.subPopSize(sp)`, size of subpopulation, which can be zero. Note that sp is zero-indexed.
- `pop.subPopBegin(sp)`, the index of the first individual in subpopulation `sp`
- `pop.subPopEnd(sp)`, the index of the last individual in subpopulation `sp` plus one.
- `pop.subPopIndPair(ind)`, return the subpopulation and relative index of individual ind.

- `pop.absIndIndex(ind, sp).  return the absolute index of an individual`

Listing 2.5: population structure functions

```
>>> print pop.popSize()
5
>>> print pop.numSubPop()
2
>>> print pop.subPopSize(0)
2
>>> print pop.subPopSizes()
(2, 3)
>>> print pop.subPopBegin(1)
2
>>> print pop.subPopEnd(1)
5
>>> print pop.subPopIndPair(3)
(1, 1)
>>> print pop.absIndIndex(1,1)
3
>>>
```

There are another set of functions that deal with population/subpopulation size changes. In these functions, the `info` field of each individual plays an important role. This field represents an individual's (new) subpopulation ID most of the times. For example, function `rearrangeByIndID()` rearrange individuals in the order of their info values. Similar functions are

- `pop.setIndSubPopID(info),` set individual info using a vector of size of the population
- `pop.setIndSubPopIDWithID()`, use subpopulation id to set individual info
- `pop.setSubPopByIndID()`, rearrange individual and set subpopulation structure according individual info values
- `pop.removeEmptySubPops()`, remove empty subpopulations.
- `pop.removeSubPops(subPops),` remove subpopulation
- `pop.reorderSubPops(order, rank, removeEmptySubPops=False)`,
- `pop.newPopByIndID(keepAncestralPops=True, info=[]),`
- `pop.removeLoci(remove=[], keep=[]),` remove some loci from the population
- `pop.newPopWithPartialLoci(remove=[], keep=[]),` return a new reduced population
- `pop.splitSubPop(which, subPopSizes, subPopID),` split subpopulation `which` according to subPopSizes
- `pop.spliSubPopByProportion(which, proportions, subPopID),` use proportion
- `pop.mergeSubPop(subPops).` merge subpopulations

These functions may look useful and appealing but you will almost never use them directly. All these operations will be performed by various operators, in a more user-friendly way. Only when you begin to write your own operators will you have to read about the details of these functions.

Example 2.6 demonostrate the use of functions `setIndSubPopID, setSubPopByIndID` and `removeLoci.`

Listing 2.6: population structure functions

```
>>> pop.setIndSubPopID([1,2,2,3,1])
>>> pop.setSubPopByIndID()
>>> pop.removeLoci(keep=range(2,7))
>>> Dump(pop)
Ploidy:                 2
Number of chrom:        2
Number of loci:         3 2
Maximum allele state:   3
Loci positions:
                2 3 4
                0 2
Loci names:
                loc1-3 loc1-4 loc1-5
                loc2-1 loc2-2
population size:        5
Number of subPop:       4
Subpop sizes:           0 2 2 1
Number of ancestral populations:        0
individual info:
sub population 1:
   0: MU CTA AA | GAT TG
   1: MU TTA TG | CTT TA
sub population 2:
   2: FU AAG CT | CAA AC
   3: MU ATC GT | GTA TA
sub population 3:
   4: FU TTC TA | CTA TC
End of individual info.


No ancenstral population recorded.
>>>
```

2.7 Individuals

You can access individuals of a population through `individual()` function. There are two forms of this function, one with and one without parameter `subPop`,

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

The iterator can simplify the access of individuals, by using

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

instead of the older

```
for i in range(pop.popSize()):
  ind = pop.individual(i)
  print ind.affected()
```

The returned `individual` object also has its own member functions. You can retrieve genotypic information of an individual through the same set of functions. You can also get/set genotype of an individual. Note that you can not create an individual object directly.

Listing 2.7: Individual member functions

```
>>> # get an individual
>>> ind = pop.individual(9)
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
IndexError: src/population.h:452 individual index (9) is out of range of 0 ~ 4
>>> # oops, wrong index
>>> ind = pop.individual(3)
>>> # you can access genotypic structure info
>>> print ind.ploidy()
2
>>> print ind.numChrom()
2
>>> # ...
>>> # as well as genotype
>>> print ind.allele(1)
2
>>> ind.setAllele(1,5)
>>> print ind.allele(1)
2
>>> # you can also use an overloaded function
>>> # with a second parameter being the ploidy index
>>> print ind.allele(1,1) # second locus at the second copy of chromosome
2
>>> # other information
>>> print ind.affected()
False
>>> print ind.affectedChar()
U
>>> ind.setAffected(1)
>>> print ind.affectedChar()
A
>>> print ind.sexChar()
M
>>>
```

Again, you will very seldom have to use these functions directly unless when you write pure python operators.

2.8 Reference of class individual

Class `individual`

individuals with genotype, affection status, sex etc.

Details

Individuals are the building blocks of populations, each having the following individual information:

- shared genotypic structure information
- genotype
- sex, affection status, subpopulation ID
- optional information fields

Individual genotypes are arranged by locus, chromosome, ploidy, in that order, and can be accessed from a single index. For example, for a diploid individual with two loci on the first chromosome, one locus on the second, its genotype is arranged as `1-1-1 1-1-2 1-2-1 2-1-1 2-1-2 2-2-1` where `x-y-z` represents ploidy `x` chromosome `y` and locus `z`. An allele `2-1-2` can be accessed by `allele(4)` (by absolute index), `allele(2, 1)` (by index and ploidy) or `allele(1, 1, 0)` (by index, ploidy and chromosome).

Initialization

Individuals are created by populations automatically. Do not call this function directly.

```
individual()
```

Member Functions

`x.affected()` whether or not an individual is affected

`x.affectedChar()` return `A` or `U` for affection status

`x.allele(index)` return the allele at locus `index`

index absolute index from the beginning of the genotype, ranging from `0` to `totNumLoci()*ploidy()`

`x.allele(index, p)` return the allele at locus `index` of the `p-th` copy of the chromosomes

index index from the begining of the `p-th` set of the chromosomes, ranging from `0` to `totNumLoci()`

p index of the ploidy

`x.allele(index, p, ch)` return the allele at locus `index` of the `ch-th` chromosome of the `p-th` chromosome set

ch index of the chromosome in the `p-th` chromosome set

index index from the begining of chromosome `ch` of ploidy `p`, ranging from `0` to `numLoci(ch)`

p index of the polidy

`x.alleleChar(index)` return the name of `allele(index)`

`x.alleleChar(index, p)` return the name of `allele(index, p)`

`x.alleleChar(index, p, ch)` return the name of `allele(idx, p, ch)`

`x.arrGenotype()` return an editable array (a Python list of length `totNumLoci()*ploidy()`) of genotypes of an individual

This function returns the whole genotype. Although this function is not as easy to use as other functions that access alleles, it is the fastest one since you can read/write genotype directly.

`x.arrGenotype(p)` return only the `p-th` copy of the chromosomes

x.arrGenotype(p, ch) return only the ch-th chromosome of the p-th copy

x.arrInfo() return an editable array of all information fields (a Python list of length infosSize())

x.info(idx) get information field idx

idx index of the information field

x.info(name) get information field name

Equivalent to info(infoIdx(name)).

name name of the information field

x.setAffected(affected) set affection status

x.setAllele(allele, index) set the allele at locus index

allele allele to be set

index index from the begining of genotype, ranging from 0 to totNumLoci()*ploidy()

x.setAllele(allele, index, p) set the allele at locus index of the p-th copy of the chromosomes

allele allele to be set

index index from the begining of the poloidy p, ranging from 0 to totNumLoci(p)

p index of the poloidy

x.setAllele(allele, index, p, ch) set the allele at locus index of the ch-th chromosome in the p-th chromosome set

allele allele to be set

ch index of the chromosome in ploidy p

index index from the begining of the chromosome, ranging from 0 to numLoci(ch)

p index of the ploidy

x.setInfo(value, idx) set information field by idx

x.setInfo(value, name) set information field by name

x.setSex(sex) set the sex. You should use setSex(Male) or setSex(Female) instead of 1 and 2.

x.setSubPopID(id) set new subpopulation ID, pop.rearrangeByIndID will move this individual to that population

x.sex() return the sex of an individual, 1 for males and 2 for females. However, this is not guranteed so please use sexChar().

x.sexChar() return the sex of an individual, M or F

x.subPopID() return the ID of the subpopulation to which this individual blongs

Note: subPopID is not set by default. It only corresponds to the subpopulation in which this individual resides after pop::setIndSubPopID is called.

x.unaffected() equals to not affected()

2.9 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. Example 2.9 demonstrates how `stat` set variables `popSize`, `alleleFreq` etc.

You can refer to these variables using `population::vars()` or `population::dvars()` function. The returned value of `vars()` and `dvars()` reflects the same dictionary. However, `dvars()` uses a little Python magic so that you can use attribute syntax to access dictionary keys. Since `a.allaleFreq[0]` is a lot easier to read than `a['alleleFre'][0]`, `dvars()` is always preferred to `vars()`. A function `ListVars` defined in `simuUtil` can be used to display the variables. With wxPython installed, this function will open a nice window with a tree representing the variables. Without wxPython (or use parameter `useWxPython=False`), variables will be displayed in a indented form. Several parameters can be used to limit your display. They are

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

Listing 2.8: Population variables

```
>>> from simuUtil import ListVars
>>> ListVars(pop.vars(), useWxPython=False)
 rep :  -1
 grp :  -1
>>> Stat(pop, popSize=1, alleleFreq=[0])
>>> # subPop is True by default, use name to limit the variables to display
>>> ListVars(pop.vars(), useWxPython=False, subPop=False, name='alleleFreq')
 alleleFreq :
   [0]
     [0]        0.2
     [1]        0.5
     [2]        0.2
     [3]        0.1
>>> # print number of allele 1 at locus 0
>>> print pop.vars()['alleleNum'][0][1]
5
>>> print pop.dvars().alleleNum[0][1]
5
>>> print pop.dvars().alleleFreq[0]
[0.20000000000000001, 0.5, 0.20000000000000001, 0.10000000000000001]
>>>
```

These variables form a Python dictionary, and furthermore a local namespace for functions like `population::evaluate`. *Local namespace* means that you can use dictionary items as variables duing evaluation. For example:

Listing 2.9: Local namespaces of populations

```
>>> print pop.evaluate('alleleNum[0][1] + alleleNum[0][2]')
7
>>> pop.execute('newPopSize=int(popSize*1.5)')
>>> ListVars(pop.vars(), level=1, useWxPython=False)
 newPopSize :   7
 grp :  -1
```

```
 rep :  -1
 popSize :      5
 numSubPop :    4
 alleleNum :
   list of length 1
 subPopSize :
   list of length 4
 alleleFreq :
   list of length 1
 subPop
   list of length 4
>>> # this variable is 'local' to the population and is
>>> # not available in the main namespace
>>> newPopSize
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
NameError: name 'newPopSize' is not defined
>>>
```

As you can see, these variables are *local* to the population and is not directly accessible from the main namespace. `vars(subPop)` and `dvars(subPop)` function can be used. Both functions takes an optional `subPop` option. If ignored, they will return population dictionary; otherwise, they will return dictionary for subpopulation `subPop`. This is a very convenient feature, because subpopulations and populations have similar keys, you can calculate the same statistics for the whole population and individual subpopulations, just by specifying different namespaces.

2.10 Sample from a Population

Sampling (or ascertainment) is a complicated issue. simuPOP provides several methods to generate samples from an existing population. Details please refer to Chapter 7.12.

2.11 Information fields

The information fields are information that is attached to each individual. For example, an individual may need `father_idx` and `mother_idx` to track pedigree information, may need `penetrance` to set affectedness.

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

- `pop.setInfoFields(fields)`
- `pop.addIndField(field)`

Note that changing infoField for a simulator is dangerous since all populations in a simulator share the same genotypic structure. You should addIndField to all populations to avoid potential problems.

One can set/retrieve information at the level of individuals

- `ind.info(idx or field)`
- `ind.setInfo(idx or field)`
- `ind.arrInfo()`

or set the population level

- `pop.indInfo(idx or field, order)`
- `pop.indInfo(idx or field, subPop, order)`
- `pop.setIndInfo(idx or field, [subPop])`
- `pop.arrIndInfo(order)`
- `pop.arrIndInfo(subPop, order)`

Idx or field means that you can use field index obtained from infoIdx(field), or use field name directly. field is easier to use but idx is faster. Although population information is kept in a population object linearly, there is no gurantee that they are ordered. If you would like to access info individual by individual, passing `order=True` will ensure that the returned information fields are ordered by individual order. If you only need to get a summary of some information fields, passing `order=False` will speed up the process.

For each individual, `ind.arrInfo()` will return `f1, f2, f3, ...` etc for that individual. From a population point of view, `pop.arrIndInfo([subPop])` will return a list of `f1, f2, f3, ..., f1, f2, f3, ...`. Note that the order of individuals may not be kept in this (sub)population-wise array. That is to say, `pop.arrIndInfo()[0]` does not have to be the first field of the first individual. This property is also true for `setIndInfo(values, idx or name).` That is to say, if you want to set information field for individuals in a population unordered, you can use

```
setIndInfo(values, idx)
```

Otherwise, you will have to use the less efficient way:

```
for i in range(pop.popSize()):
    pop.individual(i).setInfo(values[i], idx)
```

Note that `indInfo` is more convenient but it is less efficient (fields must be copied out) than `arrIndInfo.` To handle the return value of `arrIndInfo,` you would usually do:

```
idx = pop.infoIdx('trait2')
step = pop.infoSize()
arr = pop.arrIndInfo(subPop=2)
for i in range(pop.subPopSize(2)):
  # note again that arr is writable.
  arr[idx + step*i] = something
```

2.12 Ancestral populations

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

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

Several important usage of ancestral populations:

- `dumper()` operator and `Dump()` function has a parameter `ancestralPops`. If set to true, they will dump all ancestral generations.

- function `population::setAncestralDepth()` and operator `setAncestralDepth()` set the number of ancestral generations to keep for a population. A typical use of `setAncestralDepth()` is

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

  which save the last three generations in populations so that pedigree based sampling schemes can sample from the population.

- `pop.useAncestralPop(idx)` set current generation of population `pop` to `idx` generation. `idx` = 1 for the first ancestral generation, 2 for second ancestral ..., and 0 for current generation. After this function, all functions, operators will be applied to this ancestral population. You should always call `setAncestralPop(0)` after you examed the ancestral populations.

A typical use of this function is demonstrated in example 2.10. In this example, a population with two loci is created and with intial genotype 0. Two kamMutator with different mutation rates are applied to these two loci. Five most recent populations are kept. The allele frequencies at these generations are calculated afterwards. (Note that this is not the best way to exam the changes of allele frequencies, a `stat` operator should be used.)

