simuPOP User's Guide

Release 1.1.7 (Rev: 5000)

Bo Peng

December 2004

Last modified
January 20, 2016

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

© 2004-2008 Bo Peng

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies. Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the sections entitled Copying and GNU General Public License are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one. Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Abstract

simuPOP is a general-purpose individual-based 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. In contrast to competing applications that use command-line options or configuration files to direct the execution of a limited number of predefined evolutionary scenarios, users of simuPOP's scripting interface could make use of many of its unique features, such as customized chromosome types, arbitrary nonrandom mating schemes, virtual subpopulations, information fields and Python operators, to construct and study almost arbitrarily complex evolutionary scenarios.

simuPOP is provided as a number of Python modules, which consist of a large number of Python objects and functions, including population, mating schemes, operators (objects that manipulate populations) and simulators to coordinate the evolutionary processes. It is the users' responsibility to write a Python script to glue these pieces together and form a simulation. At a more user-friendly level, an increasing number of functions and scripts contributed by simuPOP users is available in the online simuPOP cookbook. They provide useful functions for different applications (e.g. load and manipulate HapMap samples, import and export files from another application) and allow users who are unfamiliar with simuPOP to perform a large number of simulations ranging from basic population genetics models to generating datasets under complex evolutionary scenarios.

This user's guide shows you how to install and use simuPOP using a large number of examples. It describes all important concepts and features of simuPOP and demonstrates how to use them in a simuPOP script. Although the new Python 3.x releases are incompatible with Python 2.x, examples in this book are written in a style that is compatible with both versions of Python. For a complete and detailed description about all simuPOP functions and classes, please refer to the *simuPOP Reference Manual*. All resources, including a pdf version of this guide and a mailing list can be found at the simuPOP homepage `http://simupop.sourceforge.net`.

How to cite simuPOP:

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

Bo Peng and Christopher Amos (2008) Forward-time simulations of nonrandom mating populations using simuPOP. *bioinformatics*, **24** (11) 1408-1409.

Contents

1 Introduction **3**
- 1.1 What is simuPOP? . . . 3
- 1.2 An overview of simuPOP concepts . . . 4
- 1.3 Features . . . 5
- 1.4 License, Distribution and Installation . . . 7
- 1.5 How to read this user's guide . . . 7
- 1.6 Other help sources . . . 8

2 Loading and running simuPOP **9**
- 2.1 Pythonic issues . . . 9
  - 2.1.1 `from simuPOP import` * v.s. `import simuPOP` . . . 9
  - 2.1.2 References and the `clone()` member function . . . 10
  - 2.1.3 Zero-based indexes, absolute and relative indexes . . . 10
  - 2.1.4 Ranges and iterators . . . 11
  - 2.1.5 Empty, `ALL_AVAIL` and dynamic values for parameters `loci`, `reps`, `ancGen` and `subPops` . . . 11
  - 2.1.6 User-defined functions and class `WithArgs` * . . . 12
  - 2.1.7 Exception handling * . . . 13
- 2.2 Loading simuPOP modules . . . 13
  - 2.2.1 Short, long, binary, mutant and lineage modules and their optimized versions . . . 13
  - 2.2.2 Execution in multiple threads . . . 15
  - 2.2.3 Graphical user interface . . . 15
- 2.3 Online help system . . . 16
- 2.4 Debug-related functions and operators * . . . 16
- 2.5 Random number generator * . . . 18

3 Individuals and Populations **21**
- 3.1 Genotypic structure . . . 21
  - 3.1.1 Haploid, diploid and haplodiploid populations . . . 23
  - 3.1.2 Autosomes, sex chromosomes, mitochondrial, and other types of chromosomes * . . . 23
  - 3.1.3 Information fields . . . 24
- 3.2 Individual . . . 26
  - 3.2.1 Access individual genotype . . . 26
  - 3.2.2 individual sex, affection status and information fields . . . 28
- 3.3 Population . . . 28
  - 3.3.1 Access and change individual genotype . . . 29
  - 3.3.2 Subpopulations . . . 29
  - 3.3.3 Virtual subpopulations and virtual splitters * . . . 30
  - 3.3.4 Advanced virtual subpopulation splitters ** . . . 31
  - 3.3.5 Access individuals and their properties . . . 33
  - 3.3.6 Attach arbitrary auxillary information using information fields . . . 35
  - 3.3.7 Keep track of ancestral generations . . . 36
  - 3.3.8 Change genotypic structure of a population . . . 38

3.3.9 Remove or extract individuals and subpopulations from a population . . . 39
3.3.10 Store arbitrary population information as population variables . . . 40
3.3.11 Save and load a population . . . 41
3.3.12 Import and export datasets in unsupported formats * . . . 42

4 simuPOP Operators **45**
4.1 Introduction to operators . . . 45
4.1.1 Apply operators to selected replicates and (virtual) subpopulations at selected generations . . . 45
4.1.2 Applicable populations and (virtual) subpopulations . . . 46
4.1.3 Dynamically determined loci (parameter `loci`) * . . . 47
4.1.4 Write output of operators to one or more files . . . 48
4.1.5 During-mating operators . . . 51
4.1.6 Function form of an operator . . . 52
4.2 Initialization . . . 53
4.2.1 Initialize individual sex (operator `InitSex`) . . . 53
4.2.2 Initialize genotype (operator `InitGenotype`) . . . 53
4.2.3 Initialize information fields (operator `InitInfo`) . . . 55
4.3 Expressions and statements . . . 56
4.3.1 Output a Python string (operator `PyOutput`) . . . 56
4.3.2 Execute Python statements (operator `PyExec`) . . . 56
4.3.3 Evaluate and output Python expressions (operator `PyEval`) . . . 57
4.3.4 Expression and statement involving individual information fields (operator `InfoEval` and `InfoExec`) * . . . 58
4.3.5 Using functions in external modules in simuPOP expressions and statements . . . 59
4.4 Demographic changes . . . 60
4.4.1 Migration (operator `Migrator`) . . . 60
Migration by probability . . . 60
Migration by proportion and counts . . . 61
Theoretical migration models . . . 62
migrate from virtual subpopulations * . . . 62
Arbitrary migration models ** . . . 63
4.4.2 Migration using backward migration matrix (operator `BackwardMigrator`) . . . 64
4.4.3 Split subpopulations (operators `SplitSubPops`) . . . 67
4.4.4 Merge subpopulations (operator `MergeSubPops`) . . . 68
4.4.5 Resize subpopulations (operator `ResizeSubPops`) . . . 68
4.4.6 Time-dependent migration rate . . . 69
4.5 Genotype transmitters . . . 70
4.5.1 Generic genotype transmitters (operators `GenoTransmitter`, `CloneGenoTransmitter`, `MendelianGenoTransmitter`, `SelfingGenoTransmitter`, `HaplodiploidGenoTransmitter`, and `MitochondrialGenoTransmitter`) * . . . 70
4.5.2 Recombination (Operator `Recombinator`) . . . 71
4.5.3 Gene conversion (Operator `Recombinator`) * . . . 73
4.5.4 Tracking all recombination events ** . . . 74
4.6 Mutation . . . 75
4.6.1 Mutation models specified by rate matrixes (`MatrixMutator`) . . . 75
4.6.2 k-allele mutation model (`KAlleleMutator`) . . . 76
4.6.3 Diallelic mutation models (`SNPMutator`) . . . 77
4.6.4 Nucleotide mutation models (`AcgtMutator`) . . . 78
4.6.5 Mutation model for microsatellite markers (`StepwiseMutator`) . . . 80
4.6.6 Simulating arbitrary mutation models using a hybrid mutator (`PyMutator`)* . . . 81
4.6.7 Mixed mutation models (`MixedMutator`) ** . . . 81
4.6.8 Context-dependent mutation models (`ContextMutator`)** . . . 82
4.6.9 Manually-introduced mutations (`PointMutator`) . . . 84
4.6.10 Apply mutation to (virtual) subpopulations * . . . 85

