APPENDIX

FORWARD-TIME SIMULATIONS USING simuPOP

A.1 INTRODUCTION

A.1.1 What is simuPOP?

simuPOP is a *general-purpose individual-based forward-time population genetics simulation environment* based on Python, a dynamic *object-oriented* programming language that has been widely used in biological studies. More specifically,

- simuPOP is a *population genetics simulator* that simulates the evolution of populations. It uses a discrete generation model, although overlapping generations could be simulated using nonrandom mating schemes.
- simuPOP explicitly models *populations with individuals* who have their own genotype, sex, and auxiliary information such as age. The evolution of a population is modeled by populating an offspring population with offspring produced from parents in this population.

Forward-time Population Genetics Simulations: Methods, Implementation, and Applications, Bo Peng, Marek Kimmel, and Christopher I. Amos.

- Unlike coalescent-based programs, simuPOP evolves populations *forward in time*, subject to an arbitrary number of genetic and environmental forces such as mutation, recombination, migration, and population size changes.
- simuPOP is a *general-purpose* simulator that does not aim at any particular application area. It is a development tool with which a large number of simulations can be implemented. Owing to an *object-oriented design*, all classes can be extended by users to define customized genetic effects in Python. In contrast, other programs either do not allow customized effects or force users to modify code at a lower (e.g., C/C++) level.

simuPOP consists of a number of Python modules that provide a large number of Python classes and functions, including population, mating schemes, operators, (objects that manipulate populations) and simulators to coordinate the evolutionary processes. More than 70 operators are provided, covering all important aspects of genetic studies such as mutation, migration, recombination, gene conversion, natural selection, penetrance, quantitative trait, statistics calculation, and sample generation. In addition, because simuPOP provides a large number of functions to manipulate populations, it can also be used as a data manipulation and analysis tool.

Although it is generally easy to translate an evolutionary process into a simuPOP script, it can be a daunting task if the script involves complex demographic and genetic features and functions to interoperate with other applications and file formats. Fortunately, an increasing list of simuPOP functions and scripts, many of which are contributed by simuPOP users, are provided in the simuPOP online cookbook (`http://simupop.sourceforge.net/cookbook`). These recipes include functions to manipulate genetic data in other formats (e.g., the HapMap data set) [1], user-defined operators to extend the functionality of simuPOP, examples to use some of the advanced features of simuPOP, scripts to demonstrate classic population genetic models, and complete scripts that simulate a variety of evolutionary processes. It is strongly recommended that users of simuPOP make use of this resource and contribute to it whenever possible.

A.1.2 An Overview of simuPOP Concepts

A simuPOP *population* consists of *individuals* of the same *genotype structure*, which consists of properties such as number of homologous sets of chromosomes (ploidy), number of chromosomes, names and locations of

markers on each chromosome, and names of auxiliary information attached to each individual (*information fields*). Individuals can be divided into *subpopulations* that can be further grouped into *virtual subpopulations* (VSPs) according to individual properties such as sex and affection status. Each population has a dictionary, called its *local namespace*, that is used to store arbitrary Python variables.

simuPOP uses *a discrete-generation model* in which the evolution of a population for one generation is characterized by the generation of an offspring population from a parental population (Figure A.1). During this process, arbitrary numbers of *operators* (Python objects that act on a population) can be applied to the parental population (*premating operator*), offspring population (*postmating operators*), or to offspring when he or she is produced (*during-mating operators*). At the end of a generation, the offspring population becomes the parental population of the next generation. This process can repeat for specified generations or be terminated if a pre- or postoperator fails to apply.

A simuPOP *mating scheme* is responsible for choosing parent or parents from a parental (virtual) subpopulation and for populating an offspring population. simuPOP provides a number of predefined *homogeneous mating schemes*, such as random, monogamous, or polygamous mating, selfing, and haplodiploid mating in hymenoptera. More complicated nonrandom mating schemes such as mating in age-structured populations can be constructed using *heterogeneous mating schemes*, which apply multiple homogeneous mating schemes to different (virtual) subpopulations.

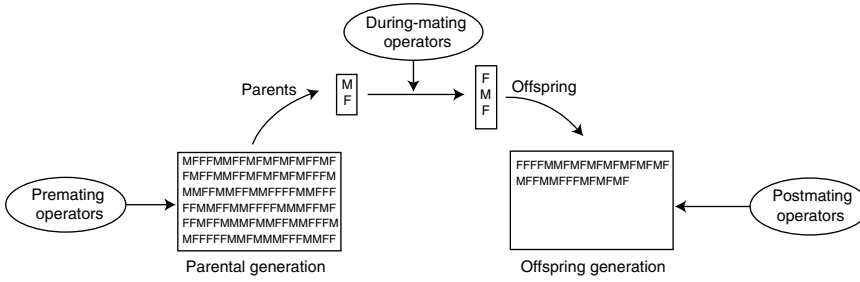

FIGURE A.1 A life cycle of an evolutionary process. Illustration of the discrete-generation evolutionary model used by simuPOP. A life cycle of a generation starts from a parental population and ends at an offspring population. Operators can be applied to the parental population before mating and to the offspring population after mating. A mating scheme is responsible for choosing parents and produce offspring. During-mating operators are used to transmit genotype and other information from parents to offspring.

SOURCE CODE A.1 A Simple Example

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=1000, loci=2)
>>> pop.evolve(
...     initOps=[
...         # Initialize individuals with random sex (MALE or FEMALE)
...         sim.InitSex(),
...         # Initialize individuals with two haplotypes.
...         sim.InitGenotype(haplotypes=[[1, 2], [2, 1]])
...     ],
...     # Random mating using a recombination operator
...     matingScheme=sim.RandomMating(ops=sim.Recombinator(rates=0.01)),
...     postOps=[
...         # Calculate Linkage disequilibrium between the two loci
...         sim.Stat(LD=[0, 1], step=10),
...         # Print calculated LD values
...         sim.PyEval(r"'%2d: %.2f\n' % (gen, LD[0][1])", step=10),
...     ],
...     gen=100
... )
 0: 0.25
10: 0.22
20: 0.19
30: 0.17
40: 0.16
50: 0.17
60: 0.16
70: 0.14
80: 0.12
90: 0.10
100
```

These concepts are demonstrated in Source code A.1, where a standard diploid Wright–Fisher model with recombination is simulated. This source code records a Python interactive session where Python commands are executed immediately after they are entered. This is extremely useful for debugging and testing and for the demonstration of simuPOP features. On the other hand, you can put all the commands in a file (usually with a `.py` file extension) and execute the file in batch mode. Refer to the Python documentation for how to write a Python script.

The first line of Source code A.1 imports the standard simuPOP module. It imports simuPOP as module `sim` to make the script easier to read (e.g., `sim.InitSex` instead of `simuPOP.InitSex`). The second line creates a diploid population of 1000 individuals, each having 1 chromosome with 2 loci. The `evolve()` function evolves the population using a random mating scheme and five operators.

Operators `InitSex` and `InitGenotype` are applied at the beginning of the evolutionary process. Operator `InitSex` initializes individual sex randomly and `InitGenotype` initializes all individuals with two

haplotypes `-1-2-` and `-2-1-` at equal probabilities (default value of parameter `freq`). The populations are then evolved for 100 generations. A random mating scheme is used to generate offspring by selecting male and female individuals randomly from the parental population. Instead of using the default Mendelian genotype transmitter that transmits one of the two homologous chromosomes from parents to offspring, a `Recombinator` (during-mating operator) is used to recombine parental chromosomes with a recombination rate of `0.01` before one of the recombinants is transmitted to offspring.

Two operators are applied to the offspring generation (postmating) at every 10 generations (parameter `step`). Operator `Stat` calculates linkage disequilibrium between the first and second loci. The results of this operator are stored in the local namespace of the population. The last operator `PyEval` retrieves calculated linkage disequilibrium values from the local namespace and outputs it with a generation number and a trailing new line. The result represents the decay of linkage disequilibrium of this population at 10 generation intervals. The return value of the `evolve` function, which is the number of evolved generations, is also printed.

A.2 POPULATION

Populations are the most important objects in simuPOP because all other functions and objects are designed to examine or change population properties. This section describes how to create a population, and how to examine and modify its properties using its member functions. Because there are more than 80 member functions in the `Population` class, this section lists only some of the more frequently used ones. Refer to the simuPOP reference manual for a complete list and description of all member functions.

A.2.1 Creating a Population

A `Population` object consists of one or more generations of individuals, grouped by subpopulations, and a Python dictionary to hold arbitrary variables. It is instantiated (a process to create an object from the construction function of the corresponding class) from the `__init__` function of class `Population`. Several parameters can be used to specify the genotype and population structure of a population. Acceptable parameters and their usages are listed in Table A.1.

TABLE A.1 Parameters to Create a `Population` Object

Parameter with Default Value	Usage	Examples
`size=[]`	A list of subpopulation sizes. The length of this list determines the number of subpopulations of this population	`size=2000` `size=[1000, 2000]`
`ploidy=2`	Number of homologous sets of chromosomes	`ploidy=1`
`loci=[]`	Numbers of loci on each chromosome. The length of this parameter determines the number of chromosomes	`loci=10` `loci=[20]*5`

`chromTypes=[]`	A list that specifies the type of each chromosome, which can be `AUTOSOME`, `CHROMOSOMEX`, `CHROMOSOMEY`, or `CUSTOMIZED`	`chromTypes=`
		`[AUTOSOME]*22`
		`+ [CHROMOSOMEX, CHROMOSOMEY]`

`lociPos=[]`	Positions of all loci on all chromosomes, as a list of float numbers. Default to 1, 2, ..., and so on on each chromosome	`lociPos=range(100)`
`ancGen=0`	Number of the most recent ancestral generations to keep during evolution	`ancGen=2`
`chromNames=[]`	A list of chromosome names Default to `''` for all chromosomes	`chromNames=['ch1', 'ch2']`

(*Continued*)

TABLE A.1 (*Continued*)

Parameter with Default Value	Usage	Examples
`alleleNames=[]`	A list or a nested list of allele names. If a list of alleles is given, it will be used for all loci in this population. Otherwise, it should specify allele names for all loci.	`alleleNames=['A', 'C', 'G', 'T']` `alleleNames=[['A', 'T'],['C', 'G']]`
`lociNames=[]`	A list of names for each locus.	`lociNames=['a','b']`
`subPopNames=[]`	A list of subpopulation names. All subpopulations will have name '' if this parameter is not specified.	`subPopNames=['CEU', 'YRI']`
`infoFields=[]`	Names of information fields (named float number) that will be attached to each individual.	`infoFields='fitness'` `infoFields=[ 'a', 'b']`

Using these parameters, small or large populations with different structures can be created. For example,