Listing 2.10: Ancestral populations

```
>>> simu = simulator(population(10000, loci=[2]), randomMating())
>>> simu.evolve(
...   ops = [
...     setAncestralDepth(5, at=[-5]),
...     kamMutator(rate=0.01, atLoci=[0], maxAllele=1),
...     kamMutator(rate=0.001, atLoci=[1], maxAllele=1)
...   ],
...   end = 20
... )
True
>>> pop = simu.population(0)
>>> # start from current generation
>>> for i in range(pop.ancestralDepth()+1):
...   pop.useAncestralPop(i)
...   Stat(pop, alleleFreq=[0,1])
...   print '%d␣␣␣%5f␣␣␣%5f' % (i, pop.dvars().alleleFreq[0][1], pop.dvars().alleleFreq[
...
0   0.168100   0.019450
1   0.161450   0.018550
2   0.154900   0.017000
3   0.147350   0.018050
4   0.140550   0.019600
5   0.131500   0.018600
>>> # restore to the current generation
>>> pop.useAncestralPop(0)
>>>
```

2.13 Save and Load a Population

Internally, population can be saved/loaded in "`txt`", "`xml`" or "`bin`" formats using `savePopulation(file, format, compress=True)` member function, global `SavePopulation(pop, file, format)` and `LoadPopulation`. (Yes, it is `Load..` not `load..` since `savePopulation` is a member function and `LoadPopulation` is a global function.) These formats have their own advantages and disadvantages:

- `xml:` most human readable, easy transformation to other formats, largest file size
- `bin:` not readable, small file size. May not be portable.
- `txt:` human readable with no structure, portable, median file size.

Listing 2.11: Save and load population

```
>>> # save it in various formats, default format is "txt"
>>> pop = population(1000, loci=[2, 5, 10])
>>> pop.savePopulation("pop.txt")
>>> pop.savePopulation("pop.txt", compress=False)
>>> pop.savePopulation("pop.xml", format="xml")
>>> pop.savePopulation("pop.bin", format="bin")
>>>
>>> # load it in another population
>>> pop1 = LoadPopulation("pop.xml", format="xml")
>>>
```

Populations are by default compressed in gzip format. If you are interested in viewing the content of the file, you can use `compress=False` when saving a populaton, or decompress the saved files using `gzip -d` command.

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

It is also possible to save a bunch of populations in a single file, provided that they have the same genotypic structure. The functions are

- `SavePopulations([pop1,pop2,...,], filename, format='auto', compress=True)`
- `LoadPopulations(filename)`

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

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

2.14 View a population (GUI, wxPython required)

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

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

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

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

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

CHAPTER

THREE

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.
- Randomly choose parent(s) to generate offsprings to fill the next generation.
- During-mating operators are applied to all offsprings.
- Apply selection if applicable.

3.1 Create a Mating Scheme

Most mating schemes take the following parameters:

- `numOffsprings` number of offsprings or p for a random distribution. default to 1. This parameter determines number of offsprings a mating event will produce so it determines family size.
- `numOffspringsFunc` a python function that return number of offspring or p.
- `maxNumOffsprings` used when numOffsprings is generated from a binomial distribution.
- `mode` One of `MATE_NumOffspring, MATE_NumOffspringsEachFamily, MATE_GeometricDistribution, MATE_PoissonDistribution, MATE_BinomialDistribution`.
- `newSubPopSize` an array of sizes of subpopulaitons.
- `newSubPopSizeExpr` an expression that will return the new subpopulation size. Details about python expression will be discussed later.
- `newSubPopSizeFunc` Added for more convenience. This should be a function that accept a int parameter (generation), an array of current population size and return an array of subpopulation sizes. This is usually easier to use than the expression version of this parameter.

3.2 Determine number of offsprings during mating

The default values `numOffsprings` parameters makes a mating scheme produce one offspring per mating. This is the real random mating and should be used whenever possible. However, various situations requires larger family size or even changing family size. simuPOP provides a comprehensive way to deal with this problem.

The method to determine the number of offsprings are set by `mode` parameter:

- `MATE_NumOffsprings:` if `numOffspringsFunc` is not given, number of offsprings will be constant `numOffsprings` all the time. Otherwise, `numOffspringsFunc(gen)` will be called **once** for each generation to get the number of offsprings for the matings happen in this generation.
- `MATE_NumOffspringsEachFamily:` `numOffspringsFunc` has to be given and will be called whenever a mating happens. Since `numOffspringsFunc` can be **any** python function, this mode allows arbitrary model of assigning number of offsprings during mating. The mode can be slow though.
- `MATE_GeometricDistribution:` `numOffsprings` or result of `numOffspringsFunc` (evaluated at each generation) will be considered as p for a geometric distribution. The number of offsprings for each mating is determined by
$$P(k) = p(1-p)^{k-1} \quad \text{for } k \geq 1$$
- `MATE_PoissonDistribution:` `numOffsprings` or result of `numOffspringsFunc` (evaluated at each generation) will be considered as p for a Poission distribution. The number of offsprings for each mating is determined by
$$P(k) = \frac{p^{k-1}}{(k-1)!}e^{-p} \quad \text{for } k \geq 1$$
Since the mean of this shifted Poisson distribution is $p+1$, you need to specify, for example, 2, if you want a mean family size 3. (FIXME: this part needs more consideration.)
- `MATE_BinomialDistribution:` `numOffsprings` or result of `numOffspringsFunc` (evaluated at each generation) will be considered as p for a Binomial distribution. Let N=`maxNumOffsprings`, the number of offsprings for each mating is determined by
$$P(k) = \frac{(n-1)!}{(k-1)!(n-k)!}p^{k-1}(1-p)^{n-k} \quad \text{for } N \geq k \geq 1$$

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

3.3 Determine subpopulation sizes of next generation

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

- If you would like to have fixed subpopulation sizes, use `newSubPopSize=some_fixed_values` . This is useful when subpopulation sizes are changed by migration and you do want to keep constant subpopulation sizes.
- If subpopulation size can be easily calculated through an expression, you can use `newSubPopSizeExpr` to determine the new subpopulation sizes. For example `newSubPopSizeExpr='[gen+10]'` uses generation number + 10 as the new population size. More complicated expression can be used, maybe along with `pyExec` operators, but in this case, a specialized function and `newSubPopSizeFunc` is recommened. Note that the expression uses variables from local namespace.
- A more organized (and thus recommended) way to set new population/subpopulation sizes is through parameter `newSubPopSizeFunc`. To use this paarmeter, you need to define a Python function that take two parameters: generation number and current subpopulation sizes and return an array of new subpoulation sizes. (return `[newsize]` instead of `newsize` when you do not have any subpopulation structure). For example, the following function defines a linear expansion demographic scenario where a real example where a single population is splitted at 200 generations (using a `spliPopulation` operator).

```
def lin_exp(gen, oldSize=[]):
  if gen < 200:   # burn in, constant population size
    return [1000]
  else:           # increase subpopulation sizes
    incSize = (10000-1000)/(500-200)/len(oldSize)
    return [oldSize[x]+incSize for x in range(0, len(oldSize))]
```

you can then use this function as follows

```
...randomMating(newSubPopSizeFunc=lin_exp) ...
```

3.4 Demographic change functions

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

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

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

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

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

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

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

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

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

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

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

3.5 Different Mating Schemes

Currently, simuPOP provides the following mating schemes:

- `noMating()` parent generation will be considered as offspring genertion. subpopsizes will be ignored although some during-mating operators can be applied.
- *`binomialSelection`*`()` no sex is involved. Offspring is chosen from parental generation by random or according to fitness values.
- `radomMating()` sexed random mating. A prameter `(contIfUniSex)` can be set to determine the behavior when only one sex exists in a subpopulation. Default is continue without warning.
- `pyMating()` (**not usable right now)** Hybird mating scheme. This mating scheme takes two parameters: `mateFunc` and `keepSubPopStru`. `mateFunc` should be a python function that accept a (parental) population and return parent indices for each offspring. If `keepSubPopStru=True` (default), parents should come from the same subpopulation and the offspring population will have subpopulation strcture. Otherwise, mating can across subpopulation strcuture.

Detailed information of each mating scheme can be found through `help(...)` function.

3.6 Sex chromosomes

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

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

CHAPTER

FOUR

Operators

Operators are objects that act on populations. They (there are exceptions) can be applied to populations directly using `apply`() member function, but most of the time they are managed and applied by a simulator.

4.1 Class reference

Class `Operator`

base class of all classes that manipulate populations

Details

Operators are objects that act on populations. They can be applied to populations directly using their function forms, but they are usually managed and applied by a simulator.

Operators can be applied at different stages of the life cycle of a generation. More specifically, they can be applied at *pre-*, *during-*, *post-mating*, or a combination of these stages. Applicable stages are usually set by default but you can change it by setting `stage=(PreMating|PostMating|DuringMating|PrePostMating)` parameter. Some operators ignore `stage` parameter because they only work at one stage.

Operators do not have to be applied at all generations. You can specify starting/ending generation, gaps between applicable generations, or even specific generations. For example, you might want to start applying migrations after certain burn-in generations, or calculate certain statistics only sparsely.

Operators can have outputs. Output can be standard (terminal) or a file, which can vary with replicates and/or generations. Outputs from different operators can be accumulated to the same file to form table-like outputs.

Operators are applied to every replicate of a simulator by default. However, you can apply operators to one or a group of replicates using parameter `rep` or `grp`.

Filenames can have the following format:

- `'filename'` this file will be overwritten each time. If two operators output to the same file, only the last one will succeed;
- `'>filename'` the same as `'filename'`;
- `'>>filename'` the file will be created at the beginning of evolution (`simulator::evolve`) and closed at the end. Output from several operators is allowed;
- `'>>>filename'` the same as `'>>filename'` except that the file will not be cleared at the beginning of evolution if it is not empty;

- `'>'` standard output (terminal);
- `''` supress output.

Initialization

common interface for all operators (this base operator does nothing by itself.)

```
Operator(output, outputExpr, stage, begin, end, step, at, rep, grp,
infoFields)
```

at an array of active generations. If given, `stage`, `begin`, `end`, and `step` will be ignored.

begin the starting generation. Default to `0`. Negative numbers are allowed.

end stop applying after this generation. Negative numbers are allowed.

grp applicable group. Default to `GRP_ALL`. A group number for each replicate is set by `simulator.__init__` or `simulator::setGroup()`.

output a string of the output filename. Different operators will have different default `output` (most commonly `'>'` or `''`).

outputExpr an expression that determines the output filename dynamically. This expression will be evaluated against a population's local namespace each time when an output filename is required. For example, `"'>>out%s_%s.xml' % (gen, rep)"` will output to `>>>out1_1.xml` for replicate `1` at generation `1`.

rep applicable replicates. It can be a valid replicate number, `REP_ALL` (all replicates, default), or `REP_LAST` (only the last replicate). `REP_LAST` is useful in adding newlines to a table output.

step the number of generations between active generations. Default to `1`.

Note

Negative generation numbers are allowed for `begin`, `end` and `at`. They are intepretted as `endGen + gen + 1`. For example, `begin = -2` in `simu.evolve(..., end=20)` starts at generation `19`.

Member Functions

`x.MPIReady()` determine if this operator can be used in a MPI module

`x.applicableGroup()` return applicable group

`x.applicableReplicate()` return applicable replicate

`x.apply(pop)` apply to one population. It does not check if the operator is activated.

`x.canApplyDuringMating()` set if this operator can be applied *during-mating*

`x.canApplyPostMating()` set if this operator can be applied *post-mating*

`x.canApplyPreMating()` set if this operator can be applied *pre-mating*

`x.canApplyPreOrPostMating()` set of this operator can be applied *pre-* or *post-mating*

`x.clone()` deep copy of an operator

`x.diploidOnly()` determine if the operator can be applied only for diploid population

x.haploidOnly() determine if the operator can be applied only for haploid population

x.infoField(idx) get the information field specified by user (or by default)

x.infoSize() get the length of information fields for this operator

x.setActiveGenerations(begin=0, end=-1, step=1, at=[]) set applicable generation parameters: `begin`, `end`, `step` and `at`

x.setApplicableGroup(grp=GRP_ALL) set applicable group

Default to `GRP_ALL` (applicable to all groups). Otherwise, the operator is applicable to only *one* group of replicates. Groups can be set in `simulator::setGroup()`.

x.setApplicableReplicate(rep) set applicable replicate

x.setApplicableStage(stage) set applicable stage. Another way to set `stage` parameter.

x.setOutput(output="", outputExpr="") set ouput stream, if was not set during construction

4.2 Type of operators

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 during simulation. 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. Note that although it is possible to write pure python operators to operate directly on populations, it might work very slowly compared to the built-in ones.

4.2.1 Applicable Stages

Operators can be applied at different stage(s) of a life cycle. More specifically, at pre-, during- or post mating stage(s). Note that it is possible for an operator to apply multiple times in a life cycle. For example, an save-to-file operator might be applied before and after mating to trace parental information. Applicable stages are usually set by default but you can change it by setting `stage=(PreMating|PostMating|DuringMating|PrePostMating)` parameter. Note that some operators ignore `stage` parameter since they only work at one stage.

Listing 4.1: Operator stage

```
>>> d = dumper()
>>> print d.canApplyPreMating()
False
>>> print d.canApplyDuringMating()
False
>>> # so dumper is a post mating operator
>>> print d.canApplyPostMating()
True
>>>
```

4.2.2 Active Generations

Operators do not have to be applied at all generations. You can specify starting generation, ending generation, gaps between applicable generations, or even specific generations to apply. For example, you might want to start applying migrations after certain burn-in generation; or you want to calculate every 10 generations. Operators take the following parameters during initialization:

- `begin` start generation. default to 1. negative number is interpreted as endGeneration + begin
- `end` stop applying after this generation. negative number is allowed
- `step` number of generations between active generations. default to 1
- `at` an array of active generations. If given, begin, end, step will be ignored.

For example

Listing 4.2: Set active generations of an operator

```
>>> simu = simulator(population(1),binomialSelection(), rep=3)
>>> op1 = output("a", begin=5, end=20, step=3)
>>> op2 = output("a", begin=-5, end=-1, step=2)
>>> op3 = output("a", at=[2,5,10])
>>> op4 = output("a", at=[-10,-5,-1])
>>> simu.evolve( [ pyEval(r"str(gen)+'\n'", begin=5, end=-1, step=2)],
...               end=10)
5
5
5
7
7
7
9
9
9
True
>>>
```

The last example displays variable `gen` for each replicate. Note that you can use negative generation number whenever you specifies the `end` parameter of evolve. In this case, generation -1 is the last generation (end), -2 is end-1, and so on.

4.2.3 Replicates and Groups

Most operators are applied to every replicate of a simulator during evolution. However, you can apply operators to one or a group of replicates only. For example, you can initialize different replicates with different initial values and then start evolution. c.f. `simulator::setGroup` .

The most useful example is

```
output('\n',rep=REP_LAST)
```

that will output `\n` at the end of each generation. Here is an example of using replicate groups:

Listing 4.3: Replicate group

```
>>> from simuUtil import *
```

```
>>> simu = simulator(population(1),binomialSelection(), rep=4,
...                   grp=[1,2,1,2])
>>> simu.apply([ pyEval(r"grp+3", grp=1),
...              pyEval(r"grp+6", grp=2),
...              output('\n', rep=REP_LAST)]
... )
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'simulator' object has no attribute 'apply'
>>>
```

4.2.4 Output Specification

Operators can have outputs. Output can be standard output (terminal) or a file, which can be constant, or change with generation or replicate. Different operators can append to the same file to form table-like outputs.

Filename can have the following format:

- `'filename'` this file will be closed after each use. I.e., if several operators output to the same file, only the last one will succeed.
- `'>filename'` the same as 'filename'
- `'>>filename'` The file will be created at the beginning of evolution (`simulator::evolve`) and close at the end. Several operators can output to this file to form a table.
- `'>>>filename'` The same as `'>>filename'` except that the file will not be cleared at the beginning of evolution if it is not empty.
- `'>'` out put to standard output.
- `''` suppress output.