4.6.11 Allele mapping ** . . . 87
4.6.12 Mutation rate and transition matrix of a `MatrixMutator` ** . . . 87
4.6.13 Infinite-sites model and other simulation techniques ** . . . 88
4.6.14 Recording and tracing individual mutants ** . . . 90
4.7 Penetrance . . . 91
4.7.1 Map penetrance model (operator `MapPenetrance`) . . . 92
4.7.2 Multi-allele penetrance model (operator `MaPenetrance`) . . . 92
4.7.3 Multi-loci penetrance model (operator `MlPenetrance`) . . . 93
4.7.4 Hybrid penetrance model (operator `PyPenetrance`) . . . 94
4.8 Quantitative trait . . . 95
4.8.1 A hybrid quantitative trait operator (operator `PyQuanTrait`) . . . 96
4.9 Natural Selection . . . 96
4.9.1 Natural selection through the selection of parents . . . 96
4.9.2 Natural selection through the selection of offspring * . . . 97
4.9.3 Are two selection scenarios equivalent? ** . . . 99
4.9.4 Map selector (operator `MapSelector`) . . . 99
4.9.5 Multi-allele selector (operator `MaSelector`) . . . 100
4.9.6 Multi-locus selection models (operator `MlSelector`) . . . 101
4.9.7 A hybrid selector (operator `PySelector`) . . . 103
4.9.8 Multi-locus random fitness effects (operator `PyMlSelector`) . . . 104
4.9.9 Alternative implementations of natural selection . . . 105
4.9.10 Frequency dependent or dynamic selection pressure * . . . 106
4.9.11 Support for virtual subpopulations * . . . 107
4.9.12 Natural selection in heterogeneous mating schemes ** . . . 109
4.10 Tagging operators . . . 109
4.10.1 Inheritance tagger (operator `InheritTagger`) . . . 109
4.10.2 Summarize parental informatin fields (operator `SummaryTagger`) . . . 110
4.10.3 Tracking parents (operator `ParentsTagger`) . . . 111
4.10.4 Tracking index of offspring within families (operator `OffspringTagger`) . . . 111
4.10.5 Assign unique IDs to individuals (operator `IdTagger`) . . . 112
4.10.6 Tracking Pedigrees (operator `PedigreeTagger`) . . . 113
4.10.7 A hybrid tagger (operator `PyTagger`) . . . 114
4.10.8 Tagging that involves other parental information . . . 115
4.11 Statistics calculation (operator `Stat`) . . . 116
4.11.1 How statistics calculation works . . . 116
4.11.2 `defdict` datatype . . . 116
4.11.3 Support for virtual subpopulations . . . 117
4.11.4 Counting individuals by sex and affection status . . . 118
4.11.5 Number of segregating and fixed sites . . . 119
4.11.6 Allele count and frequency . . . 120
4.11.7 Genotype count and frequency . . . 121
4.11.8 Homozygote and heterozygote count and frequency . . . 122
4.11.9 Haplotype count and frequency . . . 122
4.11.10 Summary statistics of information fields . . . 123
4.11.11 Linkage disequilibrium . . . 124
4.11.12 Genetic association . . . 126
4.11.13 population structure . . . 127
4.11.14 Hardy-Weinberg equilibrium test . . . 128
4.11.15 Measure of Inbreeding . . . 129
4.11.16 Effective population size . . . 130
4.11.17 Other statistics . . . 134
4.11.18 Support for sex and customized chromosome types . . . 134
4.12 Conditional operators . . . 135
4.12.1 Conditional operator (operator `IfElse`) * . . . 135

4.12.2 Conditionally terminate an evolutionary process (operator `TerminateIf`) . . . 137
4.12.3 Conditionally revert an evolutionary process to a saved state (operator `RevertIf`) . . . 137
4.12.4 Conditional during mating operator (operator `DiscardIf`) . . . 139
4.13 Miscellaneous operators . . . 140
4.13.1 An operator that does nothing (operator `NoneOp`) . . . 140
4.13.2 dump the content of a population (operator `Dumper`) . . . 141
4.13.3 Save a population during evolution (operator `SavePopulation`) . . . 142
4.13.4 Pause and resume an evolutionary process (operator `Pause`) * . . . 142
4.13.5 Measuring execution time of operators (operator `TicToc`) * . . . 143
4.14 Hybrid and Python operators . . . 143
4.14.1 Hybrid operators . . . 143
4.14.2 Python operator `PyOperator` * . . . 144
4.14.3 During-mating Python operator * . . . 146
4.14.4 Define your own operators * . . . 146

5 Evolving populations **149**
5.1 Mating Schemes . . . 149
5.1.1 Control the size of the offspring generation . . . 149
5.1.2 Advanced use of demographic functions * . . . 153
5.1.3 Determine the number of offspring during mating . . . 154
5.1.4 Dynamic population size determined by number of offspring * . . . 156
5.1.5 Determine sex of offspring . . . 157
5.1.6 Monogamous mating . . . 159
5.1.7 Polygamous mating . . . 160
5.1.8 Asexual random mating . . . 161
5.1.9 Mating in haplodiploid populations . . . 161
5.1.10 Self-fertilization . . . 162
5.1.11 Heterogeneous mating schemes * . . . 163
5.1.12 Conditional mating schemes . . . 166
5.2 Simulator . . . 167
5.2.1 Add, access and remove populations from a simulator . . . 167
5.2.2 Number of generations to evolve . . . 169
5.2.3 Evolve populations in a simulator . . . 169
5.3 Non-random and customized mating schemes * . . . 172
5.3.1 The structure of a homogeneous mating scheme * . . . 172
5.3.2 Offspring generators * . . . 173
5.3.3 Genotype transmitters * . . . 174
5.3.4 A Python parent chooser * . . . 176
5.3.5 Using C++ to implement a parent chooser ** . . . 178
5.4 Age structured populations with overlapping generations ** . . . 181
5.5 Tracing allelic lineage * . . . 184
5.6 Pedigrees . . . 186
5.6.1 Create a pedigree object . . . 186
5.6.2 Locate close and remote relatives of each individual . . . 187
5.6.3 Identify pedigrees (related individuals) . . . 189
5.6.4 Save and load pedigrees . . . 190
5.7 Evolve a population following a specified pedigree structure ** . . . 192
5.8 Simulation of mitochondrial DNAs (mtDNAs) * . . . 197

6 Utility Modules **201**
6.1 Module `simuOpt` (function `simuOpt.setOptions`) . . . 201
6.1.1 Class `simuOpt.Params` (deprecated) . . . 201
Define a parameter specification list. . . . 202
Get parameters (function `Params.getParam`) . . . 204

Access, manipulate and extract parameters . . . 205
6.2 Module simuPOP.utils . . . 206
6.2.1 Trajectory simulation (classes Trajectory and TrajectorySimulator) . . . 206
Forward-time trajectory simulations (function simulateForwardTrajectory) . . . 207
Backward-time trajectory simulations (function simulateBackwardTrajectory). . . . 208
6.2.2 Graphical or text-based progress bar (class ProgressBar) . . . 211
6.2.3 Display population variables (function viewVars) . . . 211
6.2.4 Import simuPOP population from files in GENEPOP, PHYLIP and FSTAT formats (function importPopulation) . . . 211
6.2.5 Export simuPOP population to files in STRUCTURE, GENEPOP, FSTAT, Phylip, PED, MAP, MS, and CSV formats (function export and operator Exporter) . . . 212
6.2.6 Export simuPOP population in csv format (function saveCSV, deprecated) . . . 215
6.3 Module simuPOP.demography . . . 216
6.3.1 Predefined migration models . . . 216
6.3.2 Uniform interface of demographic models . . . 218
6.3.3 Demographic models defined by outcomes . . . 219
6.3.4 Demographic models defined by population changes (events) . . . 223
6.3.5 Predefined demographic models for human populations . . . 226
6.3.6 Demographic model without predefined generations to evolve * . . . 227
6.4 Module simuPOP.plotter . . . 229
6.4.1 Derived keyword arguments * . . . 230
6.4.2 Plot of expressions and their histories (operator plotter.VarPlotter) . . . 230
6.4.3 Scatter plots (operator plotter.ScatterPlotter) . . . 234
6.5 Module simuPOP.sampling . . . 237
6.5.1 Introduction . . . 237
6.5.2 Sampling individuals randomly (class RandomSampler, functions drawRandomSample and drawRandomSamples) . . . 237
6.5.3 Sampling cases and controls (class CaseControlSampler, functions CaseControlSample and CaseControlSamples) . . . 237
6.5.4 Sampling Pedigrees (functions indexToID and plotPedigree) . . . 238
6.5.5 Sampling affected sibpairs (class AffectedSibpairSampler, functions drawAffectedSibpairSample(s)) . . . 239
6.5.6 Sampling nuclear families (class NuclearFamilySampler, functions drawNuclearFamilySample and drawNuclearFamilySamples) . . . 239
6.5.7 Sampling three-generation families (class ThreeGenFamilySampler, functions drawThreeGenFamilySample and drawThreeGenFamilySamples) . . . 240
6.5.8 Sampling different types of samples (class CombinedSampler, functions drawCombinedSample and drawCombinedSamples) . . . 240
6.5.9 Sampling from subpopulations and virtual subpopulations * . . . 241
6.6 Module simuPOP.gsl . . . 242
6.7 Module simuPOP.sandbox . . . 242

7 A real world example **243**
7.1 Simulation scenario . . . 243
7.2 Demographic model . . . 243
7.3 Mutation and selection models . . . 245
7.4 Output statistics . . . 245
7.5 Initialize and evolve the population . . . 247
7.6 Option handling . . . 248