```
pop = sim.Population(size=1000, ploidy=1, loci=2)
```

creates a haploid population of 1000 individuals, each of them having 1 chromosome with 2 loci.

```
pop = sim.Population(size=[1000]*5, loci=[10]*5, infoFields='fitness')
```

creates a diploid population with 5 subpopulations, each with 1000 individuals. These individuals have 5 chromosomes, each with 10 loci and an information field named `fitness`. Note that simuPOP uses a naming convention such that plural forms of parameter names are used (e.g., `infoFields`, `loci`, `rates`) if they accept multiple inputs, although both

single and list formats of inputs are acceptable by these parameters (e.g., `loci=5`, `loci=[20, 40]`).

An independent copy of a population object can be created using its `clone()` member function. This function is very useful because assignment in Python creates only a new reference to an existing object. For example, if `pop` is a `Population` object,

```
pop1 = pop
pop2 = pop.clone()
```

create a new reference of `pop` as `pop1` and a new population object `pop2`. Modifying object `pop` will modify `pop1`, but not `pop2`.

A.2.2 Genotype Structure of a Population

The genotype structure of a population includes the number of homologous copies of chromosomes, the chromosome types and names, the number of loci on each chromosome, the position and name of each locus, and the information fields attached to each individual. A number of member functions are provided to retrieve such information from a `Population` object, along with some utility functions to, for example, look up the index of a locus by its name (e.g., `locusByName()`). Table A.2 lists some of the frequently used functions.

Source code A.2 demonstrates how to create a population and use its member functions to access its structural and genotypic information. This Source code creates a diploid population with two chromosomes. Loci names are specified so that correct loci can be identified by their names even if some other loci are inserted or removed during evolution (and lead to change of loci indices). It is worth noting that following the convention of the Python programming language, *all indices in simuPOP start from 0* and *end points are not part of ranges*. That is to say, loci are indexed as `0`, `1`, `2`, ..., `n-1` if there are `n` loci, and `range(chromBegin(1), chromEnd(1))` can be used to iterate through loci on chromosome 1 (the second chromosome) because `chromEnd(1)` returns the index of the last locus on chromosome 1 plus 1.

SOURCE CODE A.2 Access Genotype Structure of a Population

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[20, 30], loci=[10, 20], lociPos=list(range(20, 50)),
...     lociNames=['loc1_%d' % x for x in range(1, 11)] +
...         ['loc2_%d' % x for x in range(1, 21)],
...     alleleNames=['A', 'C', 'G', 'T'], infoFields='a')
>>> pop.ploidy()
```

TABLE A.2 Genotype Structure-Related Member Functions

Function	Usage
`ploidy()`	Returns the number of homologous sets of chromosomes
`numChrom()`	Returns the number of chromosomes
`numLoci(chrom)`	Returns the number of loci on a chromosome
`totNumLoci()`	Returns the total number of loci on all chromosomes
`absLocusIndex(chrom, locus)`	Returns the absolute index of the `locus` locus on chromosome `chrom`
`chromLocusPair(locus)`	Returns the index of a locus on a chromosome from its absolute index
`chromBegin(chrom)`	Returns the index of the first locus on a chromosome
`chromEnd(chrom)`	Returns the index of the last locus on a chromosome plus 1
`lociNames()`	Returns the names of all loci
`locusPos(locus)`	Returns the position of a locus
`locusName(loc)`	Returns the name of a locus
`locusByName(name)`	Looks up the index of a locus from its name
`lociByNames(names)`	Looks up indices of loci from their names
`alleleName(allele, locus=0)`	Returns the name of an allele at a locus
`infoField(idx)`	Returns the name of an information field
`infoFields()`	Returns the names of all information fields
`infoIdx(name)`	Returns the index of an information field

```
2
>>> pop.numChrom()
2
>>> pop.numLoci(1)
20
>>> pop.chromBegin(1)
10
>>> pop.chromEnd(1)
```

```
30
>>> pop.totNumLoci()
30
>>> pop.chromLocusPair(22) # chromosome number and relative index
(1, 12)
>>> pop.locusName(22)
'loc2_13'
>>> pop.lociByNames(['loc1_4', 'loc2_2'])
(3, 11)
>>> pop.alleleName(1)
'C'
>>> pop.locusPos(15)
35.0
>>> pop.infoFields()
('a',)
>>> pop.infoField(0)
'a'
```

A.2.3 Subpopulations and Virtual Subpopulations

A simuPOP population can have several subpopulations. After a population is created, you can check its structural information using functions `popSize()`, `numSubPop()`, and `subPopSize(sp)`, which return the total population size, number of subpopulations, and size of a particular subpopulation, respectively. A simuPOP subpopulation is usually anonymous but a name can be assigned to a subpopulation so that it can be identified after its index has been changed due to the merge and split of other subpopulations (see Table A.3).

TABLE A.3 Population Structure-Related Member Functions

Function	Usage
`popSize()`	Return the size of a population
`numSubPop()`	Return the number of subpopulations
`subPopIndPair(idx)`	Return the subpopulation ID and relative index of an individual, given its absolute index
`subPopSize(sp)`	Return the size of a subpopulation
`subPopName(sp)`	Return the name of a subpopulation
`subPopByName(name)`	Return the index of a subpopulation from its name
`setVirtualSplitter(splitter)`	Set a virtual splitter to define virtual subpopulations
`numVirtualSubPop()`	Return the number of virtual subpopulations

Individuals in a simuPOP subpopulation can be further grouped into virtual subpopulations according to their properties. For example, all male individuals, all unaffected individuals, all individuals with information field age greater than 20, and all individuals with genotype `(0, 0)` at a given locus can form VSPs. VSPs do not have to add up to the whole subpopulation, nor do they have to be nonoverlapping. Unlike subpopulations that have strict boundaries, VSPs change easily with the change of individual properties.

VSPs are defined by *splitters*, which are simply *definition* of VSPs. A splitter defines the same number of VSPs in all subpopulations, although sizes of these VSPs may vary across subpopulations due to individual differences. For example, a `SexSplitter()` defines two VSPs, the first with all male individuals and the second with all female individuals; an `InfoSplitter(field='x', values=[1, 2, 4])` defines three VSPs whose members have values `1`, `2`, and `4` at information field `x`, respectively. These VSPs have their own names (e.g., `'Male'` and `'Female'` for two VSPs defined by a `SexSplitter`) that describe the definition by which they are defined. Several splitters are provided in simuPOP and more complex VSPs can be defined as unions or intersections of existing VSPs.

A VSP is represented by a `(sp, vsp)` pair where `sp` and `vsp` are index or name of the subpopulation and the VSP within this subpopulation. A VSP can be used in most places where a subpopulation is needed. For example, function `subPopSize([0, 'Male'])` can be used to count the number of male individuals in a subpopulation if a `SexSplitter()` is used to define VSPs by individual sex. Source code A.3 demonstrates how to apply virtual splitters to a population, how to check VSP names and sizes, and how to apply different operations to individuals in different VSPs. This Source code uses the function form of operators `InitSex`, `InitInfo`, `InitGenotype`, and `Dumper`, which will be described in detail later. Note that parameter `subPop` accepts a single subpopulation or VSP ID, and parameter `subPops` accepts a list of subpopulation or VSP IDs. Because `subPops=[0,1]` refers to two subpopulations 0 and 1, a single VSP should be specified as `subPops=[(0,1)]`.