The following example shows the difference between "`>`" and "`>>`"

Listing 4.4: log/operatoroutput

```
>>> simu = simulator(population(100), randomMating(), rep=2)
>>> simu.step(
...   preOps=[
...     initByFreq([0.2, 0.8], rep=0),
...     initByFreq([0.5, 0.5], rep=1) ],
...   ops = [
...     stat(alleleFreq=[0]),
...     pyEval('alleleFreq[0][0]', output='a.txt')
...   ]
... )
True
>>> # only from rep 1
>>> print open('a.txt').read()
0.45
>>>
>>> simu.step(
...   ops = [
...     stat(alleleFreq=[0]),
```

```
...     pyEval('alleleFreq[0][0]', output='>>a.txt')
...   ])
True
>>> # from both rep0 and rep1
>>> print open("a.txt").read()
0.210.505
>>>
>>> outfile='>>>a.txt'
>>> simu.step(
...   ops = [
...     stat(alleleFreq=[0]),
...     pyEval('alleleFreq[0][0]', output=outfile),
...     output("\t", output=outfile),
...     output("\n", output=outfile, rep=0)
...   ],
... )
True
>>> print open("a.txt").read()
0.210.5050.18
0.565
>>>
```

In the first simulator, all operators uses `"a.txt"` (the same as `">a.txt"`). This file is repeatedly covered by other operators so what we finally get is a newline written by `output("\n")`. The second simulator works fine by using `">>a.txt"`.

Output filename does not have to be fixed. If `outputExpr` parameter is used (`output` will be ignored), it will be evaluated when a filename is needed. This is useful when you need to write to different files for different replicate/-generations.

Listing 4.5: log/operatoroutputexpr

```
>>> outfile="'>>a'+str(rep)+'.txt'"
>>> simu.step(
...   ops = [
...     stat(alleleFreq=[0]),
...     pyEval('alleleFreq[0][0]', outputExpr=outfile)
...   ]
... )
True
>>> print open("a0.txt").read()
0.19
>>> print open("a1.txt").read()
0.555
>>>
```

4.3 Python expression and statistics calculation

4.3.1 Expressions and Statements

Expressions are used extensively in operators so basic knowldege of python is required. If you know almost nothing about Python, please spend some time on the Python tutorial from python website.

Unlike C/C++, assignments in Python do not return values. This is the biggest difference between Python expression and statement:

- expressions consist of constants, variables, operators, functions, but *no* assignments, condition, loop etc. Expression returns a value when executed. An example of expression is `range(1,5)+10` .
- statements consist of arbitrary valid python code. Statement does *not* return a value when executed. An example of statement is `a=range(1,5)` .

4.3.2 simuPOP variables

All populations have their own attached variables. We have seen the structure of a population dictionary: it starts empty and will have many variables created by various operators. You can access local namespace of each replicate through a simulator's `vars(rep)` function:

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

It is important to know that

- `simulator::vars(0), vars(1)` etc are the *local namespaces* for each replicate.
- `subPop[0], subPop[1]` etc have almost the same set of keys as those for the whole population. This is because `stat` operator calculate statistics of each replicate of population, and all subpopulations.

To list these variables, you can use the `ListVars()` function defined in `simuUtil.py`. For example

```
ListVars(simu.vars(0), level=2)
```

list all variables for the first replicate. `Level=2` stops `ListVars` from expanding lists and dictionaries after two levels.

Two functions can be used to access simulator and population variables: `vars()` and `dvars()`. We have known `population::vars()` and `population::dvars()`, `simulator::vars()` and `simulator::dvars()` work in almost the same way.

- `simulator::vars(rep)`, `dvars(rep)`: return replicate `rep`'s local namespace
- `simulator::vars(rep, subPop)`, `dvars(rep, subPop)`: return the namespace of `subPop` subpopulation of replicate `rep`.

The return values of `vars()` and `dvars()` are different. `vars()` returns a Python dictionary. You should access their keys in the usual Python way. `dvars()` returns a 'wrapped' Python dictionary. You can access dictionary keys as attributes. `dvars()` is usually considered to be easier to use.

4.3.3 `evaluate` function and `pyEval` and `pyExec` operators

Function `population::evaluate` and operator `pyEval/pyExec` will work in local namespaces. For example, if there are `a` and `b` in the main namespace and `a` in `pop`, `pop.evaluate('a')` will return `pop.vars()['a']`, `pop.evaluate('b')` will return global `b` since there is no `b` in the local namespace. It this is still too abstract, here is a real example

Listing 4.6: python expression

```
>>> simu = simulator(population(10),noMating(), rep=2)
>>> # evaluate an expression in different areas
>>> print simu.vars(0)
{'rep': 0, 'gen': 0, 'grp': 0}
>>> print simu.population(0).evaluate("grp*2")
0
>>> print simu.population(1).evaluate("grp*2")
2
>>> print simu.population(0).evaluate("gen+1")
1
>>> # a statement (no return value)
>>> simu.population(0).execute("myRep=2+rep*rep")
>>> simu.population(1).execute("myRep=2*rep")
>>> print simu.vars(0)
{'rep': 0, 'myRep': 2, 'gen': 0, 'grp': 0}
>>>
```

- `simulator` creates a simulator with two replicates 0 and 1.
- We evaluate `grp*2` in different replicates and get different results.
- `gen` is not in either replicate's namespace so the global one will be used.
- Using statements can create variables in local namespaces. (You can use `global` statement to create global variable if you are familiar with python.)

`pyEval/pyExec` operators execute python expression/statements, *using local namespaces*.

- `pyEval` (operator) evaluate a Python expression and return its value, optional execute a list of statements beforehand.
- `pyExec` (operator) execute a list of statements in the form of a multi-line string. No return value or output.

Here, `expr` is a simple string containing an expression that will return a value when executed; `stmts` is a string of statements, separated by `'\n'`.

For example, you can return a string of "gen:rep" using the following function

```
pop.evaluate(r'%d:%d' % (gen,rep))
```

but if you would like to change/create variables, you have to use statements like

```
pop.evalulate(rmyval, stmts=rmyval=rep+1)
```

Since you are executing Python statements, you can of course do it directly in python. For example, the above function does exactly the following

```
pop.vars()['myval'] = pop.vars()['rep'] + 1
pop.vars()['myvar']
```

As a matter of fact, we seldom use `evaluate` function directly (maybe for debuging), usually

- we use expressions for dynamic parameters. For example:

  ```
  newSubPopSizeExpr=range(10,20)*1.2
  outputExpr= '␣saveAt%s.txt␣%␣gen'
  ```

  These parameters will be evaluated whenever they are referred.

- we use expression/statements in `pyEval/pyExec` operators. These statements will work in local namespaces. For example:

Listing 4.7: Expression evaluation

```
>>> simu.step([ pyExec("myRep=2+rep*rep") ])
True
>>> print simu.vars(0)
{'rep': 0, 'selection': False, 'myRep': 2, 'gen': 0, 'grp': 0}
>>>
```

Because of the interactive nature of python, it is very easy to write short programs, quote them in `r"'program"'` and put them in to `pyEval/pyExec` operators.

CHAPTER

FIVE

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.

5.1 Generation Number

Several aspects of generation number may cause confusion:

- generation starts from zero
- a generation number presents a 'to-be-evolved' generation
- 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 three times. Generation 0, generation 1, and generation 2. At the end of 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).

When you use `start=0, step=5, end=10` for your operator, it will be applied at generations `0,5,10` etc.

5.2 Operator calling sequence

Simulators separate operators into `pre-`, `during-` and `post-` mating operators. During evolution, simulator first apply all `PreMating` operators and then call the `mate()` function of the given mating scheme, which will call

`DuringMating` operators during the birth of each offspring. After the new generation is generated, `PostMating` operators are applied in the order they appear in the operator list.

Anyway, operators are not always active. They can be applied to certain generations or certain replicate(s) of population. A simulator will always apply preOps and postOps operators, but will ask if an operator is active (by providing `rep, grp, gen` information) before its is called.

The order of applying operators usually does not matter but errors can occur if you are not careful. For example, `stat(...)` calculate the statistics of 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 post-mating population. However, it 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 real evolution is always a good idea.

5.3 Evolution

Simulators can evolve a given number of generations (the `'end'` parameter of evolve), or evolve indefinitely using a certain type of operators called terminators. In this case, one or more terminators will check the status of evolution and determine if the simulation should be stopped. An obvious example of such a terminator is a fixation-checker. Useful simulator functions are

- `gen()` return current generation number
- `setGen()` set current generation. Usually used to reset a simulator
- `population()` return temporary reference of one of the populations. 'Reference' means that the changes to the referred population will reflect to the one in simulator. 'Temporary' means that the referred population might be invalid after evolution.
- `evolve()` evolve all replicates of the population
- `apply()` apply a list of operators to all populations.
- `step()` evolve one generation.

The most useful function is of course `evolve`, which takes parameters

- `preOps:` operators that will be applied before evolution
- `ops:` opeartors that will be applied at each generation.
- `postOps:` operators that will be applied after evolution.
- `end:` ending generation. Default to -1. In this case, a simulator will only be ended by a terminator.
- `dryrun:` dryrun mode. see previous section
- `saveAs:` saveAt, format: see next section

5.4 Save and Load

A simulator can be saved to a file in the format of `'txt', 'bin'`, or `'xml'`. This enables us to stop a simulation and resume it at another time or on another machine. It is also a good idea to save a snapshot of a simulation every several generations. Note that mating scheme can not be saved and has to be re-specified in `LoadSimulator()`.

Listing 5.1: save and load a simulator

```
>>> simu.saveSimulator("s.txt")
>>> simu.saveSimulator("s.xml", format="xml")
>>> simu.saveSimulator("s.bin", format="bin")
>>> simu1 = LoadSimulator("s.txt", randomMating())
>>> simu2 = LoadSimulator("s.xml", randomMating(), format="xml")
>>> simu3 = LoadSimulator("s.bin", randomMating(), format="bin")
>>>
```

simulators can also be saved during evolution. Three relevant parameters of `evolve()` function are:

- `saveAs:` filename to save the simulator. Default to `simu`.
- `saveAt:` generations at which to save the simulator. Generation can be negative, meaning counting backwards.
- `format:` format. Default to 'bin'.

During evolution, simulator will be saved at `saveAt` generations with filenames `saveAs+gen+format` (for example `simu1000.bin`).

It is also possible to build a simulator from a bunch of populations:

- `SimulatorFromPops(pops, mating)`, build a simulator with given populations and mating scheme
- `SimulatorFromFiles(files, mating)`, load populations from a given list of files (population images) and build a simulator with given mating scheme.

CHAPTER

SIX

Option handling

6.1 Conventions of simuPOP scripts

A simuPOP script is usually composed of the following parts:

1. First line:

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

2. Introduction to the whole script:

   ```
   '''
   This script simulates ....
   '''
   ```

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

3. Options: (see the next section)

   ```
   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. Auxillary functions

5. 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.

Note that these scripts, if proper written, can also be imported. Other scripts (or interactive session) can import a script and call its simulation function directly.

6.2 Parameter handling and user input

Although simuPOP scripts, simply Python scripts, can be in any valid Python style, it is highly recommended that all simuPOP scripts follow the same writing style and provide a uniform interface to users. From a user's point of view, a simuPOP script `cmd.py` should

1. Start a Tk/wxPython dialog to accept user input when `--noDialog` is not specified.
2. List all commandline/config file options through `-h` or `--help` option.
3. Accept `-c` or `--config` parameter to read a configuration file and set parameters.
4. Be able to use command line arguments to set parameters if `--useDefault` is not specified.
5. When `--noDialog` and `--useDefault` is specified, use default values for all parameters, if they can not be obtained from commandline parameters, configuration file, and have a default value.
6. Accept `--saveconfig file` to save current configuration (input my commandline argument) into `file`.
7. Be able to make use of optimized libraries through the use of command line parameter (`--optimized`), config file entry (`optimized=True`) or environment variable (`SIMUOPTIMIZED`).
8. Be able to make use of longallele libraries through the use of command line parameter (`--longallele`), config file entry (`longallele=True`) or environment variable (`SIMULONGALLELE`).

To alleviate throuble of doing all these, `simuPOP` has provided a set of functions. Here is how parameters should be handled. The first step is decribe each parameter in details. This includes (not all is necessary) short and long argument name, entry on a configuration file, prompt when asking for user input, default value, description that will be shown in usage, allowed types of parameter, function to validate the input value. All these should be put in a list of dictionaries like follows:

```
options = [
  { 'arg':'h', 'longarg':'help', 'default':False,
    'allowedTypes':[IntType],
    'description':'print␣this␣message'},
  { 'longarg':'saveconfig=', 'default':'', 'allowedTypes':[StringType],
    'description':'Save␣current␣configuration␣in␣a␣file.'},
  { 'arg':'m', 'longarg':'mu', 'label'='mutationRate',
```

```
    'default':0.005,
    'validate': simuOpt.valueBetween(0,1),
    'description':'mutation rate (a number or an array of numbers) at each loci'
  } ]
```

The entries:

- `arg` and `longarg` are command line argument format. For example,
  - `arg:'h'` checks the presence of argument `-h`, return `True` if succeeds
  - `arg:'f:'` checks the presence of argument pair `-f something`, return `something` if succeeds
  - `longarg:'help'` cheks the prescence of argument `--longarg`, return `True` if succeeds
  - `longarg:'mu='` checks the presence of argument pair `--mu number`, return `number` if succeeds.
- `label` will be used as the label of input field in a parameter dialog, and as the prompt for user input.
- `default` is used when prompt is empty, or when user press enter directly.
- `useDefault` use default value without asking, if the value can not be determined from GUI, command line option or config file. This is useful for options that rarely need to be changed. Setting them to `useDfault` allows shorter command lines, and easier user input.
- `description` is the description of this parameter, will be put into usage information. (`-h` or help button in parameter dialog).
- `allowedTypes` is the accepted types. If `allowedTypes` is `types.ListType` or `types.TupleType` and user input is a scalar, the input will be converted to a list automatically.
- `validate` is a function to validate the parameter. You can define your own functions or use the following from `simuOpt`
  - `valueGT(a), valueLT(a), valueGE(a), valueLE(a)`: check greater than, less than, greater equal, less equal to a value `a.`
  - `valueBetween(a,b), valueOneOf(list):` check if the value is between `a` and `b` or is one of `list`
  - `valueValidFile(), valueValidDir():` check if the parameter is a valid file/directory name.
  - `valueIsNum():` check if the parameter is a number.
  - `valueListOf():` check if parameter is a list of given type, in a list of types, or pass a validator. For example, you can use `valueListOf(types.IntType)`, `valueListOf([types.IntType, types.LongType])` or `valueListOf( valueValidFile()).` As you can see, validators can be nested.
  - `valueOr(validator), valueAnd(val1, val2), valueOr(val1,val2)` accepts other validators and perform respective logical calculation. For example

    ```
    valueOr( valueGT(0), valueListOf( valueGT(0) ))
    ```

    accept a positive number, or a list of positive number.
- `chooseOneOf:` If specified, `simuOpt` will choose one from a list of values using a listbox (tk) or a combo box (wxPython) .
- `chooseFrom:` If specified, `simuOpt` will choose one or more items from a list of values using a listbox (tk) or a combo box (wxPython) .
- `separator:` if specified, a blue label will be used to separate groups of parameters.

- `jump:` jump is used to skip some parameter when doing interative user input. For example, `getParam` will skip the rest of the parameters if `-h` is specified since parameter `-h` has item `'jump':-1` which means jump to the end. Another use of this value is when you have a hierarchical parameter sets. For example, if mutation is on, specify mutation rate, otherwise proceed....
- `jumpIfFalse:` The same as `jump` but jump if current parameter is false.

With all these information at hand, the rest is routine, if you follow the coding conventions.

CHAPTER

SEVEN

Operator and Function References

This chapter will list all functions, types and operators by category.

7.1 Library-dependent functions/constants

Several functions and constants are defined for each library

- `alleleType()`: return `'binary'`, `'short'`, or `'long'`.
- `MaxAllele`: 1 for binary libraries, usually 255 for short libraries and $2^{32}-1$ for long libraries. Note that this number for short and long libraries might change on different platforms.
- `simuVer()`: return version string
- `simuRev()`: simuPOP revision number. If your script needs a recent version of simuPOP, it is a good idea to test `simuRev()` against the revision when the feature you need became available.

7.2 `carray` type

The return value of simuPOP functions that start with `arr` is of a special python type `carray`. This object reflects the underlying C/C++ array and you can read/write array element just as a regular list. Only a small subset of list member functions, `count, index` to be exact, are available. This is because you are not allowed to change the size of underlying C/C++ vector. The following is the operations allowed:

```
# obtain an object using one of the arrXXX functions
pop = population(loci=[3,4], lociPos=[1,2,3,4,5,6,7])
arr = pop.arrLociPos()
# print and expression (just like list)
print arr
str(arr)
# count
arr.count(2)
# index
arr.index, 2)
# can read write
arr[0] = 0.5
# convert to list
arr.tolist()
# or simply
```

```
list(arr)
# compare to list directly
arr == [0.5, 1.0, 3.0, 3.5, 5.0, 6.0, 7.0]
# you can also convert and compare
list(arr) == [0.5, 1.0, 3.0, 3.5, 5.0, 6.0, 7.0])
# slice
arr[:] = [1,2,3,4,5,6,7]
# arr1 is 1,2,3
arr1 = arr[:3]
# assign slice from a number
# IMPORTANT NOTE that arr will also be affected
# since arr1 point to a part of arr
arr1[:] = 10
# assign vector of the same length
arr1[:] = [30,40]
# assign from another part
arr[1:3] = arr[3:5]
```

No other operation is allowed.

Important note: Objects returned from `arrXXX` functions should be considered temporary. There is no gurantee the underlying array will still be valid after any population operation.

7.3 Use of R (RPy) in Python

Most of the info can be found in rpy manual. One function in simuRPy may help though:

```
def rmatrix(mat):
  '␣convert␣a␣python␣2d␣list␣to␣r.matrix␣object'
  return with_mode(NO_CONVERSION, r.do_call)('rbind',mat)
```

with this function, you can easily handle matrices in R. (List and array has been easy enough to be handled).