Index **255**

List of Examples

1.1 A simple example . . . 4
2.1 Reference to a population in a simulator . . . 10
2.2 Conversion between absolute and relative indexes . . . 10
2.3 Ranges and iterators . . . 11
2.4 Use of user-defined Python function in simuPOP . . . 12
2.5 Specify arguments of user-provided function using function WithArgs . . . 13
2.6 Use of standard simuPOP modules . . . 14
2.7 Use of optimized simuPOP modules . . . 14
2.8 Getting help using the `help()` function . . . 16
2.9 Turn on/off debug information . . . 17
2.10 Turn on and off debug information during evolution. . . . 17
2.11 Use saved random seed to replay an evolutionary process . . . 18
3.1 Genotypic structure functions . . . 22
3.2 An example of haplodiploid population . . . 23
3.3 Different chromosome types . . . 24
3.4 Basic usage of information fields . . . 25
3.5 Access individual genotype . . . 27
3.6 Access Individual properties . . . 28
3.7 Manipulation of subpopulations . . . 29
3.8 Use of subpopulation names . . . 30
3.9 Define virtual subpopulations in a population . . . 30
3.10 Applications of virtual subpopulations . . . 31
3.11 Advanced virtual subpopulation usages. . . . 32
3.12 Access individuals of a population . . . 33
3.13 Access Individual properties in batch mode . . . 35
3.14 Add and use of information fields in a population . . . 35
3.15 Ancestral populations . . . 37
3.16 Add and remove loci and chromosomes . . . 38
3.17 Extract individuals, loci and information fields from an existing population . . . 39
3.18 population variables . . . 40
3.19 Expression evaluation in the local namespace of a population . . . 41
3.20 Save and load a population . . . 42
3.21 Import a population from another file format . . . 42
4.1 Applicable generations of an operator. . . . 46
4.2 Apply operators to a subset of populations . . . 47
4.3 Natural selection with dynamically determined loci . . . 48
4.4 Use the output and outputExpr parameters . . . 49
4.5 Output to a Python function . . . 50
4.6 Genotype transmitters . . . 51
4.7 The function form of operator `InitGenotype` . . . 52
4.8 Initialize individual sex . . . 53
4.9 Initialize individual genotype . . . 54
4.10 initialize information fields . . . 56

4.11 Execute Python statements during evolution . . . 56
4.12 Evaluate a expression and statements in a population's local namespace. . . . 57
4.13 Evaluate expressions using individual information fields . . . 58
4.14 Execute statements using individual information fields . . . 59
4.15 Write the status of an evolutionary process every 10 seconds . . . 59
4.16 Migration by probability . . . 61
4.17 Migration by proportion and count . . . 61
4.18 Migration from virtual subpopulations . . . 63
4.19 Manual migration . . . 64
4.20 Migration using a backward migration matrix . . . 65
4.21 Split subpopulations by size . . . 67
4.22 Split subpopulations by proportion . . . 67
4.23 Split subpopulations by individual information field . . . 68
4.24 Merge multiple subpopulations into a single subpopulation . . . 68
4.25 Resize subpopulation sizes . . . 69
4.26 Varying migration rate . . . 69
4.27 Genetic recombination at all and selected loci . . . 72
4.28 Genetic recombination rates specified by intensity . . . 72
4.29 Gene conversion . . . 74
4.30 Tracking all recombination events . . . 74
4.31 General mutator specified by a mutation rate matrix . . . 75
4.32 A k-allele mutation model . . . 76
4.33 A diallelic directional mutation model . . . 77
4.34 A Kimura's 2 parameter mutation model . . . 79
4.35 A standard and a generalized stepwise mutation model . . . 80
4.36 A hybrid mutation model . . . 81
4.37 A mixed k-allele and stepwise mutation model . . . 82
4.38 A context-dependent mutation model . . . 83
4.39 A hybrid context-dependent mutation model . . . 84
4.40 Use a point mutator to introduce a disease predisposing allele . . . 84
4.41 Applying mutation to virtual subpopulations. . . . 85
4.42 Allele mapping for mutation operators . . . 87
4.43 Mimicking an infinite-sites model using mutation events as alleles . . . 89
4.44 Count number of mutants from mutator outputs . . . 90
4.45 A penetrance model that uses pre-defined fitness value . . . 92
4.46 A multi-allele penetrance model . . . 92
4.47 A multi-loci penetrance model . . . 93
4.48 A hybrid penetrance model . . . 94
4.49 A hybrid quantitative trait model . . . 96
4.50 Natural selection through the selection of parents . . . 97
4.51 Natural selection through the selection of offspring . . . 98
4.52 A selector that uses pre-defined fitness value . . . 99
4.53 A multi-allele selector . . . 100
4.54 A multi-locus multi-allele selection model in a haploid population . . . 101
4.55 A multi-loci selector . . . 102
4.56 A hybrid selector . . . 103
4.57 Random fitness effect . . . 104
4.58 Natural selection according to individual affection status . . . 106
4.59 Frequency dependent selection . . . 107
4.60 Selector in virtual subpopulations . . . 108
4.61 Selection against offspring in virtual subpopulations . . . 108
4.62 Use an inherit tagger to track offspring of individuals . . . 110
4.63 Using a summary tagger to calculate mean fitness of parents. . . . 110
4.64 Keeping constant family size in the presence of natural selection against offspring . . . 111

4.65 Assign unique IDs to individuals . . . 112
4.66 Output a complete pedigree of an evolutionary process . . . 113
4.67 Use of a hybrid tagger to pass parental information to offspring . . . 114
4.68 Tagging that involves other parental information . . . 115
4.69 The defdict datatype . . . 117
4.70 Add suffixes to variables set by multiple Stat operators . . . 118
4.71 Count individuals by sex and/or affection status . . . 118
4.72 Count number of segregating and fixed sites . . . 119
4.73 Calculate allele frequency in affected and unaffected individuals . . . 120
4.74 Counting genotypes in a population . . . 121
4.75 Counting homozygotes and heterozygotes in a population . . . 122
4.76 Counting haplotypes in a population . . . 122
4.77 Calculate summary statistics of information fields . . . 123
4.78 Linkage disequilibrium measures . . . 125
4.79 Genetic association tests . . . 127
4.80 Measure of population structure . . . 128
4.81 Hardy Weinberg Equilibrium test . . . 128
4.82 Frequency of IBD as a measure of inbreeding coefficient . . . 129
4.83 Demographic effective population size . . . 130
4.84 Temporal effective population size using a fixed baseline sample . . . 131
4.85 Temporal effective population size between consecutive samples . . . 132
4.86 Effective population size estimated using a LD based method . . . 133
4.87 Statistics for sex and customized chromosome types . . . 135
4.88 A conditional opeartor with fixed condition . . . 135
4.89 A conditional opeartor with dynamic condition . . . 136
4.90 Terminate the evolution of all populations in a simulator . . . 137
4.91 Revert an evolutionary process to a previous saved state when an introduced allele is lost . . . 138
4.92 Use operator DiscardIf to generate case control samples . . . 139
4.93 dump the content of a population . . . 141
4.94 Save snapshots of an evolving population . . . 142
4.95 Pause the evolution of a simulation . . . 142
4.96 Monitor the performance of operators . . . 143
4.97 Use a hybrid operator . . . 144
4.98 A frequency dependent mutation operator . . . 145
4.99 Use a PyOperator during evolution . . . 145
4.100 Use a during-mating PyOperator . . . 146
4.101 Define a new Python operator . . . 147
5.1 Free change of subpopulation sizes . . . 150
5.2 Force constant subpopulation sizes . . . 151
5.3 Use a demographic function to control population size . . . 151
5.4 Use parental population to determine the size of offspring population . . . 152
5.5 Change of population size caused by natural selection . . . 152
5.6 Use a demographic function to split parental population . . . 153
5.7 Control the number of offspring per mating event. . . . 155
5.8 Dynamic population size determined by number of offspring . . . 156
5.9 Determine the sex of offspring . . . 158
5.10 Sexual monogamous mating . . . 159
5.11 Sexual polygamous mating . . . 160
5.12 Asexual random mating . . . 161
5.13 Random mating in haplodiploid populations . . . 161
5.14 Selfing mating scheme . . . 162
5.15 Applying different mating schemes to different subpopulations . . . 163
5.16 Applying different mating schemes to different virtual subpopulations . . . 164
5.17 A weighting scheme used by heterogeneous mating schemes. . . . 166