SOURCE CODE A.3 Define and Use Virtual Subpopulations

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=[200, 400], loci=5, infoFields='x')
>>> sim.initSex(pop)
>>> # assign random numbers using operator InitInfo
>>> sim.initInfo(pop, lambda: random.randint(0, 3), infoFields='x')
```

```
>>> # define a virtual splitter by information field 'x'
>>> pop.setVirtualSplitter(sim.InfoSplitter(field='x', values=[0, 1, 2, 3]))
>>> pop.numVirtualSubPop()     # Number of defined VSPs
4
>>> pop.subPopName([0, 0])     # Each VSP has a name
'x = 0'
>>> pop.subPopSize([0, 0])     # Size of VSP 0 in subpopulation 0
62
>>> pop.subPopSize([1, 0])     # Size of VSP 1 in subpopulation 1
110
>>> # define a virtual splitter by sex
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.numVirtualSubPop()     # Number of defined VSPs
2
>>> pop.subPopName([0, 0])     # Each VSP has a name
'Male'
>>> pop.subPopSize([0, 1])     # Size of VSP 0 in subpopulation 0
109
>>> # initialize male and females with different genotypes.
>>> sim.initGenotype(pop, genotype=[0]*5, subPops=[(0, 0)])
>>> sim.initGenotype(pop, genotype=[1]*5, subPops=[(0, 1)])
>>> sim.dump(pop, max=6, subPops=0, structure=False)
SubPopulation 0 (), 200 Individuals:
   0: FU 11111 | 11111 |  1
   1: FU 11111 | 11111 |  0
   2: MU 00000 | 00000 |  3
   3: MU 00000 | 00000 |  1
   4: MU 00000 | 00000 |  1
   5: MU 00000 | 00000 |  0
```

A.2.4 Accessing Individuals in a Population

Individuals are building blocks of a population. An individual object cannot be created independently from a population, but references to individuals can be retrieved using member functions of a population object. More specifically,

- `pop.individual(idx)` returns a reference to the idx-th individual in a population object `pop`.
- `pop.individuals()` returns an iterator that iterates through all individuals in this population (e.g., "`for ind in pop.individuals()`").
- `pop.individuals(subPop)` returns an iterator that iterates through individuals within a (virtual) subpopulation.

The individual objects returned by these functions are instances of the `Individual` class. They have access to all genotypic structure-related functions listed in Table A.2 (technically speaking, both the `Individual` and `Population` classes are derived from the `GenoStruTrait` class,

TABLE A.4 Member Functions of the `Individual` Class

Function	Usage
`sex()`	Return the name of a subpopulation
`setSex(sex)`	Return the index of a subpopulation from its name
`affected()`	Set a virtual splitter to define virtual subpopulations
`setAffected()`	
`allele(idx, ploidy=-1, chrom=-1)`	Return the size of a population
`setAllele(allele, idx, ploidy=-1, chrom=-1)`	Return the number of subpopulations
`genotype(ploidy=ALL_AVAIL, chroms=ALL_AVAIL)`	Return the subpopulation ID and relative index of an individual, given its absolute index
`setGenotype(geno, ploidy=ALL_AVAIL, chroms=ALL_AVAIL)`	Return the size of subpopulation
`info(field)`	Return the number of virtual subpopulations.
`setInfo(value, field)`	Remove specified subpopulations

so they both have access to all member functions of the base class) and to some member functions to read and write individual sex, affection status, genotype, and information fields (Table A.4). For example, function `Individual.sex()` returns the sex of an individual, which can be `MALE` or `FEMALE`.

From a user's point of view, genotypes of an individual are stored sequentially and can be accessed locus by locus or in batch. The alleles are arranged by position, chromosome, and ploidy. That is to say, the first allele on the first chromosome of the first homologous set is followed by alleles at other loci on the same chromosome, then alleles on the second and later chromosomes, followed by alleles on the second homologous set of the chromosomes for a diploid individual. A consequence of this memory layout is that alleles at the same locus of a nonhaploid individual are separated by `Individual.totNumLoci()` loci.

simuPOP provides several functions to read and write individual genotypes. For example, functions `Individual.allele()` and `Individual.setAllele()` can be used to read and write single alleles; functions `Individual.genotype()` and

`Individual.setGenotype()` can be used to read and write individual genotypes in batch mode. The `setGenotype` function accepts a list of alleles, which will be reused if its length is less than the total number of required alleles. For example, you can quickly set all alleles of an individual `ind` to 1 using function `ind.setGenotype(1)`.

Individual information fields can be accessed using functions `Individual.info(field)` and `Individual.setInfo(value, field)`, or as attributes of an `Individual` object. For example, if an individual `ind` has information field `id`, you can read its value using `ind.id` and set its value using statement `ind.id=55`. Source code A.4 demonstrates how to access and modify individual sex, affection status, and information fields using these functions.

SOURCE CODE A.4 Access to Individuals in a Population

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=[4, 5], loci=4, infoFields='a')
>>> # iterate through all individuals in the first subpopulation of pop
>>> for ind in pop.individuals(0):
...     ind.setSex(sim.FEMALE)
...     ind.setAllele(1, idx=1, ploidy=1)
...     ind.a = random.randint(2, 5)
...
>>> ind = pop.individual(2)
>>> ind.sex() # The numeric value of FEMALE is printed
2
>>> ind.genotype()
[0, 0, 0, 0, 0, 1, 0, 0]
>>> ind.setGenotype([1, 0, 1])
>>> ind.genotype()
[1, 0, 1, 1, 0, 1, 1, 0]
>>> ind.a
5.0
>>> ind.a = 10
>>> ind.a
10.0
```

A.2.5 Population Variables

Each simuPOP population has a Python dictionary that can store arbitrary Python variables. These variables are usually used by various operators to set and retrieve population statistics. For example, the `Stat` operator calculates population statistics and stores the results in this dictionary. Other operators such as the `PyEval` and `TerminateIf` read from this dictionary and act upon its values.

The `Population` class provides two member functions, namely, `Population.vars()` and `Population.dvars()` to access a population dictionary. These functions return the same dictionary object, but `dvars()` returns a wrapper class so that you can access keys in this dictionary as attributes. For example, `pop.vars()['alleleFreq'][0]` is equivalent to `pop.dvars().alleleFreq[0]`.

It is important to understand that this dictionary forms a *local namespace* in which Python expressions can be evaluated. That is to say, *items in this dictionary can be treated as variables in a namespace and be used to execute Python statements and expression*s. This is the basis of how expression-based operators work. For example, the `PyEval` operator in Source code A.1 evaluates an Python expression

```
'%d: %.2f\n' % (gen, LD[0][1])
```

in a population's local namespace when it is applied to that population. This expression uses two variables `gen` and `LD`. Variable `gen` is created and maintained automatically during the evolution of a population. It records the current generation of the population. Variable `LD` is set by operator `Stat` when it is applied to the population before operator `PyEval` is applied. Because arbitrary Python expressions can be evaluated, a `PyEval` operator can output these statistics or their derived values in any format. For example, expression

```
'1-sum([x*x for x in alleleFreq[0].values()])'
```

calculates the expected heterozygosity ($H = 1 - \sum p_i^2$ where p_i is the allele frequency of allele i) at locus 0 after a dictionary of allele frequencies `alleleFreq` is calculated by operator `Stat(alleleFreq=0)`.

A.2.6 Altering the Structure, Genotype, or Information Fields of a Population

The `Population` class provides a number of member functions to alter the structure of a population and to access individual genotype and information fields in batch mode. Table A.5 lists some of the frequently used functions. Complete prototypes are ignored because some of them accept many parameters.

Function `removeIndividuals` removes individuals from a population according to their indices, IDs (value at an information field), or the return values of a filter function that accepts each individual as its input value.

TABLE A.5 Population Modification Functions of the `Population` Class

Function	Usage
`removeSubPops`	Remove specified subpopulations
`mergeSubPops`	Merge specified subpopulations into one subpopulation
`splitSubPop`	Split a subpopulation into subpopulations with given sizes
`resize`	Resize a population with new subpopulation sizes
`removeIndividuals`	Remove individuals by indices, IDs, or a filter function
`addIndFrom`	Add individuals with the same genotype structure from another population
`addChrom`	Add a chromosome to the current population
`addChromFrom`	Add chromosomes from another population with the same number of individuals
`addLoci`	Add some loci to the current population
`addLociFrom`	Add loci from another population with the same number of individuals
`removeLoci`	Remove selected loci from a population
`genotype`	Return a list-like object that represents genotypes of all individuals in a (sub)population
`setGenotype`	Set genotype of all individuals in a (sub)population using a lists of alleles
`addInfoFields`	Add additional infomation fields to a population
`removeInfoFields`	Remove specified information fields of a population
`indInfo`	Return values of an information field of all individuals or individuals in a (virtual) subpopulation
`setIndInfo`	Set values of an information field of all individuals or individuals in a (virtual) subpopulation using a list of values

Alternatively, function `removeSubPops` can be used to remove subpopulations or groups of individuals who share the same properties (use VSP). Functions `extractIndividuals` and `extractSubPops` work similarly. However, instead of removing selected individuals or (virtual) subpopulations, they copy these individuals and form a new population from them. These functions are usually used to draw samples from an existing population.

Functions `mergeSubPops` and `splitSubPop` merge and split existing subpopulations and are usually used to merge or split parental

populations during the simulation of demographic models with population merge and split. If population size needs to be changed, the function `resize` can resize a population and the function `addIndFrom` can merge individuals from another population to the current population.

When simuPOP is used to process real empirical data sets, functions such as `removeLoci`, `addLociFrom`, and `addChromFrom` can be used to remove loci or combine data sets with different loci. We will not go into the details of these functions because they are used primarily for data processing, which is not the focus of this book. Refer to scripts such as `loadHapMap3.py` (a script to import the HapMap data sets and save them in simuPOP formats) in the simuPOP online cookbook for examples on how to use these functions.

Individuals in a population share the same set of information fields, so the addition and removal of information fields can be performed only at the population level. Because the unused information fields tend to hinder the efficiency of simulations, it is a common practice to use minimal set of information fields during evolution and add additional information fields using function `addInfoFields` for postevolution data analysis.