```
>>> a = [[1,2],[4,5]]
>>> r.image( rmatrix(a))
```

With the help of this function, you can call almost any R function directly, maybe except some R-only syntax like formula, expression etc. In this case, you can always do

```
>>> r('''whatever R expression''')
```

Since Rpy is not always available, you may see the following scenario again and again in simuPOP scripts:

```
try:
  from simuRPy import *
except:
  hasRPy = False
else:
  hasRPy = True
...
if hasRPy:
  r.....
```

7.4 Operator (Hybrid) pyOperator, pyIndOperator

This is the single most powerful hybrid operator. Whenever you think that something is too complicated to be done by standard operators, you can do it here in python. This operator accepts a Python function which accepts a population and optionally a parameter. To use this operator, you will need to

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

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

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

- use `pyOperator` like

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

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

When `pyOperator` is called, it will simply pass the accepted population to the function. If your function returns `False`, the simulation will be stopped.

This operator allows implementation of arbitrarily complicated operators, at a cost of efficiency. Of course, 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 frequency.

Listing 7.1: define a python operator

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

Listing 7.2: use of python operator

```
>>> pop = population(size=10000, ploidy=2, loci=[2, 3])
>>>
>>> simu = simulator(pop, randomMating())
>>>
>>> simu.evolve(
...   preOps = [
...     initByFreq( [.6, .4], atLoci=[0,2,4]),
...     initByFreq( [.8, .2], atLoci=[1,3]) ],
...   ops = [
```

```
...     pyOperator( func=dynaMutator, param=(.5, .1, 0) ),
...     stat(alleleFreq=range(5)),
...     pyEval(r'"%f\t%f\n"%(alleleFreq[0][1],alleleFreq[1][1])', step=10)
...     ],
...   end = 30
... )
0.397000        0.203100
0.382950        0.202300
0.365150        0.208750
0.373650        0.188500
True
>>>
```

Note that

- Currently, `pyOperator` does not support parameter `output` and `outputExpr`. This is because of the incompatibility between the Python way and underlying C++ way of handling file I/O stream. Consequently, you will have to handle file input/output by yourself through `param` parameter. Be careful that you **can not** mix output of `pyOperator` with those of other (normal) operators.
- If parameter `param` is ignored, `myOperator` must be without `para` as well. Note that you can pass arbitrary number of parameters by putting them into a tuple and pass to `myOpeartor.`
- Since you can attach any information to a population, you can in practise use `pop.dvars()` to pass parameters.
- `pyOperator` is a post-mating operator by default. Remember to use `stage` parameter to change this when necessary.

`pyOperator` can also be a DuringMating operator, you will need to define a function

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

or

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

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

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

or

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

The two additional parameters are:

- `formOffGenotype:` (default to `False`) By default, a mating scheme will set the genotype of offspring by copy one of the parental chromosomes. However, if `formOffGenotype` is true, the mating scheme will let you do the job. You will have to set offspring genotype and sex by yourself.
- `passOffspringOnly:` In case that your function will only deal with offspring, you can set this parameter to true and use a shorter form of the function.

Note that if your duringMating 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 big.

An example of duringMating pyOperator can be found in `scripts/demoPyOperator.py`.

Another general python operator is `pyIndOperator`, it is similar to `pyOpertor` but it passes the user individuals, rather than the whole population.

```
def func(ind, param):
  ind.setInfo(param[0], 'myinfo)
pyIndOperator(func=func, param=(1,))
```

is the same as

```
def func(pop, param):
  for ind in pop.individuals():
    ind.setInfo(param[0], 'myinfo')
pyIndOperator(func=func, param=(1,))
```

The `pyIndOperator` may have some performance advantage over `pyOperator` in some cases.

7.5 Initialization

Initializers are used to initialize populations before evolution. They are set to be PreMating operators by default. simuPOP provides three initializers, one assigns alleles by random, one assigns a fixed set of genotype, and the last one calls a user-defined function.

7.5.1 Operator (C++) `initByFreq`, function `InitByFreq`

`initByFreq` operator accepts `alleleFreq` or `alleleFreqs`. The first one ignores subpopulation structure while the second one gives different initial allele frequencies to different subpop or ranges. These parameters are

- `subPop`: specifies applicable subpopulations. If `alleleFreqs` are given, `alleleFreqs` should have the same length as subPop. (One freq each subPop)
- `indRange`: range(s) of absolute index of individuals. I.e., one (`[1,2]`) or more (`[[1,4],[5,6]]`) ranges are acceptable.This is how you can initialize individuals differently within subpopulations. Note that ranges are in the form of `[a,b)`. I.e., range `[4,6]` will intialize individual 4, 5, but not 6. As a shortcut for `[4,5]`, you can use `[4]` to specify one individual. (Note that some earlier versions of simuPOP may use [4,6] for 4,5 and 6. I changed this to let the range specification in line with the Python convention.)
- `atLoci`: loci at which initialization will be done.
- `maleFreq`: intialize sex with this male frequency.
- `identicalInds`: if true, copy the genotype of the first randomly initialized individual to other individuals in the subpop/range.

Here is an example of using `alleleFreq`:

Listing 7.3: Init by freq

```
>>> simu = simulator( population(subPop=[2,3], loci=[5,7]),
...     randomMating(), rep=1)
>>> simu.apply([
```

```
...     initByFreq(alleleFreq=[ [.2,.8],[.8,.2]]),
...     dumper(alleleOnly=True)
...   ])
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'simulator' object has no attribute 'apply'
>>>
```

Please refer to `test/test_init.py` for more complicated examples.

7.5.2 Operator (C++) `initByValue`, function `InitByValue`

`initByValue` operator gets the one copy of chromosomes or the whole genotype (or of those corresponds to `atLoci`) of an individual and copy them to all or subset of individuals.

Listing 7.4: Init by value

```
>>> simu.apply([
...     initByValue([1]*5 + [2]*7 + [3]*5 +[4]*7),
...     dumper(alleleOnly=True)])
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'simulator' object has no attribute 'apply'
>>>
```

Parameters `subPop, indRange, atLoci, maleFreq` are also supported. Note that

- If `value` is an array of values, it should have the same length as `subpop, indRange` or `proportions`.
- `proportions`: if given, assign given genotypes randomly.

7.5.3 Operator (C++) `spread`, function `Spread`

`Spread(ind, subPop)` spread the genotype of ind to all individuals in an array of subpopulations. The default value of subPop is the subpopulation where ind resides.

7.5.4 Operator (hybrid) `pyInit`, function `PyInit`

`pyInit` is a hybrid initializer. User should define a function with parameters allele, ploidy and subpop indices, and return an allele value.

Listing 7.5: Init by value

```
>>> def initAllele(ind, p, sp):
...   return sp + ind + p
...
>>> simu.apply([
...     pyInit(func=initAllele),
...     dumper(alleleOnly=True, dispWidth=2)])
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
```

```
AttributeError: 'simulator' object has no attribute 'apply'
>>>
```

7.6 Migration

Mating is strictly within subpopulations in simuPOP so migrator is the only way to mix genotypes of several subpopulations. Migrators are quite flexible in simuPOP in the sense that

- Migration can happen from and to a subset of subpopulations.
- Migration can be done by probability, proportion or by counts. In the case of probability,
  - if the migration rate from subpopulation `a` to `b` is r, then everyone in subpopulation `a` will have this probability to migrate to `b`.
  - In the case of proportion, exactly `r*size_of_subPop_a` individuals (chosen by random) will migrate to subpop b.
  - In the last case, a given number of individuals will migrate.
- New subpopulation can be generated through migration. You simply need to migrate to a new subpop number.

Note that overall population size will not change. (Mating schemes can do that). If you would like to keep subpop size after migration, you can use the `newSubPopSize` or `newSubPopSizeExpr` parameter of a mating scheme.

7.6.1 Constants: `MigrByProbability, MigrByProportion, MigrByCount`

Possible values of parameter `mode`.

7.6.2 Opertor (C++) `migrator`

Opertor `migrator` is used to migrate from `'fromSubPop'` to `'toSubPop'`. From and to subpop can be a number or an array of subpopulations. The migration probability/rate/counts from `i->j` is specified in the rate matrix. The `'fromSubPop'` and `'toSubPop'` are default to all subpopulations.

An detailed example can be found in 'some real examples' -> 'complex Migration Scheme' section.

7.6.3 Functions (Python) `MigrIslandRates, MigrStepstoneRates` (simuUtil.py)

Migrator is very flexible. It can accept arbitrary migration matrix, from any subset of subpops to any (even new) other subset of subpops. Several functions are defined in simuUtil.py, however, for easy use of popular migration models:

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

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

- `MigrStepstoneRates(r,n,circular=False)` returns a 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`

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

A lot of such functions may be defined later. I guess 2-d stepstone will be the first one?

7.6.4 Operator (C++/Hybrid) `pyMigrator`

For evan more complicated migration schemes, you do DIY it using a `pyMigrator`. This operator is not strictly hybrid since it does not call python function. However, it takes a `carray` as subPop id for each individual. `pyMigrator` then complete migration according its content. For example:

Listing 7.6: pyMigrator

```
>>> simu = simulator(population(subPop=[2,3], loci=[2,5]),
...     randomMating())
>>> # an Numeric array, force to Int type
>>> spID = [2,2,1,1,0]
>>> simu.apply( [
...     initByFreq([.2,.4,.4]),
...     dumper(alleleOnly=True, stage=PrePostMating),
...     pyMigrator(subPopID=spID)
...     ])
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'simulator' object has no attribute 'apply'
>>>
```

Note that

- the application sequence of the operators is `initByFreq`, `dumper, pyMigrator` and then `dumper` again since its stage is set to `PrePostMating`.
- Usually you will use a `pyEval` operator to re-assign `spID` during evolution.

7.6.5 Operator (C++) `splitSubPop`, function `SplitSubPop`

This operator takes parameters

- which: which subpopulation to split. If there is no subpopulation structure, use 0 as the first (and only) subpopulation.

- sizes: new subpopulation sizes. The sizes should add up to the original subpopulation (subpop which) size.
- proportions: Optionally, you can specify proportions of new subpops. (easier to use) The proportions should add up to 1.
- subPopID: the operator will automatically set new subpop ID to new subpops. You can also specify the IDs.

7.6.6 Operator (C++) `mergeSubPops`, function `MergeSubPops`

This operator merges subPopulations `subPops` (the only parameter) to a single subpopulation. If `subPops` are ignored, all subpopulations will be merged.

7.7 Mutation

`rate` can be a number (uniform rate) or an array of mutation rates, `atLoci` is defaulted to all loci. The only differences between the following mutors are they way they actually mutate an allele, and corresponding input parameters.

Mutators record the number of mutation events at each loci. You can retrieve this information using

```
mut.mutationCount(locus)
mut.mutationCounts()
```

where `mut` is any mutator and `locus` is locus index.

7.7.1 Operator (C++) `kamMutator`, function `KamMutate`

`kamMutator` (K-allele mutator) mutate an allele to another allelic state with equal probability. The specified mutation rate is actually 'probability to mutate' so the mutation rate to any other allelic state is actually $\frac{r}{K-1}$, where K is specified by parameter `maxAllele`. Here is an example of mutation:

Listing 7.7: kamMutator

```
>>> simu = simulator(population(size=5, loci=[3,5]), noMating())
>>> simu.apply([
...     kamMutator( rate=[.2,.6,.5], atLoci=[0,2,6], maxAllele=9),
...     dumper(alleleOnly=True)])
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'simulator' object has no attribute 'apply'
>>>
```

You can also specify states for this mutator. If `states` parameter is given, all alleles must be one of the state and mutation will happen among them. `states` is defaulted to `1-maxAllele`.

7.7.2 Operator (C++) `smmMutator`, function `SmmMutate`

Stepwise mutation model (SMM) Mutation model assumes that alleles are represented by integer values and that a mutation either increases or decreases the allele value by one. For variable number tandem repeats loci (VNTR), the allele value is generally taken as the number of tandem repeats in the DNA sequence.

The following example demonstrate the use of `smmMutator`. Note that although the mutation rate is 1, some allele 1 is not mutated since they can not be mutated to 0. The same will hold for the upper bound `maxAllele` which is defaulted to 99 in this case.

Listing 7.8: smmMutator

```
>>> simu = simulator(population(size=3, loci=[3,5]), noMating())
>>> simu.apply([
...     initByFreq( [.2,.3,.5]),
...     smmMutator(rate=1,  incProb=.8),
...     dumper(alleleOnly=True, stage=PrePostMating)])
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'simulator' object has no attribute 'apply'
>>>
```

7.7.3 Operator (C++/Hybrid) `gsmMutator`, function `GsmMutate`

Generalized stepwise model is an extension to stepwise mutation model. In this model, the change in the allelic state is draw from a random distribution. A *geometric generalized stepwise model* uses a geometric distribution with parameter p, which has mean $\frac{p}{1-p}$ and variance $\frac{p}{(1-p)^2}$.

Operator `gsmMutator` implements both models. If you specify a python function without parameter, the operator will use its return value each time a mutation occur; otherwise, a parameter p should be provided and the operator will act as a geometric generalized stepwise model.

Listing 7.9: gsmMutator