5.18 Apply different mating schemes for different replicates at different generations . . . 166
5.19 Create a simulator and access populations . . . 168
5.20 Generation number of a simulator . . . 169
5.21 describe an evolutionary process . . . 170
5.22 Define a random mating scheme . . . 172
5.23 Define a sequential selfing mating scheme . . . 173
5.24 A controlled random mating scheme . . . 174
5.25 A customized genotype transmitter for sex-specific recombination . . . 175
5.26 A sample generator function . . . 176
5.27 A hybrid parent chooser that chooses parents by their social status . . . 177
5.28 Use built-in parent choosers in a Python parent chooser . . . 178
5.29 Implement a parent chooser in C++ . . . 179
5.30 An interface file for the myParentsChooser class . . . 179
5.31 Building and installing the myParentsChooser module . . . 180
5.32 Implement a parent chooser in C++ . . . 180
5.33 Example of the evolution of age-structured population. . . . 182
5.34 Contribution of genetic information from ancestors . . . 185
5.35 Distribution of age of mutants . . . 185
5.36 Locate close and distant relatives of individuals . . . 188
5.37 Identify all ancestors, offspring or all related individuals . . . 189
5.38 Save and load a complete pedigree . . . 190
5.39 Use a pedigree mating scheme to replay an evolutionary process. . . . 192
5.40 Replay an evolutionary process of an age-structured population . . . 194
5.41 Transmission of mitochondrial chromosomes . . . 198
5.42 Evolution of multiple organelles in mitochondrion . . . 199
6.1 A sample parameter specification list . . . 203
6.2 Using simuOpt.param to specify parameters . . . 203
6.3 Get prameters using function getParam . . . 204
6.4 Use the simuOpt object . . . 206
6.5 Simulation and use of forward-time simulated trajectories. . . . 207
6.6 Simulation and use of backward-time simulated trajectories. . . . 209
6.7 Using a text-based progress bar . . . 211
6.8 Using function viewVars to display population variables . . . 211
6.9 Save and load a population . . . 212
6.10 Export and import in MS format . . . 213
6.11 Using function saveCSV to save a simuPOP population in different formats . . . 215
6.12 A demographic model for human population . . . 221
6.13 A event-based demographic model . . . 224
6.14 A demographic model with a terminator . . . 227
6.15 Use rpy or matplotlib to plot an expression . . . 231
6.16 Separate figures by replicate . . . 231
6.17 Separate figures by Dimension . . . 232
6.18 Use ScatterPlotter to plot ancestry of individuals with geographic information. . . . 235
6.19 Draw random samples from a structured population . . . 237
6.20 Draw case control samples from a population and perform association test . . . 238
6.21 Draw affected sibpairs from a population . . . 239
6.22 Draw nuclear families from a population . . . 239
6.23 Draw three-generation families from a population . . . 240
6.24 Draw different types of samples from a population . . . 240
6.25 Draw samples from a virtual subpopulation. . . . 241
6.26 Sampling separately from different virtual subpopulations . . . 241
7.1 A demographic function producer . . . 244
7.2 A customized operator to calculate effective number of alleles . . . 246
7.3 Evolve a population subject to mutation and selection . . . 247

7.4 A complete simulation script 248

List of Figures

1.1 A life cycle of an evolutionary process . . . 4

3.1 Inheritance of different types of chromosomes in a diploid population . . . 25
3.2 Memory layout of individual genotype . . . 27

5.1 A homogeneous mating scheme . . . 150
5.2 Illustration of a heterogeneous mating scheme . . . 163
5.3 Orders at which operators are applied during an evolutionary process . . . 170

6.1 A sample parameter input dialog . . . 205
6.2 Simulated trajectories of one locus in two subpopulations . . . 209
6.3 Simulated trajectories of two unlinked loci . . . 210
6.4 Using wxPython to display population variables . . . 212
6.5 A linear and two stage exponential population growth model, followed by population admixture . . . 221
6.6 A exponential population growth followed by bottleneck demographic model . . . 223
6.7 A event-based demographic model . . . 227
6.8 Out of Africa model for YRI, CEU, and CHB populations . . . 228
6.9 Settlement of New World model for Mexican America population . . . 229
6.10 Demographic models for African, Asian and European populations (cosi) . . . 230
6.11 rpy_80.png saved at generation 80 for Example 6.15 . . . 232
6.12 Allele frequency trajectories separated by replicates . . . 233
6.13 Allele frequency trajectories separated by loci . . . 234
6.14 Plot of individuals with ancestry marked by different colors . . . 236

7.1 Parameter input dialog of the simuCDCV script . . . 253

Chapter 1

Introduction

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 from parents in the parental population.
- 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.
- simuPOP is a **general-purpose** simulator that is designed to simulate arbitrary evolutionary processes. In contrast to competing applications that use command-line options or configuration files to direct the execution of a limited number of predefined evolutionary scenarios, users of simuPOP's scripting interface could make use of many of its unique features, such as customized chromosome types, arbitrary nonrandom mating schemes, virtual subpopulations, information fields and Python operators, to construct and study almost arbitrarily complex evolutionary scenarios. In addition, because simuPOP provides a large number of functions to manipulate populations, it can be used as an data manipulatation and analysis tool.

simuPOP is provided as a number of Python modules, which consist of a large number of Python objects and functions, including Population, mating schemes, operators (objects that manipulate populations) and simulators to coordinate the evolutionary processes. It is the users' responsibility to write a Python script to glue these pieces together and form a simulation. At a more user-friendly level, an increasing number of functions and scripts contributed by simuPOP users is available in the online simuPOP cookbook (`http://simupop.sourceforge.net/cookbook`). They provide useful functions for different applications (e.g. load and manipulate HapMap samples, import and export files from another application) and allow users who are unfamiliar with simuPOP to perform a large number of simulations ranging from basic population genetics models to generating datasets under complex evolutionary scenarios.

1.2 An overview of simuPOP concepts

A simuPOP **population** consists of **individuals** of the same **genotype structure**, which includes properties such as number of homologous sets of chromosomes (ploidy), number of chromosomes, and names and locations of markers on each chromosome. In addition to basic information such as genotypes and sex, individuals can have arbitray auxillary values as **information fields**. Individuals in a population can be divided into **subpopulations** that can be further grouped into **virtual subpopulations** according to individual properties such as sex, affection status, or arbitrary auxiliary information such as age. Whereas subpopulations define boundaries of individuals that restrict the flow of individuals and their genotypes (mating happens within subpopulations), virtual subpopulations are groups of individuals who share the same properties, with membership of individuals change easily with change of individual properties.

Figure 1.1: A life cycle of an evolutionary process

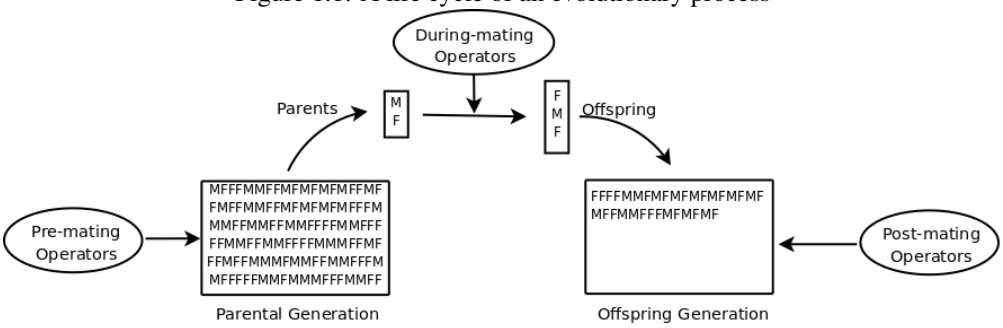


Illustration of the discrete-generation evolutionary model used by simuPOP.

Operators are Python objects that act on a population. They can be applied to a population before or after mating during a life cycle of an evolutionary process (Figure 1.1), or to parents and offspring during the production of each offspring. Arbitrary numbers of operators can be applied to an evolving population.

A simuPOP **mating scheme** is responsible for choosing parent or parents from a parental (virtual) subpopulation and for populating an offspring subpopulation. simuPOP provides a number of pre-defined **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 applies multiple homogeneous mating schemes to different (virtual) subpopulations.

simuPOP evolves a population generation by generation, following the evolutionary cycle depicted in Figure 1.1. Briefly speaking, a number of **operators** such as a `KAlleleMutator` are applied to a population before a mating scheme repeatedly chooses a parent or parents to produce offspring. **During-mating operators** such as `Recombinator` can be applied by a mating scheme to transmit parental genotype to offspring. After an offspring population is populated, other **operators** can be applied, for example, to calculate and output population statistics. The offspring population will then become the parental population of the next evolutionary cycle. Many simuPOP operators can be applied in different stages so the type of an operator is determined by the stage at which it is applied. Several populations, or replicates of a single population, could form a **simulator** and evolve together.