Finally, the `Population` class provides functions to access individual genotype and information fields in batch mode. For example, function `indInfo` returns values of an information field of all individuals or individuals belonging to a (virtual) subpopulation. This makes it easy to calculate summary statistics of these information fields. Similarly, function `genotype` returns genotypes of all or individuals in certain (virtual) subpopulation so that you can count, for example, the number of mutants in a population. For performance considerations, function `genotype` returns a special object that directly exposes the underlying genotypes to users. Modifying this object will change individual genotypes.

Source code A.5 demonstrates how to use some of the mentioned functions.

SOURCE CODE A.5 Use of Population Modification and Batch Access Functions

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=[4, 6], loci=2, infoFields='x')
>>> pop.setIndInfo([random.randint(0, 10) for x in range(10)], 'x')
>>> sum(pop.indInfo('x')) / pop.popSize()  # get the mean of field x
6.0
>>> pop.setGenotype([0, 1, 2, 3], 0)
>>> pop.genotype(0)  # for the first subpopulation
[0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
>>> pop.setVirtualSplitter(sim.InfoSplitter(cutoff=[3], field='x'))
>>> pop.mergeSubPops(subPops=[0,1]) # merge two subpopulations
```

```
0
>>> pop.setGenotype([0])    # clear all values
>>> pop.setGenotype([5, 6, 7], [0, 1])  # only for x >= 3
>>> pop.indInfo('x', 0)
(8.0, 9.0, 10.0, 8.0, 10.0, 5.0, 0.0, 1.0, 2.0, 7.0)
>>> pop.removeSubPops([(0,0)])          # remove individuals with x < 3
>>> pop.popSize()
7
>>> pop.genotype(0) # so all existing individuals have genotype 5, 6, 7
[5, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6, 7, 5]
```

A.2.7 Multigeneration Populations and Parental Information

All simulations we have described so far discard parental information. That is to say, a parental population is discarded when it is replaced by its offspring population at the end of an evolutionary cycle. This behavior can be changed by setting the *ancestral depth* of a population, namely, how many ancestral generations to keep during an evolutionary process. For example, a population

```
Population(10000, loci=5, ancGen=2)
```

created in Source code A.6 will keep its parental (`ancGen=1`) and grandparental (`ancGen=2`) generations during an evolutionary process. At the end of each generation, the existing grandparental generation is dicarded, the parental generation becomes the grandparental generation, the present population becomes the parental generation, and the offspring population becomes the present generation. After 20 generations, the population object will have 3 generations, namely, the offspring population at the end of generation 17, 18, and 19. These generations could be set as the present population using function `Population.useAncestralGen(gen)` where `gen` is `0` for present, `1` for parental, and `2` for grandparental generation.

SOURCE CODE A.6 Keeping Multiple Ancestral Generations During an Evolutionary Process

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=10000, loci=5, ancGen=2)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.7, 0.3]),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.MaPenetrance(loci=2, penetrance=[0.05, 0.1, 0.12]),
...     ],
```

```
...     gen=20
... )
20
>>> for gen in range(pop.ancestralGens() + 1):
...     pop.useAncestralGen(gen)
...     sim.stat(pop, numOfAffected=True)
...     print('Number of affected individuals in generation %d is %d' % \
...         (gen, pop.dvars().numOfAffected))
...
Number of affected individuals in generation 0 is 760
Number of affected individuals in generation 1 is 773
Number of affected individuals in generation 2 is 787
```

Although the multigeneration population created in Source code A.6 keeps two ancestral generations, it does not keep any parental information, so it is not possible to identify parents of each individual. In order to keep parental information, it is necessary to assign unique IDs to all individuals and store IDs of parents to their offspring. simuPOP reserves information fields `ind_id`, `father_id` and `mother_id` and operators `IdTagger` and `PedigreeTagger` for such purposes. More specifically, in order to track parentship of a population, you should add information fields `ind_id`, `father_id`, and `mother_id` to the population and use operator `IdTagger` to assign a unique ID to field `ind_id` of each individual and an operator `PedigreeTagger` to record the ID of the father and mother of each offspring to fields `father_id`, and `mother_id` respectively.

Source code A.7 demonstrates how to assign IDs and record parentship of each individual during evolution. In this Source code, an operator `IdTagger` is used to initialize all individuals in the starting population with IDs 1, 2, ..., 1000. During evolution, operator `IdTagger` assigns a unique ID to each offspring and operator `PedigreeTagger` copies `ind_id` of his or her parents to fields `father_id` and `mother_id`. If we examine the value of these information fields of the simulated population, we can see that individuals have unique IDs, and two individuals share the same parents because each mating event produces two offspring (`numOffspring=2`).

SOURCE CODE A.7 Recording Parentship of Individuals During Evolution

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=1000, loci=5, ancGen=2,
...     infoFields=['ind_id', 'father_id', 'mother_id'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.7, 0.3]),
...         sim.IdTagger(),
```

```
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.IdTagger(),
...         sim.PedigreeTagger()],
...         numOffspring=2),
...     postOps=[
...         sim.MaPenetrance(loci=2, penetrance=[0.05, 0.1, 0.12]),
...     ],
...     gen=20
... )
20
>>> pop.indInfo('ind_id')[:5]
(20001.0, 20002.0, 20003.0, 20004.0, 20005.0)
>>> pop.indInfo('father_id')[:5]
(19443.0, 19443.0, 19838.0, 19838.0, 19755.0)
>>> pop.indInfo('mother_id')[:5]
(19181.0, 19181.0, 19854.0, 19854.0, 19756.0)
```

A.2.8 Saving and Loading a Population

A population can be saved to a disk file using function `Population.save(filename)`, and be loaded from this file using global function `loadPopulation(filename)`. simuPOP uses a binary format to save a population object. The format is not human readable, but is portable in the sense that a file saved in one platform can be loaded by simuPOP on another platform. The simuPOP cookbook provides a number of functions to save and load simuPOP populations in formats used by other genetic analysis software.

A.3 OPERATORS

Operators are objects that act on populations. During an evolutionary process, operators are applied to populations repeatedly, just like what robots do in an automotive production line. There are two types of operators: operators that are applied to populations before or after mating, and operators that are applied to offspring during mating. Some operators could be applied to both populations and individuals to perform different tasks.

Operators that are applied to populations are used in parameters `initOps`, `preOps`, `postOps`, and `finalOps` of the `Population.evolve()` function. The `initOps` operators are applied before an evolutionary process, the `preOps` operators are applied to the parental population at each generation before mating, the `postOps` operators are applied to the offspring population at each generation after mating, and the `finalOps` operators are applied after an evolutionary process. These

operators include fitness operators that set individual fitness values before mating, mutation operators that mutate alleles, statistics calculators that calculate population statistics, and operators that report the progress of an evolutionary process.

Operators that are applied to individuals are used in the `ops` parameter of a mating scheme. They are usually used to transmit genotype or other information from parents to offspring. Examples of such operators include `MendelianGenoTransmitter` that transmits parental genotype to offspring according to Mendelian laws and `PedigreeTagger` that records the IDs of parents to each offspring.

A.3.1 Applicable Generations

Operators are, by default, applied to all generations during an evolutionary process. This can be changed using the `begin`, `end`, `step`, and `at` parameters of operators. As their names indicate, these parameters control the starting generation (`begin`), ending generation (`end`), generations between two applicable generations (`step`), and an explicit list of applicable generations (`at`, a single generation number is also acceptable). Other parameters will be ignored if parameter `at` is specified. If the number of generations to evolve is fixed (parameter `gen` of the `Population.evolve` function is specified), negative generation numbers are allowed. They are counted backward from the ending generation. For example, if a simulation starts at generation `0`, and the `evolve` function has parameter `gen=10`, the simulator will stop at the *beginning* of generation `10`. Generation `-1` refers to generation `9` (the last generation), and generation `-2` refers to generation `8`, and so on.

These parameters give simuPOP the flexibility to apply operators at selected generations. For example, you can calculate and output statistics at every 10 generations to avoid excessive report or apply different migration models during different stages of an evolutionary process. Source code A.8 demonstrates how to set applicable generations of an operator. In this Source code, operators `InitSex` ad `InitGenotype` are applied once before evolution. Operators `Stat` and `PyEval` are applied at every 10 generations, namely, generations 0, 10, 20, ... At these generations, operator `Stat` is applied before mating (on the parental population) to calculate allele frequency at locus 0, and `PyEval` is applied after mating (on the offspring population) to output variable `alleleFreq[0][1]` where `0` and `1` are locus and allele indices respectively. Although `PyEval` is applied after mating, the allele frequencies it reports are actually frequencies of the parental population calculated by operator `Stat` before mating.

An operator `MapSelector` is used to select against allele 1 starting from generation 50, so the frequency of allele 1 decreases quickly after that generation.

SOURCE CODE A.8 Applicable Generations of an Operator

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=2000, loci=1, infoFields='fitness')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...     ],
...     preOps=[
...         sim.Stat(alleleFreq=0, step=10),
...         sim.MapSelector(begin=50, loci=0,
...             fitness={(0,0):1, (0,1):0.99, (1,1):0.97})
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=sim.PyEval(r"'%3d: %.2f\n' % (gen, alleleFreq[0][1])",
...         step=10),
...     gen=100
... )
  0: 0.50
 10: 0.52
 20: 0.52
 30: 0.50
 40: 0.48
 50: 0.48
 60: 0.46
 70: 0.46
 80: 0.44
 90: 0.37
100
>>>
```

A.3.2 Operator Output

All operators we have seen write their output to the standard output, namely, a terminal window. However, in a complex evolutionary system where multiple statistics are recorded, you might want to store certain statistics in one file and others in another file. In these cases, you can use parameter `output` of operators to direct their output to other destinations.