```
>>> simu.apply([
...     initByFreq( [.2,.3,.5]),
...     gsmMutator(rate=1, p=.8, incProb=.8),
...     dumper(alleleOnly=True, stage=PrePostMating)])
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'simulator' object has no attribute 'apply'
>>>
>>> import random
>>> def rndInt():
...   return random.randrange(3,6)
...
>>> simu.apply([
...     initByFreq( [.2,.3,.5]),
...     gsmMutator(rate=1, func=rndInt, incProb=.8),
...     dumper(alleleOnly=True, stage=PrePostMating)])
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'simulator' object has no attribute 'apply'
>>>
>>>
```

7.7.4 Operator (Hybrid) `pyMutator`, function `PyMutate`

If you can not accomplish your task with the above normal mutator, you can always use this hybrid mutator. Mutation rate etc are set just like others and you are supposed to provide a python function to return a new allele state given an

old state. `pyMutator` will choose an allele as usual and call your function to mutate it to another allele. Here is an example:

Listing 7.10: pyMutator

```
>>> def mut(x):
...   return 8
...
>>> simu.apply([
...   pyMutator(rate=.5, atLoci=[3,4,5], func=mut),
...   dumper(alleleOnly=True)])
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'simulator' object has no attribute 'apply'
>>>
```

7.7.5 Operator (C++) `pointMutator,` function `PointMutate`

`pointMutator,` as its name suggest, does point mutation. It is syntax is:

```
pointMutator(atLoci, toAllele, atPloidy=[], inds=[], ...)
```

This mutator will turn alleles at `atLoci` on the first chromosome copy to `toAllele` for individual `inds`. You can specify `atPloidy` to mutate other, or all ploidy copy.

7.8 Recombination

Only one recombinator is provided. Recombination events between loci a/b and b/c are independent.

7.8.1 Operator (C++) `recombinator`

This operator takes similar parameters as a mutator. However, because of potentially uneven allelic distance, you should use one of the two parameters:

- `intensity`: intensity of recombination. The actually recombination rate is `intensity*loci distance`.
- `rate`: recombination rate after all `afterLoci`. It can also be an array of recombination rates. Should have length `totNumLoci()` or length of `afterLoci`. The recombination rates are independent of loci distance.
- `afterLoci`: recombine after loci `afterLoci`.
- `maleIntensity, maleRate, maleAfterLoci`: If you need to specify different recombination model for male and female, you can specify these parameters. In this case, `intensity, rate` and `afterLoci` will be treated as female parameters.

The following example forces recombination (with rate 1, an unrealistic value since the maximum recombination rate should be .5) at loci 2,6 and 10. Here I use a `parentsTagger` to mark the parents of each individual so you can (if you have enough patience) see exactly how recombination works.

Listing 7.11: Recombinator

```
>>> simu = simulator(population(4, loci=[4,5,6],
...     infoFields=['father_idx', 'mother_idx']),
...     randomMating())
>>> simu.step([
...   parentsTagger(),
...   ],
...   preOps = [initByFreq([.2,.2,.4,.2]), dumper(alleleOnly=True) ],
...   postOps = [ dumper(alleleOnly=True)]
... )
individual info:
sub population 0:
   0: MU   3  2  3  1   2  1  2  2  2   0  2  1  2  2  1 |   2  2  3  2
2  2  2  0  2   3  1  2  2  2  2
   1: MU   2  0  2  0   1  1  3  2  1   2  2  0  1  2  3 |   1  1  3  3
2  2  3  2  1   1  0  3  3  0  3
   2: FU   3  2  2  1   2  3  1  3  2   1  0  0  2  0  2 |   2  2  2  1
3  2  2  0  2   2  0  0  2  1  0
   3: MU   2  0  0  3   2  2  2  2  0   3  3  2  2  2  0 |   2  0  2  3
1  1  2  2  3   1  2  0  2  0  2
End of individual info.


No ancenstral population recorded.
individual info:
sub population 0:
   0: MU   3  2  3  1   2  2  2  0  2   3  1  2  2  2  2 |   2  2  2  1
3  2  2  0  2   2  0  0  2  1  0
```

```
   1: FU   2  2  3  2   2  2  2  0  2   3  1  2  2  2  2 |   3  2  2  1
2  3  1  3  2   2  0  0  2  1  0
   2: MU   2  0  2  3   2  2  2  2  0   1  2  0  2  0  2 |   2  2  2  1
3  2  2  0  2   1  0  0  2  0  2
   3: FU   2  0  2  3   1  1  2  2  3   3  3  2  2  2  0 |   2  2  2  1
3  2  2  0  2   1  0  0  2  0  2
End of individual info.


No ancenstral population recorded.
True
>>> simu.step([
...   parentsTagger(),
...   recombinator(rate=[1,1,1], afterLoci=[2,6,10])
...   ],
...   postOps = [ dumper(alleleOnly=True)]
... )
individual info:
sub population 0:
   0: FU   2  2  3  1   2  2  2  3  2   3  1  0  2  1  0 |   2  2  2  3
2  2  2  0  2   1  0  0  2  0  2
   1: MU   2  2  3  1   2  2  2  3  2   3  1  0  2  1  0 |   3  2  3  1
3  2  2  0  2   3  1  0  2  1  0
   2: MU   2  0  2  1   1  1  2  0  2   3  3  0  2  0  2 |   3  2  3  1
3  2  2  0  2   2  0  2  2  2  2
   3: FU   2  0  2  1   1  1  2  0  2   1  0  2  2  2  0 |   2  0  2  1
2  2  2  0  2   1  2  0  2  0  2
End of individual info.


No ancenstral population recorded.
True
>>>
```

Note that if `sexChrom()` is true, there is no recombination between the last chromosome (sex chromosomes XY) of male individuals. This may change later if the exchanges of genes between pseudoautosomal region of XY need to be modeled.

Recombinations after each locus will be recorded. You can retrieve this information through functions

```
rec.recCount(locus)
rec.recCounts()
```

where `rec` is the recombinator, `locus` is locus index.

7.9 Selection

7.9.1 Mechanism

Genetic selection is tricky to simulate since there are many difference *fitness* values and many different way to apply selection. `simuPOP` employees an '*ability-to-mate*' approach. Namely, the probability that an individual will be chosen for mating is proportional to its fitness value. More specifically,

- PreMating selectors assign fitness values to each individual.

- During sexless mating (e.g. `binomialSelection`), individuals are chosen at probabilities that are proportional to their fitness values. If there are N individuals with fitness values $f_i, i = 1, ..., N$, individual i will have probability $\frac{f_i}{\sum_j f_j}$ to be chosen to be passed to the next generation.
- During `randomMating`, males and females are separated. Males and females are chosen from their respective groups in the same manner and mate.

It is not very clear that our method agrees with the traditional 'average number of offsprings' definition of fitness. (Note that this concept is very difficult to simulate since we do not know who will determine the number of offsprings if two parents are involved.) We can, instead, look at the consequence of selection in a simpler 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 number of 11 individuals times its probability to be chosen:

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

This is, however, is 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 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 along can not do selection! Only mating schemes can actually select on individuals.
- 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`.

7.9.2 Operator (C++) `mapSelector`, function `MapSelector`

`mapSelector` implements selection at one locus. User should supply a dictionary (map) of fitness values for each genotype and this selector will set each individual's fitness value according to its genotype.

The following example is a typical example of heterozygote superiority. When $w_{11} < w_{12} > w_{22}$, the genotype frequency 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 ($s_1 = .1$, $s_2 = .2$).

Listing 7.12: map selector

```
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[1], infoFields=['fitness']),
...     randomMating())
>>> s1 = .1
>>> s2 = .2
>>> simu.evolve([
...     stat( alleleFreq=[0], genoFreq=[0]),
...     mapSelector(locus=0, fitness={'0-0':(1-s1), '0-1':1, '1-1':(1-s2)}),
...     pyEval(r"'%.4f\n' % alleleFreq[0][1]", step=100)
...     ],
...     preOps=[  initByFreq(alleleFreq=[.2,.8])],
...     end=300)
0.7885
0.3205
0.2950
0.3280
True
>>>
```

7.9.3 Operator (C++) `maSelector`, function `MaSelect`

`maSelector` is called 'multiple-alleles' selector. It separate alleles into two groups: wildtype and disease alleles. wildtype alleles are specified by parameter `wildtype` and any other alleles are considered as diseased allele. `maSelector` accepts an array of fitness

- For single-locus, `fitness` is the fitness for genotype AA, Aa, aa while A stands for wildtype alleles.
- For a two-locus model, `fitness` is the fitness for genotype

	BB	Bb	bb
AA	w_{11}	w_{12}	w_{13}
Aa	w_{21}	w_{22}	w_{23}
aa	w_{31}	w_{32}	w_{33}

  in the order of $w_{11}, w_{12,}..., w_{32}, w_{33}$.
- For more than two-locus, use a table of length 3^n in a order similar to the two-locus model.

7.9.4 Operator (C++) `mlSelector`, function `MlSelect`

`mlSelector` is a 'multiple-loci model' selector. The selector takes a vector of selectors (can not be another `mlSelector`) and evaluate the fitness of an individual as the the product or sum of individual fitness values. The mode is determined by parameter `mode`, which takes the value

SEL_Multiplicative:

The fitness is calculated as $f = \prod_i f_i$.

SEL_Additive:

The fitness is calculated as $f = \max(0, 1 - \sum_i (1 - f_i)) = \max(0, 1 - \sum_i s_i)$. f is set to 0 when $f < 0$. What is added are s_i, not f_i directly.

7.9.5 Operator (Hybrid) `pySelector`, function `PySelect`

`pySelector` accept a list of susceptibility loci and a Python function. For each individual, this operator will pass the genotypes at these loci (in the order of `0-0,0-1,1-0,1-1` etc where X-Y is locus X - ploidy Y, in case of diploid population), generation number, and expect a returned fitness value. This, at least in theory, can accommodate all selection scenarios.

The following example simulate the same scenario as above, with $s_1 = .2, s_2 = .3$ (so$p = .6$) and a `pySelector`. Note that although alleles at two loci are passed, the `sel` function only uses the first one.

Listing 7.13: python selector

```
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[3], infoFields=['fitness'] ),
...     randomMating())
>>>
>>> s1 = .2
>>> s2 = .3
>>> # the second parameter gen can be used for varying selection pressure
>>> def sel(arr, gen=0):
...   if arr[0] == 1 and arr[1] == 1:
...     return 1 - s1
...   elif arr[0] == 1 and arr[1] == 2:
...     return 1
...   elif arr[0] == 2 and arr[1] == 1:
...     return 1
...   else:
...     return 1 - s2
...
>>> # test func
>>> print sel([1,1])
0.8
>>>
>>> simu.evolve([
...     stat( alleleFreq=[0], genoFreq=[0]),
...     pySelector(loci=[0,1],func=sel),
...     pyEval(r"'%.4f\n' % alleleFreq[0][1]", step=25)
...     ],
...     preOps=[  initByFreq(alleleFreq=[.2,.8])],
...     end=100)
0.8180
0.9805
1.0000
1.0000
1.0000
True
>>>
```

7.10 Penetrance

Penetrance is the probability that one will have the disease when he has certain genotype(s). Calculation of penetrance is similar to that of fitness. The parameter set is also similar. An individual will be randomly marked as affected/unaffected according to his penetrance value. For example, an individual will have .8 probability to be affected if the penetrance is .8.

Penetrance can be applied at any stage. The default stage is `DuringMating`. The penetrance will be calculated and affected status is set for each offspring during mating. You can also use penetrance as `PreMating, PostMating` or even `PrePostMating` operator. In these cases, affected status will be set to all individuals according to their penetrance value. It is also possible to store penetrance in a given information field if you provide on using the `infoFields` parameter (e.g. `infoFields=['penetrance']`). This is useful to actaully have a look at penetrance values and see if the values are as expected.

Affected status will be used for statistical purpose, and most importantly, ascertainment. They will be calculated along with fitness although they might not be used at every generation. You can use two operators: one for fitness/selection, active at every generation; one for affected status, active only at ascertainment, to avoid unnecessary calculation of affected status.

Pentrance values are used to set the affectedness of individuals, and are usually not saved. If you would like to know the penetrance value, you need to

- `addInfoField('penetrance')` to the population to analyze. (or use infoFields parameter of the population constructor), and
- use e.g., `mlPenetrance(...., infoFields=['penetrance'])` to add penetrance field to the penetrance operator you use. You can choose a name other than 'penetrance' as long as the field name for the operaor and population match.

Penetrance functions can be applied to the current, all, or certain number of ancestral generations. This is controlled by the `ancestralGen` parameter, which is default to `-1` (all available ancestral generations). You can set it to 0 if you only need affection status for the current generation, or specify a number n for the number of ancestral generations ($n + 1$ total generations) to process. Note that `ancestralGen` parameter is ignored if penetrance operator is used as a during mating operator.

7.10.1 Operator (C++) `mapPenetrance` (post, during-Mating), function `MapPenetrance`

Assign penetrance using a table with keys 'X-Y' where X and Y are allele numbers. For example,

```
mapPenetrance(locus=1, penetrance={'1-1':0, '1-2':0.5, '2-2':1})
```

Note that this dictionary can be more than three elements to accommendate more than one disease alleles.

7.10.2 Operator (C++) `maPenetrance` (post, during-Mating), function `MaPenetrance`

`maPenetrance` is called 'multiple-alleles' penetrance. It separate alleles into two groups: wildtype and disease alleles. wildtype alleles are specified by parameter `wildtype` and any other alleles are considered as diseased allele. `maSelector` accepts an array of fitness for AA, Aa, aa in the single-locus case, and a longer table for multi-locus case. Penetrance is then set for any given genotype. For example

```
maPenetrance(loci=[1,5], wildtype=[1], penetrance=[0,0.5,1,0,0,1,0,1,1])
```

this operator behave the same as the `mapPenetrance` example but will work if there are more than one disease alleles.

7.10.3 Operator (C++) `mlPenetrance`(post, during-Mating), function `MlPenetrance`

`mlPentrance` is the 'multiple-loci' penetrnace calculator. It accepts a list of penetrances and combine them according to the `mode` parameter which can take the values:

PEN_Multiplicative:

The penetrance is calculated as $f = \prod f_i$.

PEN_Additive

The penetrance is calculated as $f = \min(1, \sum f_i)$. f is set to 1 when $f < 0$. What is added are s_i, not f_i directly.

PEN_Heterogeneity:

The penetrance is calculated as $f = 1 - \prod (1 - f_i)$.

Please refer to Risch [1990] for detailed information about these models.

For example, if each locus follows an additive penetrance model, we can have

```
pen = []
for loc in loci:
  pen.append( maPenetrance(locus=loc, wildtype=[1],
    penetrance=[0.0.3,0.6] ) )
# the multi-loci penetrance
penMulti = mlPenetrance(mode=PEN_Multiplicative, peneOps=pen)
```

7.10.4 Operator (Hybrid) `pyPenetrance`(post, during-Mating), function `PyPenetrance`

For each individual, user provide a function to calculate penetrance. This method is very flexible but will be slower than previous operators since a function will be called for each individual. This operator accept the following parameters:

- `loci:` disease susceptibility loci. The genotype *at these loci* will be passed to the provided python function in the form of `loc1_1, loc1_2, loc2_1, loc2_2, ...` if the individuals are diploid.
- `func:` a user-defined function that takes a array of genotype and return a penetrance value. The returned value should be between 0 and 1.

For example, for the same multi-locus model before, we can define is using `pyPenetrance` as

```
def peneFunc(geno):
  p = 1
  for l in range(len(geno)/2):
    p *= (geno[l*2]+geno[l*2+1]-2)*0.3
  return p
penMulti = pyPenetrance(loci=loci, func=peneFunc)
```

As you can see, using this operator, you can define arbitrarily complex penetrance functions. Typical such penetrance functions are interaction between loci (using a multi-locus penetrance table), even random ones.

It would be useful to let peneFunc take parameters. This can be done by defining a python function that return a penetrance function. This may sound intimidating but it is really easy:

```
def peneFunc(table):
  def func(geno):
    return table[geno[0]-1][geno[1]-1]
  return func
# then, given a table, you can do
pen = pyPenetrance(loci=loci, func=peneFunc( ((0,0.5),(0.3,0.8)) ) )
```

Now, for any table, you can use `peneFunc` to return a penetrance function that uses this table.

7.11 Quantitative Trait

Quantitative trait is the measure of certain phenotype for given genotype. Quantitative trait is similar to penetrance in that the consequence of penetrance is binary: affected or unaffected; while it is continuous for quantitative trait.

The following operators/functions calculate quantitative traits for each individual and store the values in the information field specified by the user (default to `qtrait`).

The quantitative trait operators also accept the `ancestralGen` parameter to control the number of generations `qtrait` information field will be set.

7.11.1 Operator (C++) `mapQuanTrait`, function `MapQuanTrait`

Assign quantitative trait using a table with keys 'X-Y' where X and Y are allele numbers. If `sigma` is not zero, the returned value is the sum of trait plus $N\left(0,\sigma^2\right)$. This random part is usually considered as environmental factor of the trait.

7.11.2 Operator (C++) `maQuanTrait`, function `MaQuanTrait`

`maQuanTrait` is called 'multiple-alleles' quantitative trait. It separate alleles into two groups: wildtype and disease alleles. wildtype alleles are specified by parameter `wildtype` and any other alleles are considered as diseased allele. `maQuanTrait` accepts an array of fitness as described before. Quantitative trait is then set for any given genotype. $N\left(0,\sigma^2\right)$ will be added to returned trait value.

7.11.3 Operator (C++) `mlQuanTrait`, function `MlQuanTrait`

`mlQuanTrait` is the 'multiple-loci' QT calculator. It accepts a list of QuanTraits and combine them according to the `mode` parameter which can take the values:

QT_Multiplicative:

The mean of quantitative trait is calculated as $f = \prod f_i$.

QT_Additive:

The mean of quantitative trait is calculated as $f = \sum f_i$.

Note that all σ_i (for f_i) and σ (for f) will all be considerd. I.e, the trait value should be

$$f = \sum_i \left(f_i + N\left(0, \sigma_i^2\right)\right) + \sigma^2$$

for QT_Additive case. If this is not desired, you can set some of the σ to zero.

7.11.4 Operator (Hybrid) `pyQuanTrait`, function `PyQuanTrait`

For each individual, user provide a function to calculate quantitative trait.

7.12 Ascertainment (subset of population)

Ascertainment/sampling refers to ways to select individuals from a population. In simuPOP, ascerntainment operators forms separate populations in a population's namespace. All the following operators work like this except for `pySubset` which thrink the population itself.

Individuals in sampled populations may or may not keep their original order but their index in the whole population is stored in a information field `oldindex`. That is to say, you can use `ind.info('oldindex')` to check the original position of an individual.

Most of the ascertainment operators support the following options:

- `times:` how many times to sample from the population.
- `name`: name of samples in local namespace. This variable is an array of populations of size `times`. Default to `sample`. If `name=''` is set, samples will not be saved in local namespace.
- `saveAs, format:` filename and format to save the samples.
- `nameExpr, saveAsExpr:` expression version of parameter `name` and `saveAs`. They will be dynamically evaluated in population's local namespace.

Two forms of sample size specifications are supported: with/without subpopulation structure. For example, the `size` parameter of `randomSample` can be a number or an array (of the length of number of subpopulations). If a number is given, sample will be drawn from the whole population, regardless of population structure. If an array is given, individuals will be draw from each subpopulation `sp` according to `size[sp]`.

An important special case of sample size specification is when `size=[]` (default). In this case, usually all qualified individuals will be returned.

The function form of these operators are a bit different from others. They do return a value: an array of samples.

7.12.1 function `population::shrinkByIndID()`

This function look at the `subPopID()` field of each individual and remove anyone with value 0.

7.12.2 Operator (C++) `pySubset`, function `PySubset`

This operator shrink a population according to a given array or the `subPopID()` value of each indvidual. Subpopulations are kept intact.

7.12.3 Operator (C++/hybrid) `pySample`, function `PySample`

`PySample(pop, info, name, saveAs, format)` or `Sample(pop)` if you already set info for each individual using `setSubPopID()` function. The operator version of these functions are `pySample(info, times, name, nameExpr, saveAs, saveAsExpr, format)`.

7.12.4 Operator (C++) `randomSample`, function `RandomSample`

`RandomSample(pop, size, times, name, saveAs, format)` will randomly choose size individuals (or `sizes` from subpopulations) and return a new population. The operator version is `randomSample`(`size, times, name, nameExpr, saveAs, saveAsExpr, format`).

The function form returns the samples directly. The operator keeps samples in an array `name` in local namespace. You can acess them through `dvars()` or `vars()` functions.

The orginal subpopulation structure/boundary is kept in the samples.

7.12.5 Opeartor (C++) `caseControlSample`, function `CaseControlSample`

`CaseControlSample(pop, cases, controls, times, name, saveAs, format)` will randomly choose cases affected individuals and controls unaffected individuals. The operator version of this function is `caseControlSample(case, cases, control, controls, times, name, nameExpr, saveAs, saveAsExpr, format)`. The affected status is usually set by penetrance function/operators. The sample populations will have two subpopulations: cases and controls.

You can specify number of cases and controls from each subpop using the array form of the parameters. The sample population will still have only two subpoulations (case/control) though.

A special case of this sampling scheme is when one or both `cases` and `controls` are omitted (zeros). In this cases, all cases and/or controls are chosen. If both parameters are omitted, the sample is effectively the same population with affected and unaffected separated into two subpopulaitons.

The following example shows how to draw a random sample (without replacement of course) from an existing population.

Listing 7.14: random sample

```
>>> # random sample
>>> # [0]: RandomSample already return
>>> #  a list of samples even if times=1 (default)
>>> Dump( RandomSample(pop, 3)[0])
Ploidy:                2
Number of chrom:       3
Number of loci:        2 5 10
Maximum allele state:  255
Loci positions:
               1 2
               1 2 3 4 5
               1 2 3 4 5 6 7 8 9 10
Loci names:
               loc1-1 loc1-2
               loc2-1 loc2-2 loc2-3 loc2-4 loc2-5
               loc3-1 loc3-2 loc3-3 loc3-4 loc3-5 loc3-6 loc3-7 loc3-8 loc3-9 loc3-10
population size:       3
Number of subPop:      1
Subpop sizes:          3
Number of ancestral populations:       0
individual info:
sub population 0:
   0: MU   0  0   0  0  0  0  0   0  0  0  0  0  0  0  0  0  0 |   0  0
0  0  0  0  0   0  0  0  0  0  0  0  0  0  0
   1: MU   0  0   0  0  0  0  0   0  0  0  0  0  0  0  0  0  0 |   0  0
0  0  0  0  0   0  0  0  0  0  0  0  0  0  0
   2: MU   0  0   0  0  0  0  0   0  0  0  0  0  0  0  0  0  0 |   0  0
0  0  0  0  0   0  0  0  0  0  0  0  0  0  0
End of individual info.
```

```
No ancenstral population recorded.
>>>
```

7.12.6 Operator (C++) affectedSibpairSample, function AffectedSibpairSample

The use of this operator needs special preparation for the population. Obviously, to obtain affected sibpairs, we need to know the parents and the affectedness tatus of each individual. Furthermore, to get parental genotype, the population should have ancestralDepth at least 1. The biggest problem, however, comes from the mating scheme we are using.

randomMating() is usually used for diploid populations. The *real random* mating requires that a mating will generate only one offspring. Since parents are chosen with replacement, a parent can have multiple offsprings with different parents. On the otherhand, it is very unlikely that two offsprings will have the same parents. The probability of having a sibling for an offspring is $\frac{1}{N^2}$ (do not consider selection). Therefore, we will have to allow multiple offsprings per mating at the cost of smaller effective population size.

All these requirements come at a cost: multiple ancestral populations, judge affectedness status and tagging will slow down evolution; multiple offsprings will reduce effective population size. Fortunately, simuPOP is flexible enough to let all these happen only at the last several generations. For example, you can do

```
endGen = 1000
# having two offsprings only at the last three generations
def numOffsprings(gen):
  if gen >= endGen - 3:
    return 2
  else
    return 1
# evolve ...
simu = simulator(pop, randomMating(numOffspringsFunc = numOffsprings))
simu.evolve( ...
  parentsTagger(begin = endGen - 3),
  mapPenetrance(...., begin = endGen - 2),
  setAncestralDepth(1, at = endGen -2 )
...)
```

to let your population evolve *normally* and start to store ancestral generations and allow multiple offsprings at the last several generations.

Briefly, you should

- set ancestral depth to at least 1 to allow analyzing of parental generation,
- use parentsTagger to track parents for each individual, with the usual limit of no post-mating migration, and
- allow multiple offsprings at least at the last generation. (You do not have to use fixed number of offsprings. Other mating mode like MATE_GeometricDistribution can also be used.)
- use a penetrance operator to set affected status of each individual

and finally use this operator (or function)

```
affectedSibpairSample(size,  times, name, saveAs, format)
```

to get samples accessible from dvars().name[i]. Each sample will

- have 2*size of paired individuals. (e.g. individual(2n) and individual(2n+1), n=0,1,..., size-1 are siblings.

- have an ancestral generation of the same size, with parents to the sibpairs.
- if `size` is an array, get `size[sp]` sibpairs from subpop `sp` .

Other than samples `name`, variable `numSibpairs` will be set to indicate the total number of affected sibpairs in the population. Subpopulation structure will be kept in the samples so you will know how many individuals are drawn from each subpopulation. (This info is also saved in variable `numSibpairs` of each sample.

7.13 Statistics Calculation

7.13.1 Operator (C++) `stat,` function `Stat`

Operator stat calculate various basic statistics for the population and set variables in local namespace. Other operators/functions can refer to the results from the namespace after `stat` is applied. `Stat` is the function form of the operator.

For each statistics, I will list corresponding parameter (of `stat`), variables and mathematics formula if applicable. Note that these statistics are dependent to each other. For example, heterotype and allele frequency of related loci will be automatically calculated if linkage diseqilibrium is requested.

Population size

- parameter: `popSize=True/False`
- variable:
  - **`numSubPop`** number of subpopulation
  - **`popSize, subPop[sp]['popSize']`**
  - **`subPopSize,`** an array of subpopulation size. Not available for subpopulations.

Number of male/female

- parameter: `numOfMale=True/False`
- variable:
  - **`numOfMale, subPop[sp]['numOfMale']`**
  - **`numOfFemale, subPop[sp]['numOfFemale']`**

Number/proportion of affected/unaffected individuals

- parameter: `numOfAffected=True/False`
- variable:
  - **`numOfAffected, subPop[sp]['numOfAffected']`**
  - **`numOfUnaffected, subPop[sp]['numOfUnAffected']`**
  - **`propOfAffected, subPop[sp]['propOfAffected']`**
  - **`propOfUnaffected, subPop[sp]['propOfUnAffected']`**

Number of distinct alleles at a locus

This is done through the calculation of allele frequency. Therefore, allele frequency will also be calculated if this statistics is requested.

- parameter: **`numOfAlleles=[loc1, loc2, ...]`** where **`loc1`** etc are absolute locus indices.
- variable: a carray of number of alleles for **all loci**. Unrequested loci will have 0 distinct alleles.
  - **`numOfAlleles, subPop[sp]['numOfAlleles']`**, number of distinct alles at each loci. (Calculated only at requested loci.)

Allele frequency/count

- parameter: **`alleleFreq=[loc1, loc2, ...]`** where **`loc1`** etc are loci where allele frequencies will be counted.
- variable: the following `carray` objects will be set. For example **`alleleNum[1][2]`** will be the number of allele 2 at locus 1.
  - **`alleleNum[a]`**, **`subPop[sp]['alleleNum'][a]`**
  - **`alleleFreq[a]`**, **`subPop[sp]['alleleFreq'][a]`**

heterozygote frequency/count

- parameter: **`heteroFreq=[loc1,loc2,...]`**:

  an array of loci to calaulate observed heterozygosity and expected heterozygosity.
- variables: array of observed heterozygosity. **`heteroNum[`**`loc`**`][1]`** is the number of heterozygote 1x, $x \neq 1$**.** Number and frequency (proportion) of heterozygotes are calculated for each allele.

  `HeteroNum[loc]` and **`HeterFreq[loc]`** is the overall heterozygosity number and frequency. I.e., number/frequency of genotype xy $x \neq y$. From this number, we can easily derive number of homozygosity.

  Variables are:
  - **`HeteroNum[loc], subPop[sp]['HeteroNum'][loc]`**, overall heterozygote number
  - **`HeteroFreq[loc], subPop[sp]['HeteroFreq'][loc]`**, overall heterozygote frequency
  - **`heteroNum[loc][allele]`**, **`subPop[sp]['heteroNum'][loc][allele]`**
  - **`heteroFreq[loc][allele]`**, **`subPop[sp]['heteroFreq'][loc][allele]`**

expected heterozygosity

- parameter: **`expHetero=[loc1, loc2,...]`**
- Expected heterozygosity
$$h_{exp} = 1 - p_i^2$$
- variable: **`expHetero[loc], subPop[sp]['expHetero'][loc]`**

homozygosity frequency/count

- parameter: **`homoFreq=[loc1, loc2, ...]`**
- Number and frequency of homozygotes xx.
- variable: **`homoNum[loc]`**, **`homoFreq[loc]`**, **`subPop[sp]['homoNum'][loc]`**, **`subPop[sp]['homoFreq'][loc]`**

genotype frequency/count

- parameter:
  - **`genoFreq=[loc1,loc2,...]`**: an array of loci to calculate genotype frequency. All genotypes in the population will be counted.
  - `hasPhase`: if a/b and b/a are the same genotype. default is True.
- variables: Dictionary variables `genoNum`, `genoFreq`, `subPop[sp]['genoNum']`, `subPop[sp]['genoFreq']` will be set. Note that the index `a, b` of `genoFreq[a][b]` are dictionary keys (unlike list used for alleleFreq etc) so you will get KeyError when you use a wrong key. `genoNum.setDefault(a,{})` is preferred.
  - **`genoNum[a][geno]`**, **`subPop[sp]['genoNum'][a][geno]`**
  - **`genoFreq[a][geno]`**, **`subPop[sp]['genoFreq'][a][geno]`**, number/frequency of genotype `geno` at allele `a`. `geno` has the form `x-y`.

haplotype frequency

- parameter: `haploFreq`: a matrix of haplotypes (allele sequence on the different loci) to count. For example:

  ```
  haploFreq = [ [ 0,1,2 ], [1,2] ]
  ```

  will count all haplotypes on loci 0,1 and 2; and all haplotypes on loci 1, 2.
- variable: Dictionary ariables `haploNum`, `haploFreq`, will be set with keys `0-1-2` etc. For example `haploNum['1-2']['5-6']` is the number of allele pair `5,6` (on loci 1 and 2 respectively) in the population.
  - **`haploNum[haplo]`**, **`subPop[sp]['haploNum'][haplo]`**
  - **`haploFreq[haplo]`**, **`subPop[sp]['haploFreq'][haplo]`**, number/frequency of allele sequencies on loci `haplo`.

Linkage disequilibrium

- parameter:

  ```
  LD: LD = [ [1,2], [ 0,1,1,2],[1,2,1,2]]
  ```

  For each item `[loc1, loc2, allele1, allele2]`, D, D' and r^2 will be calculated based on allele1 at loc1 and allele2 at loc2. If only two loci are given, the LD values are averaged over all allele pairs. For example, for allele A at locus 1 and allele B at locus 2,

$$\begin{aligned} D &= P_{AB} - P_A P_B \\ D' &= D/D_{max} \\ D_{max} &= \begin{cases} \min\left(P_A\left(1-P_B\right), \left(1-P_A\right)P_B\right) & \text{if } D > 0 \\ \min\left(P_A P_B, \left(1-P_A\right)\left(1-P_B\right)\right) & \text{if } D < 0 \end{cases} \\ r^2 &= \frac{D^2}{P_A\left(1-P_A\right)P_B\left(1-P_B\right)} \end{aligned}$$

If A and B are not specified, D, D' and r^2 will be the averaged value: (basically $\sum\sum P_A P_B \,||\,$)

$$\begin{aligned} D &= \sum_i \sum_j P_i P_j \left|D_{ij}\right| \\ D' &= \sum_i \sum_j P_i P_j \left|D'_{ij}\right| \\ r^2 &= \sum_i \sum_j P_i P_j r^2_{ij} = \sum_i \sum_j \frac{D^2_{ij}}{(1-P_i)(1-P_j)} \end{aligned}$$

where p_i and q_j are the population allele frequencies of the ith allele on loc1 and the jth allele on loc2. Please note that some other authors uses

$$r^2 = \sum_i \sum_j \frac{D^2_{ij}}{P_i P_j}$$

If you are sure the later is correct, please send me an email (with reference).

- variables:

  if loc1 and loc2 are specified. The values are LD measures averaged over all possible allele pairs. If al1 and al2 are specified, these values are calculated using these two alleles.

  - **`ld['loc1-loc2']['al1-al2'], subPop[sp]['ld']['loc1-loc2']['al1-al2']`**
  - **`ld_prime['loc1-loc2']['al1-al2'], subPop[sp]['ld_prime']['loc1-loc2']['al1-al2']`**
  - **`r2['loc1-loc2']['al1-al2'], subPop[sp]['r2']['loc1-loc2']['al1-al2']`**
  - **`LD[loc1][loc2], subPop[sp]['LD'][loc1][loc2]`**
  - **`LD_prime[loc1][loc2], subPop[sp]['LD_prime'][loc1][loc2]`**
  - **`R2[loc1][loc2], subPop[sp]['R2'][loc1][loc2]`**

  Please note the difference between the datastructure used for `ld` and `LD`. The names are potentially very confusing but I have no better idea.

F_{st}

- parameter: `Fst: Fst = [0,1,2]`, calculate F_{st}, F_{is}, F_{it} based on alleles at locu 0, 1, 2 respectively. The locus-specific values will be used to calculate `AvgFst` etc. Terms and values that match Weir and Cockerham.

  - F (F_{IT}): the correlation of genes within individuals (inbreeding)
  - θ (F_{ST}): the correlation of genes of difference individuals in the same population (will evaluate for each subpopulation and the population as a whole)
  - f (F_{IS}): the correlation of genes within individuals within populations. Populations refers to subpopulations in simuPOP term.

- variables:

  - **`Fst[loc]`**, **`Fis[loc]`**, **`Fit[loc]`**
  - **`AvgFst, AvgFis, AvgFit`**

Relatedness

The relatedness values between two individuals, or two groups of individuals are calculated according to Queller and Goodnight [1989] and Lynch et al. [1999]. The first one is referred to as `method=REL_Queller` and the second one is `method=REL_Lynch`. The paramters are

- `relGroups:` can be in the form of either `[[1,2,3],[5,6,7],[8,9]]` or `[2,3,4]`. The first form specifies groups of individuals, the second form specifies subpopulations. By default, relatedness between subpopulations are calculated.
- `relLoci:` calculate relativeness values based on `relLoci` loci.
- `relMethod`: either `REL_Queller` or `REL_Lynch`

The results are pairwise relatedness values, in the form of a matrix. Original group or subpopulations numbers are discarded.

- `relatedness[grp1][grp2]` is the relatedness value between grp1 and grp2. There is no subpop level relatedness values.

7.14 Expression and Statements

7.14.1 Operator (C++) `output`

This operator output a simple string. For example,

```
output(r'\n', rep=REP_LAST)
```

output a newline at the last replicate.

7.14.2 Operator (Python) `tab (defined in simuUtli.py)`

Outpur a tab. (Wrapper of output operator)

7.14.3 Operator (Python) `endl (defined in simuUtli.py)`

Outpur a new line. (Wrapper of output operator)

7.14.4 Operator (hybrid) `pyEval`, function `PyEval`

We have seen the `expr` and `stmts` parameter of `pyEval`. These are python expression/statements that will be executed when `pyEval` is applied to a population. Statements can also been executed when `pyEval` is created and destroyed. The corresponding parameters are `preStmts` and `postStmts`. For example, operator `varPlotter` uses this feature to initialize R plot and save plot to a file when finished.

7.14.5 Operator (hybrid) `pyExec`, function `PyExec`

This operator takes a list of statements and execute them. No value will be returned or outputed.

7.14.6 Function (Python) `ListVars (defined in simuUtil.py)`

`ListVars(variable)`

This function list any variable in an indented text format. You can use `listVar(simuVars)` to have a look at all replicates or `listVar(simuVars[0]['subPop'][0])` to see variables for the first subpoulation in replicate one.

7.15 Visualization

There is no special visualizer (there was indeed a matlabPlotter before ver 0.5.9 but I decide to remove it since matlab is not universally available.) Since everything is exposed dynamically, all you need to do is plotting variables in whatever way you prefer. The basic steps are:

- find an appropriate tool. I prefer R/Rpy to any other tools since I am familiar with R. You can make your own choice.
- write a function to plot variable. If you would like to plot history of a variable, you can use the `Aggregator` object defined in `simuUtil.py`.
- wrap this function as an operator.

`simuRPy.py` provides a pure Python operator `varPlotter.` It is defined in `simuSciPy` and `simuMatPlt.py` as well but they are lack of subplot capacity (so the usages are different) due to the limit of SciPy/gplt and MatPlotLib's plotting capacity. Also note as of Apr, 2006, the development of gplt in scipy is stopped so support of simuPOP/simuSciPy is stopped as well.

7.15.1 Operator (Python) `varPlotter (simuRPy.py)`

The use of `varPlotter` is easy, if you would like to

Plotting with history

- plot a number in the form of a varaible or expression, use

  ```
  varPlotter(var='expr')
  ```

- plot a vector in the same window and there is only one replicate in the simulator, use

  ```
  varPlotter(var='expr', varDim=len)
  ```

  where `len` is the dimension of your variable or expression. Each line in the figure represents the history of an item of the array.

- plot a vector in the same window and there are several replicates, use

  ```
  varPlotter(var='expr', varDim=len, numRep=nr, byRep=1)
  ```

  varPlotter will try to use an appropriate layout for your subplots (for example, use 3x4 if numRep=10). You can also specify paramter `mfrow` to change the layout.

- if you would like to plot each item of your array variable in a subplot, use

  ```
  varPlotter(var='expr', varDim=len, byVal=1)
  ```

in case of a single replicate or

```
varPlotter(var='expr', varDim=len, byVal=1, numRep=nr)
```

There will be `numRep` lines in each subplot

Plotting without history

- use option `history=False.` Parameters `byVal, varDim` etc will be ignored.

Other options are

1. `title, xtitle, ytitle:` title of your figure(s). title is defauled to your expression, xtitle is defaulted to `generation`.
2. `win:` window of generations. I.e., how many generations to keep in a figure. This is useful when you want to keep track of only recent changes.
3. `update:` update figure after `update` calls. This is used when you do not want to update the figure too often, maybe for efficiency purpose.
4. `saveAs:` save figures in files `saveAs#gen.eps.` For example, if `saveAs='demo'`, you will get files `demo1.eps`, `demo2.eps` etc.
5. `separate:` plot data lines in separate panel.
6. `image:` use R image function to plot image, instead of lines.
7. `level:` level of image colors. default to 20
8. `leaveOpen:` whether or not leave the plot open when plotting is done. Default to true.

Here is an example:

7.15.2 plot through `python/SciPy/MatPltLib`

`varPlotter` is also available for SciPy/MatPltLib but there is no subplots (so no byVal etc) and the usage is different. The `__init__` function of `varPlotter` takes the following parameters:

- `win` window size. Actually the number of generations to display. default to 0 (no limit). If this is set to be a positive number, only the last `win` data will be displayed.
- `update` generations between successive re-draw of figure. We can not use '`step`' parameter of `pyEval` operator since we need to collect data at each generation.
- `title, xtitle, ytitle` titles/labels to be displayed
- `legend` an array of strings, legend of the lines. If ignored, "var0", "var1" etc will be used. If gives only one string (e,g, str), "str", "str1",... etc will be used. Otherwise, the length of legend has to be the same as data size.

7.15.3 Object (Python) freqPlotter (defined in simuRPy.py)

The plotting function used for Reich's simulation, using R as plotting engine. It is put in simuRPy.py mainly for demonstration purposes.

7.16 Tagging (used for pedigree tracking)

7.16.1 Operator (C++) inheritTagger, duringMating

This during-mating operator will copy the tag info from his/her parents. Depending on

- mode = TAG_Paternal
- mode = TAG_Maternal
- mode = TAG_Both

this tagger will obtain tag from his/her father (two tag fields), monther (two tag fields) or both (first tag field from both). You can check the tagger test under test directory for an example.

An example may be tagging one or a few parents and see, at the last generation, how many offspring they have.

7.16.2 Operator (C++) parentsTagger, duringMating

This during-mating operator set tag(), currently a pair of numbers, of each individual with indices of his/her parents in the parent population. This info will be used by pedigree-related operators like affectedSibpairSample to track pedigree information. Since parental population will be discarded or stored after mating, and tagging info will be passed with individuals, mating/population change etc will not interfere with this simple tagging system.

7.17 Data collector

Sometimes, instead of output data directly, we may want to collect history data on some expression. Data collector is designed for this purpose.

7.17.1 operator (Python) collector, in simuUtil.py

This operator accepts the following parameters:

- name: name by which the collected data will be displayed. Variable name will be list of stored values. (generation is not stored. You can always put it in expr though.)
- expr: an expression that will be evaluated. The result will be converted to a list (if needed) and stored in name[gen].

When this operator is called, it will evaluate expr and store its result in name[gen]. After evolution, you will get a dictionary of values indexed by generation number.

7.18 Output

7.18.1 operator (C++) `savePopulation`

7.18.2 function (Python) `SaveFstat` (in `simuUtil.py`)

7.18.3 operator (Python) `saveFstat` (in `simuUtil.py`)

7.18.4 function (Python) `loadFstat` (in `simuUtil.py`)

7.19 Terminator

These operators are used to see if an evolution is running as expected, and terminate the evolution if a condition fails.

7.19.1 Operator (C++) `terminateIf`

This operator terminates the evolution under certain conditions. For example,

```
terminateIf(condition='alleleFreq[0][1]<0.05', begin=100)
```

terminate the evolution if the allele frequency of allele 1 at locus 0 is less than 0.05. Of course, to make this opertor work, you will need to use an `stat` operator before it so that variable `alleleFreq` exists in the local namespace.

7.19.2 Operator (C++) `continueIf`

The same as `terminateIf` but continue if the condition if true.

7.20 Conditional operator

7.20.1 Operator (C++) `ifElse`

`ifElse` is an interesting operator. It accepts:

- an expression that will be evaluated when `ifelse` is called.
- an operator that will be applied if the expression is true. (defult to null)
- an operator that will be applied if the expression if false. (default to null)

When this operator is applied to a population, it will evaluate the expression and depend on its value, apply the supplied operators. Note that the `begin, at, step, at` parameters of if/else operators will be ignored. For example, you can mimic the `at` parameter of an operator by

```
ifElse('rep in [2,5,9]', operator)
```

Anyway, the real use of this machanism is minitoring the population statistics and act accordingly. The following example uses some advanced operators of simuPOP:

- set affected status using `maPenetrance` as a `DuringMating` operator. (penetrance can be used at other stages)
- count the number of affected individuals. Note that this has to be done after the penetrance operator is applied.
- If no one is effected, inject some mutations into the population. Note the use of `ifElse` operator.
- expose individual affectedness to local namespaces. Note the use of `exposePop` option. With this, you can call any population member function.
- plot affectedness, use image.
- Use dryrun to exam simulator first.

Listing 7.15: Conditional operator

```
>>> from simuRPy import *
>>> from simuUtil import *
>>> numRep=4
>>> popSize=100
>>> endGen=50
>>>
>>> simu = simulator(population(size=popSize, loci=[1]),
...   randomMating(), rep=numRep)
>>> simu.evolve(
...   preOps = [ initByValue([1,1])],
...   ops = [
...     # penetrance, additve penetrance
...     maPenetrance(locus=0, wildtype=[1], penetrance=[0,0.5,1]),
...     # count number of affected
...     stat(numOfAffected=True),
...     # introduce disease if no one is affected
...     ifElse(cond='numOfAffected==0',
...       ifOp=kamMutator(rate=0.01, maxAllele=2)),
...     # expose affected status
...     pyExec('pop.exposeAffectedness()', exposePop=True),
...     # plot affected status
...     varPlotter(expr='affected',plotType="image", byRep=1, update=endGen,
...       varDim=popSize, win=endGen, numRep=numRep,
...       title='affected␣status', saveAs="ifElse")
...   ],
...   end=endGen,
...   dryrun=False
... )
Traceback (most recent call last):
  File "<embed>", line 1, in ?
AttributeError: 'population' object has no attribute 'exposeAffectedness'
PostMating operator <simuPOP::pyExec > throws an exception.