Listing 1.1: A simple example

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=1000, loci=2)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[1, 2, 2, 1])
...     ],
...     matingScheme=sim.RandomMating(ops=sim.Recombinator(rates=0.01)),
```

```
...     postOps=[
...         sim.Stat(LD=[0, 1], step=10),
...         sim.PyEval(r"'%.2f\n' % LD[0][1]", step=10),
...     ],
...     gen=100
... )
0.25
0.22
0.21
0.19
0.17
0.15
0.15
0.12
0.10
0.10
100L
```

Some of these concepts are demonstrated in Example 1.1, where a standard diploid Wright-Fisher model with recombination is simulated. The first line imports the standard simuPOP module. The second line creates a diploid population with 1000 individuals, each having one chromosome with two loci. The `evolve()` function evolves the population using a random mating scheme and four 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 the same genotype `12/21`. The populations are then evolved for 100 generations. A random mating scheme is used to generate offspring. Instead of using the default Mendelian genotype transmitter, a `Recombinator` (during-mating operator) is used to recombine parental chromosomes with the given recombination rate `0.01` during the generation of offspring. The other operators are applied to the offspring generation (post-mating) 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 a local variable space of the Population. The last operator `PyEval` outputs calculated linkage disequilibrium values with 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.

1.3 Features

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

1. simuPOP provides three types of modules that use 1, 8 or 32/64 bits to store an allele. The binary module (1 bit) is suitable for simulating a large number of SNP markers, and the long module (32 or 64 bits depending on platform) is suitable for simulating some population genetics models such as the infinite allele mutation model.

2. [NEW in simuPOP 1.0.7] simuPOP provides modules to store a large number of rare variants in a compressed manner (the mutant module), and to store origin of each allele so that it is easy to track allelic lineage during evolution.

3. The core of simuPOP is implemented in C++ which is heavily optimized for large-scale simulations. simuPOP can be executed in multiple threads with boosted performance on modern multi-core CPUs.

4. In addition to autosomes and sex chromosomes, simuPOP supports arbitrary types of chromosomes through customizable genotype transmitters. Random maternal transmission of mitochondrial DNAs is supported as a special case of this feature.

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

6. simuPOP does not impose a limit on the number of homologous sets of chromosomes, the size of the genome or populations. The size of your simulation is only limited by the physical memory of your computer.

7. During an evolutionary process, a population can hold more than one most-recent generation. Pedigrees can be sampled from such multi-generation populations.

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

9. simuPOP provides more than 60 operators that cover all important aspects of genetic studies. These include mutation (*e.g.* *k*-allele, stepwise, generalized stepwise and context-sensitive mutation models), migration (arbitrary, can create new subpopulation), recombination and gene conversion (uniform or nonuniform), selection (single-locus, additive, multiplicative or hybrid multi-locus models, support selection of both parents and offspring), penetrance (single, multi-locus or hybrid), ascertainment (case–control, affected sibpairs, random, nuclear and large Pedigrees), statistics calculation (including but not limited to allele, genotype, haplotype, heterozygote number and frequency; linkage disequilibrium measures, Hardy-Weinberg test), pedigree tracing, visualization (using R or other Python modules) and load/save in simuPOP's native format and many external formats such as Linkage.

10. Mating schemes can work on virtual subpopulations of a subpopulation. For example, positive assortative mating can be implemented by mating individuals with similar properties such as ancestry and overlapping generations could be simulated by copying individuals acorss generations. The number of offspring per mating event can be fixed or can follow a statistical distribution.

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

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

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

- simuPOP is the only application that embodies the concept of virtual subpopulation that allows evolutions at a finer scale. This is required for realistic simulations of complex evolutionary scenarios.

- simuPOP allows users to examine an evolutionary process very closely because all simuPOP objects are Python objects that can be assessed using their member functions. For example, users can keep track of genotype at particular loci during evolution. In contrast, other programs work more or less like a black box where only limited types of statistics can be outputted.

1.4 License, Distribution and Installation

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

If a binary distribution is unavailable for a specific platform, it is usually easy to compile simuPOP from source, following the standard "`python setup.py install`" procedure. Please refer to the `installation` section of the simupop website for instructions for specific platforms and compilers.

simuPOP is available for Python 2.4 and later, including the new Python 3.x releases. Although Python 3 is incompatible with Python 2 in many ways, examples in this guide are written in a style that is compatible with both versions of Python. Some non-classic usages include the use of `a//b` instead of `a/b` for floored division and `list(range(3))` instead of `range(3)` for sequece `[0,1,2]` In particular, we use

```
print("Population size is %d" % size)
```

instead of

```
print "Population size is %d" % size
```

to output strings because the former is valid in Python 2.x (print a tuple with one element) and will generate the same output in Python 3.x. Of course, users of simuPOP can choose to use other styles.

Thanks to the 'glue language' nature of Python, it is easy to inter-operate with other applications within a simuPOP script. For example, users can call any R function from Python/simuPOP for the purposes of visualization and statistical analysis, using `R` and a Python module `RPy`. Because simuPOP utility modules such as `simuPOP.plotter` and `simuPOP.sampling` makes use of `R` and `rpy` (not `rpy2`) to plot figures, **it is hihgly recommended that you install R and RPy with simuPOP**. In addition, although simuPOP uses the standard `Tkinter` GUI toolkit when a graphical user interface is needed, it can make use of a `wxPython` toolkit if it is available.

1.5 How to read this user's guide

This user's guide describes all simuPOP features using a lot of examples. The first few chapters describes all classes in the simuPOP core. Chapter 4 describes almost all simuPOP operators, divided largely by genetic models. Features listed in these two chapters are generally implemented at the C++ level and are provided through the `simuPOP` module. Chapter 6 describes features that are provided by various simuPOP utility modules. These modules provide extensions to the simuPOP core that improves the usability and userfriendliness of simuPOP. The next chapter (Chapter 7) demonstrates how to write a script to solve a real-world simulation problem. Because some sections describe advanced features that are only used in the construction of highly complex simulations, or implementation details that concern only advanced users, new simuPOP users can safely skip these sections. **Sections that describe advanced topics are marked by one or two asterisks (*) after the section titles**.

simuPOP is a comprehensive forward-time population genetics simulation environment with many unique features. If you are new to simuPOP, you can go through this guide quickly and understand what simuPOP is and what features it provides. Then, you can read Chapter 7 and learn how to apply simuPOP in real-world problems. After you play with simuPOP for a while and start to write simple scripts, you can study relevant sections in details. The *simuPOP reference manual* will become more and more useful when the complexity of your scripts grows.

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

- All class names use the CapWords convention (e.g. `Population()`, `InitSex()`) .

- All standalone functions (e.g. `loadPopulation()` and `initSex()`), member functions (e.g. `Population.mergeSubPops()`) and parameter names use the mixedCases style.
- Constants are written in all capital characters with underscores separating words (e.g. `CHROMOSOME_X`, `UNIFORM_DISTRIBUTION`). Their names instead of their actual values should be used because those values can change without notice.
- simuPOP uses the abbreviated form of the following words in function and parameter names:

  `pop` (population), `pops` (populations), `pos` (position), `info` (information), `migr` (migration), `subPop` (subpopulation and virtual subpopulation), `subPops` (subpopulations and virtual subpopulations), `rep` (replicates), `gen` (generation), `ops` (operators), `expr` (expression), `stmts` (statements).
- simuPOP uses both singular and plural forms of parameters, according to the following rules:
  - If a parameter only accept a single input, singular names such as `field`, `locus`, `value`, and `name` are used.
  - If a parameter accepts a list of values, plural names such as `fields`, `loci`, `values` and `names` are used. **Such parameters usually accept single inputs.** For example, `loci=1` can be used as a shortcut for `loci=[1]` and `infoFields='x'` can be used as a shortcut for `infoFields=['x']`.

  The same rules also hold for function names. For example, `Population.addInfoFields()` accept a list of information fields but `pop.addInfoFields('field')` is also acceptable.

1.6 Other help sources

If you are new to Python, it is recommended that you borrow a Python book, or at least go through the following online Python tutorials:

1. The Python tutorial (`http://docs.python.org/tut/tut.html`)
2. Other online tutorials listed at `http://www.python.org/doc/`

If you are new to simuPOP, please read this guide before you dive into *the simuPOP reference manual*, which describes all the details of simuPOP but does not show you how to use them. Both documents are available online at `http://simupop.sourceforge.net` in both searchable HTML format and PDF format.

A *simuPOP online cookbook* (`http://simupop.sourceforge.net/cookbook`) is a wiki-based website where you can browse and download examples, functions and scripts for various simulation scenarios, and upload your own code snippets for the benefit of all simuPOP users. Please consider contributing to this cookbook if you have written some scripts that might be useful to others.

If you cannot find the answer you need, or if you believe that you have encountered a bug, or if you would like to request a feature, please subscribe to the simuPOP mailinglist (`simupop-list@lists.sourceforge.net`) and send your questions there.

Chapter 2

Loading and running simuPOP

2.1 Pythonic issues

2.1.1 `from simuPOP import *` v.s. `import simuPOP`

Generally speaking, it is recommended to use `import simuPOP` rather than `from simuPOP import *` to import a simuPOP module. That is to say, instead of using

```
from simuPOP import *
pop = Population(size=100, loci=[5])
simu = Simulator(pop, RandomMating())
```

it is recommended that you use simuPOP like

```
import simuPOP
pop = simuPOP.Population(size=100, loci=[5])
simu = simuPOP.Simulator(pop, simuPOP.RandomMating())
```

The major problem with `from simuPOP import *` is that it imports all simuPOP symbols to the global namespace and increases the likelihood of name clashes. For example, if you import a module `myModule` after simuPOP, which happens to have a variable named `MALE`, the following code might lead to a `TypeError` indicating your input for parameter sex is wrong.