Parameter `output` accepts an output specification string or a user-defined Python function (Table A.6). Because statistics are usually collected and outputted repeatedly during evolution, the most frequently used format is `'>>filename'`. Files specified in this way will be opened before evolution, accept inputs from one or more operators during evolution, and be closed afterward. A comma or tab separated file can be created in this way if proper delimiters are outputted.

TABLE A.6 Acceptable Inputs for Parameter `Output` of Operators

Parameter	Usage
`''`	Supress output
`'filename'`	Write output to a file named `filename` and close it Existing content of this file will be cleared
`'>filename'`	Equivalent to `'filename'`
`'>>filename'`	Append output to a file named `filename`. Clear the file before evolution if this file already exists
`'>>>filename'`	Append output to a file named `filename`. Do not clear the file before evolution if this file already exists
`'!expr'`	Obtain an output specification string by evaluating expression `expr` in the local namespace of the current population
`a Python function`	Send the output to a user-defined Python function

A output specification will be considered as an expression if it starts from an exclamation symbol. Such an expression will be evaluated in a population's local namespace to determine a proper output. For example, parameter `output='!"gen_%d.txt" % gen'` directs output from an operator to files `gen_0.txt`, `gen_1.txt` etc at generations `0`, `1`, ... As an advanced feature, operator output could be sent directly to a Python function for real time analysis.

Source code A.9 demonstrates how to use the `output` parameters to record two LD measures to files `LD.txt` and `R2.txt` separately.

SOURCE CODE A.9 Use of Parameter Output of Operators to Redirect Operator Output

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=2000, loci=2)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[1, 2, 2, 1])
...     ],
...     matingScheme=sim.RandomMating(ops=sim.Recombinator(rates=0.1)),
...     postOps=[
...         sim.Stat(LD=[0, 1]),
...         sim.PyEval(r"'%3d: %.4f\n' % (gen, LD[0][1])",
...             output='>>LD.txt'),
...         sim.PyEval(r"'%3d: %.4f\n' % (gen, R2[0][1])",
...             output='>>R2.txt')
```

```
...     ],
...     gen=100
... )
100
>>> # print the first five lines of the output
>>> print(''.join(open('R2.txt').readlines()[:5]))
  0: 0.6123
  1: 0.5055
  2: 0.4075
  3: 0.3513
  4: 0.2808
```

A.3.3 During-Mating Operators

Source code A.1 uses operator `Recombinator(rate=0.01)` in the `ops` parameter of mating scheme `RandomMating`. This operator retrieves parental chromosomes, recombines them with specified recombination rate, and passes one recombinant from each parent to the offspring. The random mating scheme in Source code A.8 does not specify an `ops` parameter so that a default during- mating operator for this mating scheme, namely, a `MendelianGenoTransmitter()`, is used to transmit genotype from parents to offspring according to Mendelian laws. These during-mating operators are called *genotype transmitters* just to indicate they are responsible for transmitting parental genotypes to offspring.

In addition to genotype transmitters, other during-mating operators could be applied during the production of offspring. For example, operator `PedigreeTagger` records the IDs of parents to information fields of offspring, and operator `InheritTagger` passes parental information fields to offspring. It is important to remember that a genotype transmitter needs to be explicitly specified when the `ops` parameter is used.

Source code A.10 uses an operator `InheritTagger` to record the ancestry of each individual. The simulation starts from a population of 1000 individuals, half with ancestry value 0 and half with ancestry value 1. During evolution, a `MendeliangenoTransmitter` passes parental genotypes and a `InheritTagger` passes the mean of ancestral ancestry values to their offspring. Because a random mating scheme selects parents randomly, it is not surprising that the majority of the individuals have an ancestry value around 0.5 after only a few generations. This Source code uses operator `InitInfo` to assign a sequence of values (`[0,1]`) to individuals in a population, and a population member function `Population.indInfo` to get the values of information field `anc` of all individuals.

SOURCE CODE A.10 Use of an InheritTagger to Track Individual Ancestry

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=1000, loci=20, infoFields='anc')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         sim.InitInfo([0,1], infoFields='anc'),
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.InheritTagger(mode=sim.MEAN, infoFields='anc')]),
...     gen=10
... )
10
>>> # find the ancestral values
>>> anc = pop.indInfo('anc')
>>> print('min anc: %.3f, mac anc: %.3f' % (min(anc), max(anc)))
min anc: 0.432, mac anc: 0.558
```

A.3.4 Function Form of Operators

Operators are usually applied to populations during evolution, although they can also be applied to a population directly, using their function counterparts. These functions are named similar to the corresponding classes. They take a population object as their first parameter, create an operator using the rest of the parameters, and apply the operator to the passed population. For example, operators used in the `initOps` parameter of Source code A.10 can be moved before function `pop.evolve` to initialize the population as follows:

```
pop = sim.Population(size=1000, loci=20, infoFields='anc')
sim.initSex(pop)
sim.initGenotype(pop, freq=[0.5, 0.5])
sim.initInfo(pop, [0, 1], infoFields='anc')
pop.evolve( ... )  # ignored
```

Function `stat` is the function form of operator `Stat`, which is frequently used to calculate statistics of populations. For example, it is more efficient to calculate the minimum and maximum of ancestry values of the simulated population in Source code A.10 using

```
stat(pop, minOfInfo='anc', maxOfInfo='anc')
print('min anc: %.3f, max anc: %.3f' % \
    (pop.dvars().minOfInfo['anc'], (pop.dvars().minOfInfo['anc']))
```

because the `Stat` operator calculates the statistics internally without copying values of information field `'anc'` to a list.

A.3.5 Operator `Stat`

Operator `Stat` is used to calculate statistics for populations, subpopulations, or groups of individuals in subpopulations that share certain properties (virtual subpopulations). Instead of returning calculated statistics in some way, this operator stores one or more variables in a population's local namespace. These variables can be accessed by other operators such as `PyEval` or directly from a population object using member function `Population.vars()` or `Population.dvars()`.

Table A.7 lists acceptable parameters of the `Stat` operator and the variables it sets for each statistics. Variables marked by an asterisk (∗) are the default variables that will be set for a statistic. An alternative set of variables can be specified via parameter `vars` of this operator. For example, operator

```
Stat(association=ALL_AVAIL, vars='Armitage_p')
```

performs Cochran–Armitage trend tests at all available loci of a population and sets a dictionary `Armitage_p` for the *p*-values of the tests. The default allele-based association tests will not be performed in this case.

This operator by default does not calculate statistics for subpopulation. If you list a set of (virtual) subpopulations using parameter `subPops`, individuals from these subpopulations will be pulled together for calculation. For example,

```
pop.setVirtualSplitter(SexSplitter())
stat(pop, alleleFreq=0, subPops=[(ALL_AVAIL, 'Male')])
```

calculates allele frequency at locus 0 for all male individuals in all subpopulations of population `pop`. The results will be saved to variables `alleleFreq` and `alleleNum`, although they are the allele frequency among all male individuals instead of the allele frequency for all individuals in a population. This example uses `[(ALL_AVAIL, 'Male')]` to represent the first virtual subpopulations in all available subpopulations. Similarly, `[(0, ALL_AVAIL)]` or `[(ALL_AVAIL, ALL_AVAIL)]` can be used to refer to all virtual subpopulations in a specific subpopulation or in all subpopulations.

Subpopulation-specific statistics can be calculated for variables marked by a *s* symbol in Table A.7, by specifying variables with a `_sp` suffix in parameter `vars` of the `Stat` operator. The resulting variables will be stored in dictionaries `subPop[sp]` where `sp` is the ID of (virtual) subpopulations. For example, operator

TABLE A.7 Parameters and Variables of Operator `Stat`

Statistics	Parameter	Variables
Population size	`popSize=False`	`popSize`^{*si}, `subPopSize`^{l}
Number of male individuals	`numOfMales=False`	`numOfMales`^{*si}, `numOfFemales`^{*si}, `propOfMales`^{sf}, `propOfFemales`^{sf}
Number of affected individuals	`numOfAffected=False`	`numOfAffected`^{*si}, `numOfUnaffected`^{*si}, `propOfAffected`^{sf}, `propOfUnaffected`^{sf}
Allele count and frequency	`alleleFreq=[]`	`alleleFreq`^{*sd}, `alleleNum`^{*sd}
Heterzygote frequency	`heteroFreq=[]`	`heteroFreq`^{*sd}, `heteroNum`^{sd}
Homozygote frequency	`homeFreq=[]`	`homoFreq`^{*sd}, `homoNum`^{sd}
Genotype frequency	`genoFreq=[]`	`genoFreq`^{*sd}, `genoNum`^{*sd}
Haplotype frequency	`haploFreq=[]`	`haploFreq`^{*sd}, `haploNum`^{*sd}
Sum of information fields	`sumOfInfo=[]`	`sumOfInfo`^{*sd}
Mean of information fields	`meanOfInfo=[]`	`meanOfInfo`^{*sd}
Variance of information fields	`varOfInfo=[]`	`varOfInfo`^{*sd}
Maximum of information fields	`maxOfInfo=[]`	`maxOfInfo`^{*sd}
Minimum of information fields	`minOfInfo=[]`	`minOfInfo`^{*sd}
Linkage disequilibrium	`LD=[]`	`LD`^{*sd}, `LD_prime`^{*sd}, `R2`^{*sd}, `LD_ChiSq`^{sd}, `LD_ChiSq_p`^{sd}, `CramerV`^{sd}
Association tests	`association=[]`	`Allele_ChiSq_p`^{*sd}, `Allele_ChiSq`^{sd}, `Geno_ChiSq`^{sd}, `Geno_ChiSq_p`^{sd}, `Armitage_p`^{sd}
Neutrality tests	`neutrality=[]`	`Pi`^{*sf}
Population structure	`structure=[]`	`F_st`^{*f}, `F_is`^{f}, `F_it`^{f}, `f_st`^{d}, `f_is`^{d}, `f_it`^{d}, `G_st`^{f}, `g_st`^{d}
Hardy–Weinberg equilibrium	`HWE=[]`	`HWE`^{*sd}