Traceback (most recent call last):
  File "userGuide.py", line 19, in ?
    try:
SystemError: Evalulation of statements failed
>>>
```

7.21 Miscellaneous

7.21.1 Operator: (C++) noneOp

This operator does nothing. It is used like follows:

```
if savePop :
  saveOp = savePopulation(output='a.txt')
else:
  saveOp = noneOp()
simu.evolve( [ ... saveOp ])
```

7.21.2 Operator: (C++) pause

This operator will pause the simulation and wait for user response. User can use 'q' to stop evolution, 's' to escape to a python shell, or any other key to continue.

There are two ways to use this operator, the first one is to pause the simulation at specified generations, using the usual operator parameters like `at`. Another way is to pause a simulation with any key stroke, using the `stopOnKeyStroke` parameter. This feature is useful for presentation and interative simulation.

When 's' is pressed, this operator expose the current population to the main python dictionary as variable 'pop' and enter an interactive python session. The way current population is exposed can be controled by parameter `exposePop` and `popName`. This feature is useful when you want to examine the properties of a population during evolution.

7.21.3 Operator: (C++) ticToc, function TicToc

This operator, when called, output the difference between current and last called clock time. This can be used to estimate execution time of each generation. Similar information can also be obtained from

```
turnOnDebug(DBG_PROFILE)
```

but this operator has the advantage of measuing duration between several generations (set `step` parameter.)

7.21.4 Operator: (C++) setAncestralDepth, function pop.setAncestralDepth

`setAncestralDepth` set the number of ancestral generations to keep in a population. This is useful when constructing pedigree trees from a population.

7.22 Random Number Generator

Random number generator is a tricky business. Reliable and fast RNGs are hard to find and everyone seems to trust/distrust certain RNGs. To avoid such arguments, I have included all RNGs from GNU Scientific Library and you can choose any of the 61 RNGs, if you really know what the differences between them. (I do not, except that some of them are really bad but fast.) Note that RNG that can not generate a full range of integers are removed.

Listing 7.16: Random number generator

```
>>> print ListAllRNG()
('gfsr4', 'mt19937', 'mt19937_1999', 'mt19937_1998', 'r250', 'rand', 'rand48', 'random12
>>> print rng().name()
mt19937
```

```
>>> SetRNG("taus2", seed=10)
>>> print rng().name()
taus2
>>>
```

If you need to use a random number generator in your `pyEval` operator, you can either use python random module (`import random`) or use `rng()` function to get the random number generator of simuPOP. Note that `rng()` does not have many member functions and it might be tricky to use them correctly. (This object is not designed to be used at the Python level. For a full list of member functions, check `src/utility.h`)

Listing 7.17: Random number generator

```
>>> r=rng()
>>> #help(RNG)
>>> for n in range(1,10):
...   print r.randBinomial(10, .7),
... #end
...
8 7 7 8 7 6 7 7 8
>>>
>>>
>>>
```

Since simuPOP 0.7.1, RNGs are seeded in the following order:

- use random number from /dev/urandom if it is available
- use random number from /dev/random if it is available
- use python expression (random.randint(0, sys.maxint) + int(time.time())) % sys.maxint . This method is used only when simuPOP is first loaded so if you are going to set random number generator by youself, the relevant code in simuPOP.py is recommended.

The seed can also be retrieved using rng().seed() function, which should be saved for serious simulations.

7.23 Debug-related operators/functions

Standard `simuPOP` library can print out lots of debug information upon request. These are mostly for internal debuging use but you can also use them when error happens. For example, the following code will crash simuPOP:

```
>>> population(1).individual(0).arrAllele()
```

It is not clear why this simple line will cause us trouble, instead of outputting the genotype of the only individual of this population. However, the reason is clear if you turn on debug info:

```
>>> TurnOnDebug(DBG_ALL)
Debug code DBG_ALL is turned on. cf. listDebugCode(), turnOffDebug()
>>> population(1).individual(0).arrAlleles()
Constructor of Population is called
Population size 1
Destructor of Population is called
Segmentation fault (core dumped)
```

`population(1)` creates a temporary object that is destroyed right after the execution of the input. When Python tries to display the genotype, it will refer to an invalid location. The right way to do this is create a persistent population object:

```
>>> pop = population(1)
>>> pop.individual(0).arrAllele()
```

If the output is overwhelming after you turn on all debug info, you can turn on certain part of the info by using the following functions:

- `ListDebugCode()` list all debug code.
- `turnOnDebug(), TurnOnDebug(code)` turn on a debug code
- `turnOffDebug(), TurnOffDebug(code)` turn off debug code

`turnOnDebug()` and `turnOffDebug()` are operators and accept all operator parameters `begin, step` etc. Usually, you can `turnOndebug` before simuPOP starts to misbehave to output more info about a potential bug.

Another useful debug code is `DBG_PROFILE.` When turned on, it will display running time of each operator. This will give you a good sense of which operator runs slowly (or simply the order of operator execution if you are not sure). If most of the excution time is spent on a pure-python operator, you may want to rewrite it in C++. Note that when `DBG_PROFILE` is suitable for measuing individual operator performance. If you would like to measure execution time of all operators in several generations, `ticToc` operator is better.

CHAPTER

EIGHT

Extending simuPOP

simuPOP can be extended easily using Python programming language. Because almost all data are exposed to the python interface, your ability of extending simuPOP is unlimited. However, because Python is slower than C++ and the exchange of data between internal C++ data structure and python interface may be costly, it is not recommended to write frequently used operators in python. Appropriate pure python operators are visualizers, statistics calculator, file outputers etc.

To write smuPOP extension, you will have to know more data structure and member functions of population. Note that for efficiency and implementation reasons, many of the following functions do not provide keyword parameters.

8.1 Genotypic structure

The genotypes of an individual are organized as a single array. For example, if you have an diploid individual with two chromosomes, having 2 and 3 loci respectively. The genotypes should be in the order of

```
0-0-0, 1-0-0, 0-1-0, 1-1-0, 2-1-0,0-0-1, 1-0-1, 0-1-1, 1-1-1,2-1-1
```

where X-X-X are locus-chromosome-ploidy indices. An important consequence of this arrangement is that 'locus location' + 'total number of loci' is the location of the locus on the other set of chromosomes.

Several functions are provided to retrieve genotypic info:

- `ploidy()`
- `numChrom()`
- `numLoci(chrom)`, number of loci on chromosome chrom
- `totNumLoci()`
- `genoSize()`, size of genotype. Equals to `totNumLoci()*ploidy()`
- `alleleName()`, if not specified by `alleleNames` parameter when creating a population, return allele number
- `locusPos(loc)`, locus position on chromosome (Distance to the beginning of chromosome)
- `arrlociPos()`, return an `carray` of the loci distance.

The last function is very interesting. It actually return the reference of the internal loci distance array. If you change the value of the returned array, the internal loci distance will be changed! All functions with this property will be named `arrFunc()`.

The following example shows how to change locus distance through this function.

Listing 8.1: geno stru

```
>>> pop = population(1, loci=[2,3,4])
>>> print pop.numLoci(1)
3
>>> print pop.locusPos(2)
1.0
>>> dis = pop.arrLociPos()
>>> print dis
[1.0, 2.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0, 4.0]
>>> dis[2] = 0.5
>>> print pop.locusPos(2)
0.5
>>> print pop.arrLociPos()
[1.0, 2.0, 0.5, 2.0, 3.0, 1.0, 2.0, 3.0, 4.0]
>>>
```

8.2 Accessing genotype and other info

Genotype of an individual can be retrieved through the following functions

- `ind.allele(index, p=0)`
- `ind.setAllele(value, index, p=0)`
- `ind.arrGenotype(p=0, ch=0)`

`p` means ploidy. I.e., the index of copies of chromosomes. `ch` means chromosome. For example

```
pop.individual(1).arrGenotype(1, 2)
```

returns an array of alleles on the third chromosome of the second copy of chromosomes, of the second individual in the population `pop`.

Listing 8.2: genotype

```
>>> InitByFreq(pop, [.2,.8])
>>> Dump(pop, alleleOnly=1)
individual info:
sub population 0:
   0: FU   1  0   1  1  0   1  1  1  1 |   0  1   1  1  1   1  1  1  0
End of individual info.


No ancenstral population recorded.
>>> ind = pop.individual(0)
>>> print ind.allele(1,1)
1
>>> ind.setAllele(3,1,1)
>>> Dump(pop, alleleOnly=1)
individual info:
sub population 0:
   0: FU   1  0   1  1  0   1  1  1  1 |   0  3   1  1  1   1  1  1  0
End of individual info.
```

```
No ancenstral population recorded.
>>> a = ind.arrGenotype()
>>> print a
[1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 3, 1, 1, 1, 1, 1, 1, 0]
>>> a = ind.arrGenotype(1)
>>> print a
[0, 3, 1, 1, 1, 1, 1, 1, 0]
>>> a = ind.arrGenotype(1,2)
>>> print a
[1, 1, 1, 0]
>>> a[2]=4
>>> # the allele on the third chromosome has been changed
>>> Dump(pop, alleleOnly=1)
individual info:
sub population 0:
   0: FU   1  0   1  1  0   1  1  1  1 |   0  3   1  1  1   1  1  4  0
End of individual info.


No ancenstral population recorded.
>>>
```

Sex, affected status can be accessed through `sex, setSex, affected, setAffected` functions.

Listing 8.3: genotype

```
>>> print ind.sex()
2
>>> print ind.sexChar()
F
>>> ind.setSex(Female)
>>> ind.setAffected(True)
>>> print ind.tag()
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'individual' object has no attribute 'tag'
>>> ind.setTag([1,2])
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'individual' object has no attribute 'setTag'
>>> Dump(pop)
Ploidy:                2
Number of chrom:       3
Number of loci:        2 3 4
Maximum allele state:  255
Loci positions:
               1 2
               0.5 2 3
               1 2 3 4
Loci names:
               loc1-1 loc1-2
               loc2-1 loc2-2 loc2-3
```

```
                loc3-1 loc3-2 loc3-3 loc3-4
population size:        1
Number of subPop:       1
Subpop sizes:           1
Number of ancestral populations:        0
individual info:
sub population 0:
   0: FA   1  0   1  1  0   1  1  1  1 |   0  3   1  1  1   1  1  4  0
End of individual info.


No ancenstral population recorded.
>>>
```

8.2.1 Direct populaiton manipulation

FIXME

8.3 Writing Pure Python Operator

Now we know how to access information for individuals in a population, but how can we use them in reality? Namely, how can you write an pure Python operator?

8.3.1 Use `pyOperator`

There are two kinds of pure python operators. The first one is easy: define a function and wrap it with a pyOperator operator. This method is highly recommended because of its simplicity. Many user scripts will use this kind of pure python operator. You can find such examples in scripts directory. A good one may be simuCDCV.py where a pure python operator is used to calculate and visualize special statistics.

For example, if you would like to record a silly statistics, namely the genotype of the m individual at locu n, you can do:

```
def sillyStat(pop, para):
  # para can be used to pass any number of parameters
  (filename, m, n) = para # unpack parameter
  f = open(filename)
  f.write('%d␣' % pop.individual(m).allele(n) )
  f.close()
# then in the evole function
evolve(...
  ops=[ # other operators
    pyOperator(func=sillyStat, param=('file.txt', 2, 1) )
  ]
)
```

`pyOperator` is by default a post-mating operator, you can redefine its stage by `stage` parameter.

8.3.2 Use Python eval function

This kind of pure python operator acts more like an ordinary operator. They are usually pyEval or pyExec operators returned by a wrapper function. For example, the following function defines a `tab` operator:

Listing 8.4: Tab operator

```
>>> def tab(**kwargs):
...   parm = ''
...   for (k,v) in kwargs.items():
...     parm += ', ' + str(k) + '=' + str(v)
...   cmd = r'output( """\t""" ' + parm + ')'
...   # print cmd
...   return eval(cmd)
... #end
...
```

This function actually returns an operator

```
output(r"\t", rep=REP_LAST, begin=500)
```

This kind of operators have some advantages, namely

- it acts more like ordinary operator.
- it is more efficient since it is handled (at least the first layer) by a C/C++ operator.

However, because of its complexity, such operators can only be found in system libraries. You can ignore the rest of this section if `pyOperator` is enough to you.

To define a pure python operator, here is what you will generally do:

- Write a function that act on a population. This function should be able to be called like `func(simu.population(0))` .
- Wrap this function as an operator.

For example, function `saveInFstatFormat(pop, output, outputExpr, dict)` saves a population in FSTAT format. Its defnition is (first 15 lines)

Listing 8.5: genotype

```
>>> print ind.sex()
2
>>> print ind.sexChar()
F
>>> ind.setSex(Female)
>>> ind.setAffected(True)
>>> print ind.tag()
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
AttributeError: 'individual' object has no attribute 'tag'
>>> ind.setTag([1,2])
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    #
```

```
AttributeError: 'individual' object has no attribute 'setTag'
>>> Dump(pop)
Ploidy:                 2
Number of chrom:        3
Number of loci:         2 3 4
Maximum allele state:   255
Loci positions:
                1 2
                0.5 2 3
                1 2 3 4
Loci names:
                loc1-1 loc1-2
                loc2-1 loc2-2 loc2-3
                loc3-1 loc3-2 loc3-3 loc3-4
population size:        1
Number of subPop:       1
Subpop sizes:           1
Number of ancestral populations:        0
individual info:
sub population 0:
   0: FA   1  0   1  1  0   1  1  1  1 |   0  3   1  1  1   1  1  4  0
End of individual info.


No ancenstral population recorded.
>>>
```

Note that

- You can use this function independently like

```
saveInFstatFormat(simu.population(1),'a.txt')
```

- `pop.vars()` is used to evaluate `outputExpr`

Then you can wrap this function by an operator, actually a function that returns a `pyEval` operator:

Listing 8.6: save fstat

```
>>> def saveFstat(output='', outputExpr='', **kwargs):
...   # deal with additional arguments
...   parm = ''
...   for (k,v) in kwargs.items():
...     parm += str(k) + '=' + str(v) + ', '
...   # pyEval( exposePop=1, param?, stmts="""
...   # saveInFSTATFormat( pop, rep=rep?, output=output?, outputExpr=outputExpr?)
...   # """)
...   opt = '''pyEval(exposePop=1, %s
...     stmts=r\'\'\'saveInFstatFormat(pop, rep=rep, output=r"""%s""",
...     outputExpr=r"""%s""" )\'\'\')''' % ( parm, output, outputExpr)
...   # print opt
...   return eval(opt)
... #end
...
>>>
```

This function takes all parameters of an ordinary operator:

```
saveFstat(at=[-1], outputExpr=r'a'+str(rep)+'.txt')
```

and generate a `pyEval` operator (use above example)

```
pyEval(exposePop=1, at=[-1],
  stmts=r"""saveInFSTATFormat(pop,
    output='''''', outputExpr=r''' 'a'+str(rep)+'.txt' """
  )
```

In this example,

- `pyEval` works in the local namespace of each replicate. To access that replicate of population, you should use the magic paramter `exposePop` of `pyEval`. When set true, `pyEval` will automatically set a variable `pop` in the current local namespace before any statement is executed. This is why we can call `saveInFSTATFormat(`**`pop`**`...)`
- `"'a"'` quotes are used to avoid conflict with quotes in `outputExpr` etc.

.

8.4 Ultimate extension: working in C++

It is sometimes desired to write simuPOP extension in C++. For example,

- when you need some other mating scheme
- when you need certain operator that a pure Python implementation would be too slow.
- If some aspect of simuPOP is too limited (like the number of maximum alleles)

It is not difficult to write simuPOP extension in C++, once you know how simuPOP is organized. The general procedure is

- install the latest version of SWIG (>1.3.28)
- check out simuPOP source using subversion
- build from source and see if your programming environment works well
- to add an operator, make changes in appropriate `.h` file, check `simuPOP_common.i` if your operator can not be used.

The source code is reasonably well commented with full doxygen based documentation. Please post to the simuPOP forum if you encounter any problem while writing operators in C++.

8.5 Debugging

8.5.1 test scripts

There are many test scripts under the `test` directory. It is recommended that you run the test scripts after you install simuPOP. This will make sure that your system is working correctly. To run all tests, run

```
sh run_tests.sh
```

Or, if you do not install rpy and r, run

```
sh run_tests.sh norpy
```

Please report any failed test.

8.5.2 Memory leak detection

Python extensions tend to have memory leak problem, caused by the refcount mechanism. If your simuPOP script uses more and more RAM without population size increase, you may have this problem. You may try to disable individual operators and see find out the offending operator if the problem persist.

Potential simuPOP developers can make use of simuPOP's built-in refcount detection mechanism. To use it,

- Compile python with configure option –with-pydebug . This will enable sys.totalrefcount() etc

- Compile simuPOP with -DPy_REF_DEBUG. This can be done in setup.py, or better in SConstruct.

simulator.evolve will check reference count at the end of each generation and report any increased reference count. Some operators may create python object (like ascertainment operators) but if you see repeated warning at each generation, there is definitely a memory leak.

BIBLIOGRAPHY

M Lynch, Ritl, and K. Estimation of pairwise relatedness with molecular markers. *Genetics*, 152(4):1753–1766, Aug 1999.

DC Queller and KF Goodnight. Estimating relatedness using genetic markers. *Evolution*, 43(2):258–275, Mar 1989.

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

BS Weir and CC Cockerham. Estimating f-statistics for the analysis of population structure. *Evolution*, 38(6):1358–1370, Nov 1984.

INDEX