```
from simuPOP import *
from myModule import *
pop = Population(size=100, loci=[5])
initSex(pop, sex=[MALE, FEMALE])
```

It can be even worse if the definition of `MALE` is changed to a different value of the same type (e.g. to `FEMALE`) and your simulation might produce erroranous result without a hint.

For the sake of brevity, all examples in this user's guide use `import simuPOP as sim` as an alternative form of the `import simuPOP` style. This saves some keystrokes by referring simuPOP functions as `sim.Population()` instead of `simuPOP.Population()`. Note that simuPOP has a number of submodules, which are not imported by default. The recommended syntax to load these modules is:

```
# import and use submodule simuPOP.utils
from simuPOP import utils
utils.simulateBackwardTrajectory(N=1000, endGen=100, endFreq=0.1)
```

2.1.2 References and the `clone()` member function

Assignment in Python only creates a new reference to an existing object. For example,

```
pop = Population()
pop1 = pop
```

creates a reference `pop1` to population `pop`. Modifying `pop1` will modify `pop` as well and the removal of `pop` will invalidate `pop1`. For example, a reference to the first Population in a simulator is returned from function `func()` in Example 2.1. The subsequent use of this `pop` object may crash simuPOP because the simulator `simu` is destroyed, along with all its internal populations, after `func()` is finished, leaving `pop` referring to an invalid object.

Listing 2.1: Reference to a population in a simulator

```
def func():
    simu = Simulator(Population(10), RandomMating(), rep=5)
    # return a reference to the first Population in the simulator
    return simu.population(0)

pop = func()
# simuPOP will crash because pop refers to an invalid Population.
pop.popSize()
```

If you would like to have an independent copy of a population, you can use the `clone()` member function. Example 2.1 would behave properly if the `return` statement is replaced by

```
return simu.population(0).clone()
```

although in this specific case, extracting the first population from the simulator using the `extract` function

```
return simu.extract(0)
```

would be more efficient.

The `clone()` function exists for all simuPOP classes (objects) such as *simulator*, *mating schemes* and *operators*. simuPOP also supports the standard Python shallow and deep copy operations so you can also make a cloned copy of `pop` using the `deepcopy` function defined in the Python `copy` module

```
import copy
pop1 = copy.deepcopy(pop)
```

2.1.3 Zero-based indexes, absolute and relative indexes

All arrays in simuPOP start at index 0. This conforms to Python and C++ indexes. To avoid confusion, I will refer the first locus as locus zero, the second locus as locus one; the first individual in a population as Individual zero, and so on.

Another two important concepts are the *absolute index* and *relative index* of a locus. The former index ignores chromosome structure. For example, if there are 5 and 7 loci on the first two chromosomes, the absolute indexes of the two chromosomes are (0, 1, 2, 3, 4), (5, 6, 7, 8, 9, 10, 11) and the relative indexes are (0, 1, 2, 3, 4), (0, 1, 2, 3, 4, 5, 6). Absolute indexes are more frequently used because they avoid the trouble of having to use two numbers (chrom, index) to refer to a locus. Two functions `chromLocusPair(idx)` and `absLocusIndex(chrom,index)` are provided to convert between these two kinds of indexes. An individual can also be referred by its *absolute index* and *relative index* where *relative index* is the index in its subpopulation. Related member functions are `subPopIndPair(idx)` and `absIndIndex(idx, subPop)`. Example 2.2 demonstrates the use of these functions.

Listing 2.2: Conversion between absolute and relative indexes

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[10, 20], loci=[5, 7])
>>> print(pop.chromLocusPair(7))
(1L, 2L)
>>> print(pop.absLocusIndex(1, 1))
6
>>> print(pop.absIndIndex(2, 1))
12
>>> print(pop.subPopIndPair(25))
(1L, 15L)
```

2.1.4 Ranges and iterators

Ranges in simuPOP also conform to Python ranges. That is to say, a range has the form of `[a,b)` where `a` belongs to the range, and `b` does not. For example, `pop.chromBegin(1)` refers to the index of the first locus on chromosome 1 (actually exists), and `pop.chromEnd(1)` refers to the index of the last locus on chromosome 1 **plus 1**, which might or might not be a valid index.

A number of simuPOP functions return Python iterators that can be used to iterate through an internal array of objects. For example, `Population.Individuals([subPop])` returns an iterator iterates through all individuals, or all individuals in a (virtual) subpoulation. `Simulator.populations()` can be used to iterate through all populations in a simulator. Example 2.3 demonstrates the use of ranges and iterators in simuPOP.

Listing 2.3: Ranges and iterators

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=2, loci=[5, 6])
>>> sim.initGenotype(pop, freq=[0.2, 0.3, 0.5])
>>> for ind in pop.individuals():
...     for loc in range(pop.chromBegin(1), pop.chromEnd(1)):
...         print(ind.allele(loc))
...
0
2
2
1
1
1
1
2
2
2
1
2
```

2.1.5 Empty, `ALL_AVAIL` and dynamic values for parameters `loci`, `reps`, `ancGen` and `subPops`

Parameters `loci`, `reps` and `subPops` are widely used in simuPOP to specify which loci, replicates, ancestral generations, or (virtual) subpulations a function or operator is applied to. These parameter accepts a list of indexes such as `[1, 2]`, names such as `['a', 'b']`, and take single form inputs (e.g. `loci=1` is equivalent to `loci=[1]`). For example,

- `Recombinator(loci=[])` recombine at no locus, and
- `Recombinator(loci=1)` recombine at locus 1

- `Recombinator(loci=[1,2,4])` recombine at loci 1, 2, and 4
- `Recombinator(loci=[('1', 20), ('1', 25)])` recombine at loci with position `20` and `25` on chromosome `1`. This usage is only available for parameter `loci`.
- `Recombinator(loci=['a2', 'a4'])` recombine at loci `'a2'` and `'a4'`.

The last method is easier to understand in some cases. Moreover, when you use loci names instead of indexes in an operator, this operator can be applied to populations with loci at different locations. For example

```
MaSelector(loci='a2', fitness=[1,1.01,1.02])
```

will be applied to locus `a2` regardless the actual location of this locus in the population to which this operator is applied.

However, in the majority of the cases, these parameters take a default value `ALL_AVAIL` which applies the function or operator to all available loci, replicates or subpopulations. That is to say, `Recombinator()` or `Recombinator(loci=ALL_AVAIL)` will recombine at all applicable loci, which will vary from population to population. Value `UNSPECIFIED` is sometimes used as default parameter of these parameters, indicating that no value has been specified. Similarly, `subPops=[0, 'Male']` can be used to refer a virtual subpopulation with name `'Male'`, regardless its virtual subpopulation index.

Besides `subPops=ALL_AVAIL`, which means `subPops=[0,1,2,3]` for a population with 4 subpopulations, `ALL_AVAIL` could also be used as `subPops=[(ALL_AVAIL, 1)]` to specify a specific virtual subpopulation for all subpopulations, or `subPops=[(1, ALL_AVAIL)]` or even `subPops=[(ALL_AVAIL, ALL_AVAIL)]` to specify all virtual subpopulations in specified or all subpopulations. This becomes handy when you, for example, would like to list all male individuals in a population, regardless of number of subpopulations.

2.1.6 User-defined functions and class `WithArgs` *

Some simuPOP objects call user-defined functions to perform customized operations. For example, a penetrance operator can call a user-defined function with genotype at specified loci and use its return value to determine the affection status of an individual.

simuPOP uses parameter names to determine which information should be passed to such a function. For example, a `PyOperator` will pass a reference to each offspring to a function defined with parameter `off` (e.g. `func(off)`) and references to offspring and his/her parents to a function defined with parameters `off`, `dad`, and `mom` (e.g. `func(off, dad, mom)`). For example, Example 2.4 defines a function `func(geno, smoking)` using parameters `geno` and `smoking` so operator `PyPenetrance` will pass genotype at specified loci and value at information field `smoking` to this function.

Listing 2.4: Use of user-defined Python function in simuPOP

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(1000, loci=1, infoFields='smoking')
>>> sim.initInfo(pop, lambda:random.randint(0,1), infoFields='smoking')
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>>
>>> # a penetrance function that depends on smoking
>>> def func(geno, smoking):
...     if smoking:
...         return (geno[0]+geno[1])*0.4
...     else:
...         return (geno[0]+geno[1])*0.1
...
>>> sim.pyPenetrance(pop, loci=0, func=func)
>>> sim.stat(pop, numOfAffected=True)
>>> print(pop.dvars().numOfAffected)
342
>>>
```

However, there are circumstances that you do not know the number or names of parameters in advance so it is difficult to define such a function. For example, your function may use an information field with programmed name `'off'+str(numOffspring)` where `numOffspring` is a parameter. In this case, you can create a wrapper function object using `WithArgs(func, args)` and list passed arguments in `args` (e.g. `WithArgs(func, args=['geno', 'off' + str(numOffspring)]`). As long as simuPOP knows which arguments to pass, your function can be defined in any format you want (e.g. use *args parameters). Example 2.5 provides such an example.