$*$: Default variable, s: available for (virtual) subpopulations if `varname_sp` is specified. i: integer variable, f: float variable, d: dictionary variable.

```
Stat(alleleFreq=0, vars='alleleFreq_sp')
```

calculates allele frequencies at locus 0 for all subpopulations and sets variables such as `subPop[0]['alleleFreq']`, whereas

```
Stat(alleleFreq=0, subPops=[(ALL_AVAIL, 0)], vars='alleleFreq_sp')
```

calculates allele frequencies for specified virtual subpopulations and sets variables such as `subPop[(1,0)]['alleleFreq']`. Note that `pop.dvars(sp).alleleFreq` can be used as a shortcut to access variable `pop.dvars().subPop[sp]['alleleFreq']`.

Variables set by the Stat operator can be an integer (marked by a *i* symbol in Table A.7), a float number (*f*), a list of numbers (*l*), and a dictionary (*d*). The keys of these dictionaries vary from statistic to statistic. For example, variable `meanOfInfo['fitness']` records the mean of information fitness; variable `alleleFreq[0][a]` records the frequency of allele `a` at locus `0` using alleles as keys; and variable `haploNum[(0,1,2)][(0,0,1)]` records the frequency of haplotype `-0-0-1-` at loci `0`, `1`, and `2`, using a tuple of alleles as keys. Because keys of these dictionaries sometimes cannot be determined in advance, access to these dictionaries with an invalid key will return 0 instead of triggering an `KeyError` exception. For example, if there are alleles 0, 2, 3 at locus 1, operator

```
Stat(alleleFreq=1)
```

will set dictionaries `alleleFreq[1]` and `alleleNum[1]` with keys `0`, `2`, and `3`. You can use expression `len(alleleFreq[1])` to obtain the number of alleles at this locus, expression `alleleFreq[1].keys()` to obtain a list of available alleles, and most importantly, expressions such as

```
alleleFreq[1][1]
```

to output or display frequency of a particular allele without worrying about whether or not there is such an allele at this locus.

A.3.6 Hybrid and Python Operators

Given the large number of population genetics models and statistics, it is not possible for simuPOP to provide native support for all of them. Fortunately, owing to the scripting language design, it is easy to extend

functions of simuPOP in Python through the use of user-provided *callback functions*.

A simuPOP *callback function* is a user-defined Python function that is passed to and called by simuPOP. The interface of this function is specified by simuPOP. For example, a demographic function is a callback function that is passed to the `subPopSize` parameter of a mating scheme and is called by the mating scheme at each generation before mating happens. This function accepts a generation number with parameter `gen` and/or a parental population with parameter `pop` and returns the population size (if there is no population structure) or a list of subpopulation sizes of the offspring population. It is noted that *simuPOP depends on parameter names to determine what should be passed to a callback function*. For example, a mating scheme will pass a parental population to function `demo(pop)`, a generation number to function `demo(gen)`, and both parental population and generation number to function `demo(pop, gen)`.

A *hybrid operator* is an operator that accepts a callback function (Does it accept something else, since it is called a hybrid?). The number and meaning of input parameters and return values vary from operator to operator. For example, a hybrid mutator sends a to-be-mutated allele to a callback function and uses its return value as the mutant allele. A hybrid selector uses return values of a user-defined function as individual fitness values. Such an operator handles the routine part of the work (e.g., scan through a chromosome and determine which allele needs to be mutated) and leaves the creative part to users. For example, Source code A.11 defines an asymmetric stepwise mutation model with random steps using a hybrid mutator called `PyMutator`. This mutator mutates alleles at loci 2 and 5 with specified mutation rates and sends alleles to be mutated to a callback function `randomStep`, which mutates an allele a to allele $a-1$, a, $a+1$, $a+2$ with equal probabilities. Because of the upward trend of this mutational process, the average number of tandem (why are they tandem repeats, this seems to be single repeat differences between alleles) repeats increases during evolution.

SOURCE CODE A.11 An Asymmetric Stepwise Mutation Model with Random Steps

```
>>> import simuPOP as sim
>>> import random
>>> def randomStep(allele):
...     return allele + random.randint(-1, 2)
...
>>> pop = sim.Population(size=1000, loci=[10])
```

```
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=100)
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=sim.PyMutator(func=randomStep, rates=[1e-3, 1e-2],
...             loci=[2, 5]),
...     gen = 1000
... )
1000
>>> # count the average number of tandem repeats at both loci
>>> sim.stat(pop, alleleFreq=[2, 5])
>>> sum([x*y for x,y in pop.dvars().alleleNum[2].items()])/(2.*pop.popSize())
99.859
>>> sum([x*y for x,y in list(pop.dvars().alleleNum[5].items())])/(2.*pop.popSize())
104.9455
```

A Python operator `PyOperator` is the most flexible hybrid operator in simuPOP because its callback function takes a population or an individual as its input and can perform arbitrary operations on them. When this operator is applied to a parental or offspring population, it passes the population directly to a callback function with an optional parameter (parameters `pop` and `param`). For example, operator `PyOperator` used in Source code A.12 passes offspring populations to a function `drawSample` at every 100 generations. This function draws cases and controls from these populations and saves samples for later analysis. Numbers of cases and controls are passed to this function using parameter `param`. Note that a callback function for operator `PyOperator` must return `True` or `False` and the evolution of a population will be terminated if `False` is returned.

SOURCE CODE A.12 Use of a Python Operator to Draw Sample at Every 100 Generations

```
import simuOpt
simuOpt.setOptions(quiet=True, alleleType='binary')
import simuPOP as sim
from simuPOP.sampling import drawCaseControlSample

def drawSample(pop, param):
    'Rest fixed locus to all zero alleles'
    nCase, nCtrl = param
    sample = drawCaseControlSample(pop, cases=nCase, controls=nCtrl)
    sample.save('sample_%d.pop' % pop.dvars().gen)
    return True

pop = sim.Population(size=10000, loci=1)
pop.evolve(
    initOps=[
        sim.InitSex(),
        sim.InitGenotype(freq=[0.5, 0.5]),
    ],
    matingScheme=sim.RandomMating(),
```

```
    postOps=[
        sim.MaPenetrance(loci=0, penetrance=(0.01, 0.1, 0.15), step=100),
        sim.PyOperator(func=drawSample, param=(100, 100), step=100),
    ],
    gen = 500
)
```

A.4 EVOLVING ONE OR MORE POPULATIONS

Function `Population.evolve()` evolves a population generation by generation, following a discrete-generation evolutionary model depicted in Figure A.1. This function accepts a mating scheme and several lists of operators, which are all Python objects with their own properties and member functions. Table A.8 lists the parameters that are accepted by function `Population.evolve`.

A.4.1 Mating Scheme

A mating scheme specified by parameter `matingScheme` is used to select parents from the parental population and populate an offspring population. A mating scheme is responsible for the following:

1. *Determining the Size of the Offspring Population* The offspring population has by default the same number of individuals as the parental

TABLE A.8 Acceptable Parameters of Function `Population.evolve`

Parameter	Usage
`initOps`	A list of operators that will be applied before evolution. They are usually used to initialize a population and prepare it for evolution
`preOps`	A list of operators that will be applied to the parental population at the beginning of each generation
`matingScheme`	A mating scheme that produces an offspring population from a parental population
`postOps`	A list of operators that will be applied to the offspring population at the end of each generation
`finalOps`	A list of operators that will be applied after evolution
`gen`	Generations to evolve. Default to `-1`, which will cause the evolutionary process to continue indefinitely
`dryrun`	If set to `True`, the `evolve` function will print a description of the evolutionary process and exit

population, but you can specify a fixed population size or a dynamic population size using parameter `subPopSize`. In the latter case, a Python function should be defined to return the population size of the offspring population at each generation.

2. *Determining How the Parents Are Chosen* A `RandomMating` mating scheme chooses a male and a female parent randomly, at equal probability or at a probability that is proportional to individual fitness values if an information field named `fitness` exists. It is the most frequently used mating scheme in this book.
3. *Determining the Number of Offspring Per Mating Event* This is controlled by parameter `numOffspring`, which can be a fixed number (default to 1) or a random distribution.
4. *Determining the Sex of Offspring* This can be determined randomly or may follow a fixed pattern. Parameter `sexMode` is used to control this behavior.
5. *Determining How Parental Genotypes and Other Information Are Passed to Offspring* Each mating scheme has a default genotype transmitter that can be overridden with a list of operators in the parameter `ops` of a mating scheme.