Listing 2.5: Specify arguments of user-provided function using function WithArgs

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(1000, loci=1, infoFields=('x', 'y'))
>>> sim.initInfo(pop, lambda:random.randint(0,1), infoFields=('x', 'y'))
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>>
>>> # a penetrance function that depends on unknown information fields
>>> def func(*fields):
...     return 0.4*sum(fields)
...
>>> # function WithArgs tells PyPenetrance that func accepts fields x, y so that
>>> # it will pass values at fields x and y to func.
>>> sim.pyPenetrance(pop, loci=0, func=sim.WithArgs(func, pop.infoFields()))
>>> sim.stat(pop, numOfAffected=True)
>>> print(pop.dvars().numOfAffected)
414
```

2.1.7 Exception handling *

As shown in Examples 2.6 and 2.7, optimized modules raise less exceptions than standard modules. More specifically, the standard modules check for invalid inputs frequently and raise exceptions (e.g. out of bound loci indexes). In constrast, the optimized modules only raise exceptions where proper values could not be pre-determined (e.g. looking for an individual in a population with an ID). **Only exceptions that are raised in both types of modules are documented in the simuPOP reference manual**.

Generally speaking, **you should avoid using exceptions to direct the logic of your script** (e.g. use a `try ... except ...` statement around a function to find a valid input value). Because the optimized modules might not raise these exceptions, such a script may crash or yield invalid results when an optimized module is used. If you have to use such a structure, please check the reference manual and see whether or not an exception will be raised in optimized modules.

2.2 Loading simuPOP modules

2.2.1 Short, long, binary, mutant and lineage modules and their optimized versions

There are ten flavors of the core simuPOP module: short, long, binary, mutant, and lineage allele modules, and their optimized versions.

- The short allele modules use *8 bits* to store each allele which limits the possible allele states to 256. This is enough most of the times so this is the default module of simuPOP.
- If you need to a large number of allele states to simulate, for example the infinite allele model, you should use the long allele version of the modules, which use *32 or 64 bits* for each allele and can have 2^{32} or 2^{64} possible allele states depending on your platform.

- If you would like to simulate a large number of binary (SNP) markers, binary libraries can save you a lot of RAM because they use *1 bit* for each allele.
- If you are simulating long sequence regions with rare variants, you can use the mutant module. This module uses compression technology that ignores wildtype alleles and is not efficient if you need to traverse all alleles frequently. The maximum allele state is 255 for this module. Because this module stores location and value of each allele, it uses at least 64 + 8 bits for each allele on a 64 bit system. The complexity of the storage also prevents simultaneous write access to genotypes so this module does not benefit much from running in multi-thread mode.
- If you are interested in tracing the lineage of each allele (e.g. the ID of individuals to whom the allele was introduced), you can use the lineage module for which each allele is attached with information about its origin. The maximum allele state is 255 for this module, and the cost of storing each allele is 8 (value) + 32 (lineage) bits.

Despite of differences in internal memory layout, all these modules have the same interface, although some functions behave differently in terms of functionality and performance.

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

Examples 2.6 and 2.7 demonstrate the differences between standard and optimized modules, by executing two invalid commands. A standard module checks all input values and raises exceptions when invalid inputs are detected. An interactive Python session would catch these exceptions and print proper error messages. In constrast, an optimized module returns erroneous results and or simply crashes when such inputs are given.

Listing 2.6: Use of standard simuPOP modules

```
>>> import simuPOP as sim
>>> pop = sim.Population(10, loci=2)
>>> pop.locusPos(10)
Traceback (most recent call last):
  File "/var/folders/ys/gnzk0qbx5wbdgm531v82xxljv5yqy8/T/tmpfGL_Ei", line 1, in <module>
    #begin_file log/standard.py
IndexError: genoStru.h: 561 absolute locus index (10) out of range of 0 ~ 1
>>> pop.individual(20).setAllele(1, 0)
Traceback (most recent call last):
  File "/var/folders/ys/gnzk0qbx5wbdgm531v82xxljv5yqy8/T/tmpfGL_Ei", line 1, in <module>
    #begin_file log/standard.py
IndexError: population.h: 573 individual index (20)  out of range of 0 ~ 9
```

Example 2.7 also demonstrates how to use the `setOptions` function in the `simuOpt` module to control the choice of one of the six simuPOP modules. By specifying one of the values `short`, `long` or `binary` for option `alleleType`, and setting `optimized` to `True` or `False`, the right flavor of module will be chosen when simuPOP is loaded. In addition, option `quiet` can be used suppress the banner message when simuPOP is loaded. An alternative method is to set environmental variable `SIMUALLELETYPE` to `short`, `long` or `binary` to use the standard short, long or binary module, and variable `SIMUOPTIMIZED` to use the optimized modules. Command line options `--optimized` can also be used.

Listing 2.7: Use of optimized simuPOP modules

```
% python
>>> from simuOpt import setOptions
>>> setOptions(optimized=True, alleleType='long', quiet=True)
>>> import simuPOP as sim
```

```
>>> pop = sim.Population(10, loci=[2])
>>> pop.locusPos(10)
1.2731974748756028e-313
>>> pop.individual(20).setAllele(1, 0)
Segmentation fault
```

2.2.2 Execution in multiple threads

simuPOP is capable of executing in multiple threads but it by default only makes use of one thread. If you have a multi-core CPU, it is often beneficial to set the number of threads to 2 or more to take advantage of this feature. The recommended number of threads is usually the number of cores of your CPU but you might want to set it to a lower number to leave room for the execution of other applications. The number of threads used in simuPOP can be controlled in the following ways:

- If an environmental variable `OMP_NUM_THREADS` is set to a positive number, simuPOP will be started with specified number of threads.
- Before simuPOP is imported, you can set the number of threads using function `simuOpt.setOptions(numThreads=x)` where `x` can be a positive number (number of threads) or `0`, which is intepreted as the number of cores available for your computer.

The number of threads a simuPOP session is used will be displayed in the banner message when simuPOP is imported, and can be retrieved through `moduleInfo()['threads']`.

Although simuPOP can usually benefit from the use of multiple cores, certain features of your script might prevent the execution of simuPOP in multiple threads. For example, if your script uses a sex mode of `GLOBAL_SEX_SEQUENCE` to set the sex of offspring according to the global sequence of sexes (e.g. male, male, female), simuPOP will only use on thread to generate offspring because it is not feasible to assign individual sex from a single source of list across multiple threads.

2.2.3 Graphical user interface

A complete graphical user interface (GUI) for users to interactively construct evolutionary processes is still in the planning stage. However, some simuPOP classes and functions can make use of a GUI to improve user interaction. For example, a parameter input dialog can be constructed automatically from a parameter specification list, and be used to accept user input if class `simuOpt.Params` is used to handle parameters. Other examples include a progress bar `simuPOP.utils.ProgressBar` and a dialog used by function `simuPOP.utils.viewVars` to display a large number of variables. The most notable feature of the use of GUI in simuPOP is that **all functionalities can be achieved without a GUI**. For examples, `simuOpt.getParam()` will use a terminal to accept user input interactively and `simuPOP.utils.ProgressBar` will turn to a text-based progress bar in the non-GUI mode.

The use of GUI can be controlled either globally or Individually. First, a global GUI parameter could be set by environmental variable `SIMUGUI`, function `simuOpt.setOptions(gui)` or a command line option `--gui` of a simuPOP scripts. Allowed values include

- `True`: This is the system default value. A GUI is used whenever possible. All GUI-capable functions support `wxPython` so a `wxPython` dialog will be used if `wxPython` is available. Otherwise, `tkInter` based dialogs or text-mode will be used.
- `False`: no GUI will be used. All functions will use text-based implementation. Note that `--gui=False` is commonly used to run scripts in batch mode.
- `wxPython`: Force the use of `wxPython` GUI toolkit.

- `Tkinter`: Force the use of `Tkinter` GUI toolkit.

Individual classes and functions that could make use a GUI usually have their own `gui` parameters, which can be set to override global GUI settings. For example, you could force the use of a text-based progress bar by using `ProgressBar(gui=False)`.

2.3 Online help system

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

Listing 2.8: Getting help using the `help()` function