Source code A.13 demonstrates some of the features of a simuPOP mating scheme using a monogamous mating scheme. Unlike a random mating scheme in which parents can mate with several spouses, this mating scheme chooses parents without replacement, so each parent can have only one spouse. In order to ensure equal number of male and female parents, each mating event produces exactly one male and one female offspring. A `ParentsTagger` is used to track the parents of each offspring so that we can determine the parents of each offspring in the simulated population. This mating scheme can be used to simulate theoretical models in which all parents pass their genotypes to the offspring population or to simulate strictly controlled mating schemes such as animal breeding programs for endangered species.

SOURCE CODE A.13 Control of the Number and Sex of Offspring in a Monogamous Mating Scheme

```
>>> import simuPOP as sim
>>> pop = sim.Population(10, loci=10, infoFields=['father_idx', 'mother_idx'])
>>> pop.evolve(
...     # use a proportion instead of probability to ensure equal numbers of
...     # males and females
...     initOps=[
```

```
...         sim.InitSex(maleProp=0.5),
...         sim.InitGenotype(freq=[0.2, 0.8])
...     ],
...     matingScheme=sim.MonogamousMating(numOffspring=2,
...         # fix the number of male offspring, the rest are females.
...         sexMode=(sim.NUM_OF_MALES, 1),
...         ops=[sim.MendelianGenoTransmitter(),
...             sim.ParentsTagger()]  # track indexes of parents
...         ),
...     gen=10,
... )
10
>>> # number of male and female offspring?
>>> sim.stat(pop, numOfMales=True)
>>> # sex of offspring?
>>> [ind.sex() for ind in pop.individuals()]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
>>> pop.dvars().numOfMales
5
>>> # parents of the first five offspring
>>> pop.indInfo('father_idx')[:10]
(0.0, 0.0, 2.0, 2.0, 8.0, 8.0, 4.0, 4.0, 6.0, 6.0)
```

A.4.2 Conditionally Terminating an Evolutionary Process

It is not always possible to know in advance the number of generations to evolve. For example, you may want to evolve a population until a specific allele gets fixed or lost in the population. In this case, you can let the simulator run indefinitely (do not set parameter `gen` of the `evolve` function) and depend on a *terminator* to terminate the evolution of a population. The easiest method to do this is to use population variables to track the status of a population and to use a `TerminateIf` operator to terminate the evolution according to the value of an expression.

Source code A.14 demonstrates the use of such a terminator, which terminates the evolution of a population if allele 0 at locus 5 is fixed or lost. This operator uses expression `len(alleleNum[5])==1` to determine if the allele is fixed or lost because `alleleNum[5]` is a dictionary of allele numbers, and `len(alleleNum[5])==1` implies that there is only one allele left at this locus. Source code A.14 also demonstrates the application of an interesting operator `IfElse`, which applies an operator, in this case a `PyEval`, only when an expression returns `True`.

SOURCE CODE A.14 Use a Terminator to Terminate an Evolutionary Process Conditionally

```
>>> import simuPOP as sim
>>> pop = sim.Population(50, loci=[10], ploidy=1)
>>> pop.evolve(
...     initOps=sim.InitGenotype(freq=[0.5, 0.5]),
```

```
...     matingScheme=sim.RandomSelection(),
...     postOps=[
...         sim.Stat(alleleFreq=5),
...         sim.IfElse('alleleNum[5][0] == 0',
...             sim.PyEval(r"'Allele 0 is lost at generation %d\n' % gen")),
...         sim.IfElse('alleleNum[5][0] == 50',
...             sim.PyEval(r"'Allele 0 is fixed at generation %d\n' % gen")),
...         sim.TerminateIf('len(alleleNum[5]) == 1'),
...     ],
... )
Allele 0 is fixed at generation 19
20
>>> pop.dvars().gen
20
```

A.4.3 Evolving Several Populations Simultaneously

simuPOP can evolve several populations simultaneously, which allows side-by-side comparison between instances of the same evolutionary process or evolutionary processes under slightly different settings. For example, Source code A.15 evolves three replicates of the same population simultaneously, subject to different intensity of mutations. Allele frequencies of allele 0 of three replicates are printed in a tabular format. This Source code uses a `Simulator` object, which is essentially a list of populations.

This Source code starts with the creation of a population of 5000 individuals. Instead of evolving this population directly, it creates a `Simulator` object with three replicates of this population. The `evolve` function of the simulator accepts the same set of parameters as the `evolve` function a `Population` class. The only difference is that this function evolves all populations in a simulator for one generation before it moves to the next generation.

Operators are applied to all populations in a simulator unless a parameter `reps` is used to specify indices of applicable populations. In this Source code, three `SNPMutator` operators are applied to three populations using different mutation rates. In order to print the outputs in a tabular format, a `PyEval` operator outputs a generation number once (for the first replicate), while `PyEval` operator outputs allele frequency at the first locus for each population, prefixed with a `'\t'`, followed by a newline character that is outputted by a `PyOutput` operator, which is applied only to the last replicate.

The `Simulator` class provides several functions to access its populations. Function `Simulator.population(idx)` returns a reference to the `idx`-th population in a simulator. Function `Simulator.populations()` returns an iterator that iterates through all populations in a simulator. Changing the returned population references will

change populations in a simulator. If you would like an independent copy of a population, you can extract a population from a simulator using function `Simulator.extract(idx)`.

SOURCE CODE A.15 Evolve Several Replicates of a Population Simultaneously

```
>>> import simuPOP as sim
>>> pop = sim.Population(5000, loci=1)
>>> # three copies of the same population
>>> simu = sim.Simulator(pop, rep=3)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     preOps=[
...         sim.SNPMutator(u=0.01, reps=0),
...         sim.SNPMutator(u=0.02, reps=1),
...         sim.SNPMutator(u=0.05, reps=2),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.PyEval('gen', step=20, reps=0),
...         sim.PyEval(r"'\t%.3f' % alleleFreq[0][0]", step=20),
...         sim.PyOutput('\n', step=20, reps=-1),
...     ],
...     gen=100
... )
0 0.503 0.493 0.480
20 0.411 0.316 0.177
40 0.340 0.206 0.065
60 0.277 0.112 0.024
80 0.240 0.081 0.014
(100, 100, 100)
>>> # access populations in a simulator
>>>  for pop  in simu.populations():
...     print(pop.dvars().alleleFreq[0][0])
...
0.2108
0.0542
0.0072
>>> # extract a population
>>> pop = simu.extract(1)
>>> # print out allele frequency
>>> pop.dvars().alleleFreq[0][0]
0.0542
```

A.5 A COMPLETE simuPOP SCRIPT

Although different Python modules could be used to provide command line (e.g., `getopt`, `optparse` and `argparse`) or graphical (e.g.,

`Tkinter`) user interfaces to a simuPOP script, a simuPOP utility module `simuOpt` has been designed to provide a flexible user interface specifically for simulation studies. Instead of providing a fixed user interface, this module allows users to execute the same simuPOP script in batch mode (no user interaction), interactive mode (accept user input from command line), or through a graphical parameter input dialog. Source code A.16 lists a complete simuPOP script that makes use of this module.

The first line of this script is called a *shebang* line that tells a Linux/Unix system which interpreter should be used to execute a script. Because the Python executable might reside in different paths under different operating systems (e.g., `/usr/bin` or `/usr/local/bin`), a `/usr/env python` command is usually used to locate a Python interpreter and execute it.

The string surrounded by a pair of matching triple quotes in lines 2–8 is a module doc string. It describes the main purposes of the script and is accessible via variable `__doc__`. By passing this docstring to a `simuOpt.Param` object (line 78), this string will become part of the help message.

All examples we have seen use the standard simuPOP module. This module uses 8 bits to store an allele, so each locus can have 256 possible allelic states, ranging from 0 to 255. A run-time validation mechanism is used to monitor a simulation, which will terminate the execution of a script with a detailed error message when an invalid operation is detected. Other variants of the core simuPOP modules are provided for different applications. For example, a long allele version having 2^{32} (or 2^{64} for 64 bit operating systems) possible allele states can be used to simulate certain population genetics models such as an infinite allele model; a binary allele version that uses 1 bit for each allele should be used for diallelic (SNP) markers; and an optimized module that bypasses run-time validation can be used for production scripts that have been thoroughly tested.

Lines 10–12 demonstrate how to use the `simuOpt.setOptions` function to control which variant of simuPOP to load and how to load it. This Source code uses `alleleType='binary'` and `optimized=True` to load an optimized diallelic version of simuPOP. It also uses `quiet=True` to suppress a banner message when simuPOP is imported. Finally, option `'version=1.0.4'` indicates that this script is only compatible to simuPOP 1.0.4 and later. An error message will be displayed if an earlier version of simuPOP is imported.

SOURCE CODE A.16 A Sample simuPOP Script

```
#!/usr/bin/env python
'''
This script demonstrates the decay of linkage disequilibrium under the impact
of genetic recombination. The simulation starts with populations in which
two loci are under complete linkage disequilibrium. During evolution, parental
chromosomes are recombined before they are passed to their offspring, resulting
a gradule decay of linkage disequilibrium between these two loci.
'''
import sys
import simuOpt
simuOpt.setOptions(alleleType='binary', optimized=True, quiet=True, version='1.0.4')
import simuPOP as sim

options = [
    {'name': 'popSize',
     'default': 1000,
     'type': int,
     'label': 'Population Size',
     'validator': 'popSize > 0',
    },
    {'name': 'gen',
     'default': 50,
     'type': int,
     'label': 'Generations to evolve',
     'validator': 'gen > 0',
    },
    {'name': 'recRate',
     'default': 0.01,
     'type': (int, float),
     'label': 'Recombination Rate',
     'validator': 'recRate >= 0. and recRate <= 0.5',
    },
    {'name': 'numRep',
     'default': 5,
     'type': int,
     'label': 'Number of Replicate',
     'validator': 'numRep > 0',
    },
    {'name': 'measure',
     'default': "D'",
     'type': ('chooseOneOf', ["D'", 'R2']),
     'label': 'LD measure',
     'description': '''A measure of linkage disequilibrium to be outputed.
        Acceptable input includes
        |D': Lewontin's D' measure which is the standard LD measure divided
        by the theoretical maximum for the observed allele frequencies.
        |R2: R2 is the correlation coefficient between pairs of loci.
        ''',
    },
]

def simuLDDecay(popSize, gen, recRate, numRep, measure):
    '''Simulate the decay of linkage disequilibrium as a result
    of recombination.
    '''
    simu = sim.Simulator(
        sim.Population(size=popSize, ploidy=2, loci=[2]),
        rep=numRep)
    simu.evolve(
        initOps=[
            sim.InitSex(),
            sim.InitGenotype(haplotypes=[[0, 1], [1, 0]])
        ],
```

```
        matingScheme=sim.RandomMating(ops=sim.Recombinator(rates=recRate)),
        postOps=[
            sim.Stat(LD=[0, 1]),
            sim.IfElse(measure=="D'",
                sim.PyEval(r"'%.4f\t' % LD_prime[0][1]"), # if measure=="D'"
                sim.PyEval(r"'%.4f\t' % R2[0][1]")),      # if measure=="R2"
            sim.PyOutput('\n', reps=-1),
        ],
        gen = gen
    )

if __name__ == '__main__':
    # get all parameters
    pars = simuOpt.Params(options, '''A demonstration of the decay of linkage
        disequilibrium with the impact of genetic recombination.''', __doc__)
    if not pars.getParam():
        sys.exit(0) # cancelled or -h, --help
    # call the simulation function
    simuLDDecay(pars.popSize, pars.gen, pars.recRate, pars.numRep, pars.measure)
```

Lines 14–50 define five parameters `size`, `gen`, `recRate`, `numRep`, and `measure` using a list of parameter specification dictionaries. These dictionaries can have keys `name`, `default`, `type`, `label`, `description`, and `validator`, which are used to obtain, validate, and convert user inputs for each parameter (Table A.9). For example, because the type of parameter `numRep` is `int`, simuPOP will try to convert user inputs to an integer (e.g., from string `'50'` or `'5*10'` to `50`) and reject invalid inputs such as `50.5`. Similarly, parameter `measure` will only accept one of the two specified values `D'` and `R2`. In addition to one or more allowed Python types, the `type` field also accepts values such as `'numbers'`,

TABLE A.9 Acceptable Keys in a Parameter Specification Dictionary

Field	Usage
`name`	Name of the parameter
`default`	Default value for this parameter
`type`	Type of acceptable input, which help simuPOP determine the GUI widget for a parameter, how to convert user input to appropriate format, and how to validate a parameter
`label`	A label to display in the parameter input dialog. If this field is missing, a parameter will not be displayed in the parameter input dialog
`description`	A detailed description of the parameter, which will be displayed as tooltip of the parameter in the parameter input dialog and be used to generate help messages of the script
`validator`	A function or an expression to validate a user input

and `'filename'`, which accept a list of integer or float numbers, and a valid filename, respectively.

Although the `type` of parameter provides type validation for user inputs, a `validator` can be used to provide further validations. This item accepts a function or a Python expression. When a function is provided, it will be applied to a user input. A user input will be rejected if this function returns `False`. For example, if a parameter defines a probability, a function returned by function `simuOpt.valueBetween(0,1)` will return True only if the input is between 0 and 1.

This example uses expressions to validate parameters. These expressions are evaluated in a dictionary where variables are defined from names and values of parameters. A parameter will be rejected if its validating expression returns `False`. For example, expression `'gen > 0'` will reject 0 as an input for parameter `gen`. This method is very flexible in that information from other parameters can be used to validate a parameter. For example, if two parameters `opt1` and `op2` are required to have the same length, expression `'len(opt1) == len(opt2)'` can be used to validate both parameters.

Lines 52–73 define a function `simuLDDecay`, which is the main simulation function of this script. This function is an extension to Source code A.1. It allows the simulation of several replicates of the population simultaneously and also allows the display of two linkage disequilibrium measures.

Function `simuLDDecay` will only be executed if the script is executed as a script (if `__name__ == '__main__'`, line 75), not imported as a module. This is a very useful feature of Python because the same script can be executed directly or be used as a Python module to provide functions to another module. For example, if a user would like to call function `simuLDDecay` with different parameters, he or she can import this script as a module and call this function directly as follows:

```
from simuLDDecay import simuLDDecay
for r in [0.1, 0.01, 0.002]:
    simuLDDecay(1000, 50, r, 2, 'R2')
```

The last block (lines 75–82) is the execution part of this script. It defines a `simuOpt.Params` object using the parameter specification list `options`, a short description, and a detailed description `__doc__`. A function `pars.getParam()` is used to obtain values of parameters `size`, `gen`, `recRate`, `numRep`, and `measure`. If successful, parameter values are passed to function `simuLDDecay` to perform the simulation.

There are a number of advantages of using the `simuOpt.Params` class to handle user inputs. By specifying the type of parameters, this class automatically converts user inputs into required types. Arbitrary Python expressions are allowed so that users can use expressions such as `range(10)` to input long arguments. If a list of values (e.g., `numbers`) is needed, single input will be converted to a list automatically. The `simuOpt.Params` also validates user input and will reject a value if it is not of required type or does not pass specified validation function or expression.

The `getParams()` function of the `Params` class does all the hard job of obtaining values of parameters from command line, configuration file, a graphical parameter input dialog, or interactive user input. It first checks command line for argument `-h` or `-help` and will print a help message and return `False` if one of them is specified. This usage message is created from descriptions of the script and parameters. The help message of script simuLDDecay.py is listed below.

Before falling into a particular mode to collect user input, the `getParams` function processes command line arguments such as `-gen=50` to obtain values for parameters. Parameters of a Boolean type can be set to `True` by parameter `-param`. If a configuration file is specified by command line argument `-config`, this function will try to obtain values of parameters from this file. A configuration file can be created manually, but is most frequently saved by function `Params.saveConfig(filename)`.

The `getParams` function then checks for command line option `-gui` to determine which user interface to use. If the script is executed under a batch mode (`-gui=batch`), this function will return `True` directly, so default values will be used for parameters that are not specified from command line or a configuration file. For example, command

```
> simuLDDecay.py --gui=batch --size=5000
```

will execute the script with specified population size and default parameters of all other parameters. In contrast, if the script is executed under an interactive mode (`-gui=interactive`), if running in an interactive mode, this function will prompt users for values of parameters that have not been specified through command line or a configuration file.

If no gui mode is specified, the script will be executed in GUI mode where a parameter input dialog will be displayed for users to view and edit parameters (Figure A.2). Different GUI widgets such as checkbox and listbox will be used for different types of parameters. Detailed descriptions of parameters are displayed as tooltips to these parameters. A parameter

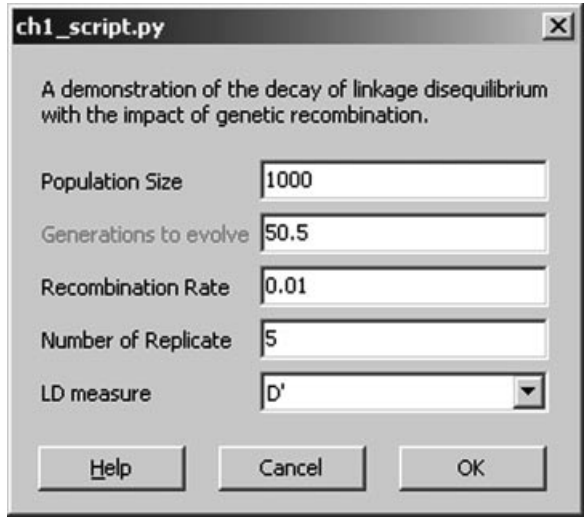

FIGURE A.2 Parameter input dialog for Source code A.16. A parameter input dialog with five parameters. The label of the second parameter is marked in red when the OK button is clicked because its input value is not of required type (`int`).

will be marked in red if an invalid parameter is detected when the OK button is pressed.

Values of each parameter can be accessed as attributes of the `Param` object if function `pars.getParam()` returns `True`. Line 82 of Source code A.16 uses this feature to pass values to function `simuLDDecay` to execute the main simulation program. Alternatively, values of all parameters can be returned as a list using function `Param.asList` or as a dictionary using function `Param.asDict`. Because parameter order and names of the `Param` object match those of the `simuLDDecay` function, we can also pass parameters to this function using

```
simuLDDecay(*pars.asList())
```

or

```
simLDDecay(**pars.asDict())
```

We prefer the method used in Source code A.16 because it is less error prone.

In summary, despite the availability of other parameter-handling modules, we recommend the use of module `simuOpt` because it allows a

self-documentary method to describe parameters and a flexible mechanism to execute the same script in both GUI and batch mode. However, for the sake of brevity, we do not use this style in examples we describe in this book.

REFERENCES

1. International HapMap Consortium, A haplotype map of the human genome. *Nature*, 437(7063):1299–1320, 2005.