```
>>> import simuPOP as sim
>>> help(sim.Population.addInfoFields)
Help on method Population_addInfoFields in module _simuPOP_std:

Population_addInfoFields(...) unbound simuPOP.simuPOP_std.Population method
    Usage:

        x.addInfoFields(fields, init=0)

    Details:

        Add a list of information fields fields to a population and
        initialize their values to init. If an information field alreay
        exists, it will be re-initialized.
```

It is important that you understand that

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

  ```
  >>> help(Population.__init__)
  ```

- Some classes are derived from other classes and have access to member functions of their base classes. For example, class `Population` and `Individual` are both derived from class `GenoStruTrait`. Therefore, you can use all `GenoStruTrait` member functions from these classes.

  In addition, the constructor of a derived class also calls the constructor of its base class so you may have to refer to the base class for some parameter definitions. For example, parameters `begin, end, step, at` etc are shared by all operators, and are explained in details only in class `BaseOperator`.

2.4 Debug-related functions and operators *

Debug information can be useful when something looks suspicious. By turnning on certain debug code, simuPOP will print out some internal information before and during evolution. Functions `turnOnDebug(code)` and `turnOffDebug(code)` could be used to turn on and off some debug information.

For example, the following code might crash simuPOP:

```
>>> Population(1, loci=[100]).individual(0).genotype()
```

It is unclear why this simple command causes us trouble, instead of outputting the genotype of the only Individual of this population. However, the reason is clear if you turn on debug information:

Listing 2.9: Turn on/off debug information

```
>>> turnOnDebug(DBG_POPULATION)
>>> Population(1, loci=100).individual(0).genotype()
Constructor of population is called
Destructor of population is called
Segmentation fault (core dumped)
```

`Population(1, loci=[100])` creates a temporary object that is destroyed right after the execution of the command. When Python tries to display the genotype, it will refer to an invalid location. The correct method to print the genotype is to create a persistent population object:

```
>>> pop = Population(1, loci=[100])
>>> pop.individual(0).genotype()
```

Another useful debug code is `DBG_WARNING`. When this code is set, it will output warning messages for some common misuse of simuPOP. For example, it will warn you that population object returned by function `Simulator.population()` is a temporary object that will become invalid once a simulator is changed. If you are new to simuPOP, it is recommended that you use

```
import simuOpt
simuOpt.setOptions(optimized=False, debug='DBG_WARNING')
```

when you develop your script.

Besides functions `turnOnDebug(code)` and `turnOffDebug(code)`, you can set environmental variable `SIMUDEBUG=code` where `code` is a comma separated debug codes. A list of valid debug code could be found in function `moduleInfo()['debug']`. Note that debug information is only available in standard (non-optimized) modules.

The amount of output can be overwhelming in some cases which makes it necessary to limit the debug information to certain generations, or triggered by certain conditions. In addition, debugging information may interfere with your regular output so you may want to direct such output to another destination, such as a dedicated file.

Example 2.10 demonstrates how to turn on debug information conditionally and turn it off afterwards, using operator `PyOperator`. It also demonstrates how to redirect debug output to a file but redefining system standard error output. Note that "`is None`" is used to make sure the lamdba functions return `True` so that the evolutionary process can continue after the python operator.

Listing 2.10: Turn on and off debug information during evolution.

```
>>> import simuPOP as sim
>>> # redirect system stderr
>>> import sys
>>> debugOutput = open('debug.txt', 'w')
>>> old_stderr = sys.stderr
>>> sys.stderr = debugOutput
>>> # start simulation
>>> simu = sim.Simulator(sim.Population(100, loci=1), rep=5)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.1, 0.9])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
```

```
...         sim.IfElse('alleleNum[0][0] == 0',
...             ifOps=[
...                 # the is None part makes the function return True
...                 sim.PyOperator(lambda : sim.turnOnDebug("DBG_MUTATOR") is None),
...                 sim.PointMutator(loci=0, allele=0, inds=0),
...             ],
...             elseOps=sim.PyOperator(lambda : sim.turnOffDebug("DBG_MUTATOR") is None)),
...     ],
...     gen = 100
... )
(100L, 100L, 100L, 100L, 100L)
>>> # replace standard stdandard error
>>> sys.stderr = old_stderr
>>> debugOutput.close()
>>> print(''.join(open('debug.txt').readlines()[:5]))
Mutate locus 0 at ploidy 0 to allele 0 at generation 12
Mutate locus 0 at ploidy 0 to allele 0 at generation 13
Mutate locus 0 at ploidy 0 to allele 0 at generation 15
Mutate locus 0 at ploidy 0 to allele 0 at generation 16
Mutate locus 0 at ploidy 0 to allele 0 at generation 21
```

2.5 Random number generator *

When simuPOP is loaded, it creates a default random number generator (`RNG`) of type `mt19937` for each thread. It uses a random seed for the first RNG and uses seeds derived from the first seed to initialize RNGs for other threads. The seed is drawn from a system random number generator that guarantees random seeds for all instances of simuPOP even if they are initialized at the same time. After simuPOP is loaded, you can reset this system RNG with a different random number generator (c.f. `moduleInfo()['availableRNGs']`) or use a specified seed using function , `setRNG(name, seed)`.

`getRNG().seed()` returns the seed of the simuPOP random number generator. It can be used to replay your simulation if `getRNG()` is your only source of random number generator. If you also use the Python `random` module, it is a good practise to set its seed using `random.seed(getRNG().seed())`. Example 2.11 demonstrates how to use these functions to replay an evolutionary process. simuPOP uses a single seed to initialize multiple random number generators used for different threads (seeds for other threads are determined from the first seed) so you only need to save the head seed (`getRNG().seed()`)

Listing 2.11: Use saved random seed to replay an evolutionary process

```
>>> import simuPOP as sim
>>> import random
>>> def simulate():
...     pop = sim.Population(1000, loci=10, infoFields='age')
...     pop.evolve(
...         initOps=[
...             sim.InitSex(),
...             sim.InitGenotype(freq=[0.5, 0.5]),
...             sim.InitInfo(lambda: random.randint(0, 10), infoFields='age')
...         ],
...         matingScheme=sim.RandomMating(),
...         finalOps=sim.Stat(alleleFreq=0),
...         gen=100
...     )
...     return pop.dvars().alleleFreq[0][0]
...
```

```
>>> seed = sim.getRNG().seed()
>>> random.seed(seed)
>>> print('%.4f' % simulate())
0.5780
>>> # will yield different result
>>> print('%.4f' % simulate())
0.6355
>>> sim.setRNG(seed=seed)
>>> random.seed(seed)
>>> # will yield identical result because the same seeds are used
>>> print('%.4f' % simulate())
0.5780
```

Chapter 3

Individuals and Populations

3.1 Genotypic structure

Genotypic structure refers to structural information shared by all individuals in a population, including number of homologous copies of chromosomes (c.f. `ploidy(), ploidyName()`), chromosome types and names (c.f. `numChrom(), chromType(), chromName()`), position and name of each locus (c.f. `numLoci(ch), locusPos(loc), locusName(loc)`), and axillary information attached to each individual (c.f. `infoField(idx), infoFields()`). In addition to property access functions, a number of utility functions are provided to, for example, look up the index of a locus by its name (c.f. `locusByName()`, `chromBegin()`, `chromLocusPair()`).

In simuPOP, locus is a (named) position and alleles are just different numbers at that position. **A locus can be a gene, a nucleotide, or even a deletion, depending on how you define alleles and mutations**. For example,

- A codon can be simulated as a locus with 64 allelic states, or three locus each with 4 allelic states. Alleles in the first case would be codons such as `AAC` and a mutation event would mutate one codon to another (e.g. `AAC -> ACC`). Alleles in the second case would be `A`, `C`, `T` or `G`, and a mutation event would mutate one nucleotide to another (e.g. `A -> G`).
- You can use 0 and 1 (and the binary module of simuPOP) to simulate SNP (single-nucleotide polymorphism) markers and ignore the exact meaning of 0 and 1, or use 0, 1, 2, 3 to simulate different nucleotide (A, C, T, or G) in these markers. The mutation model in the second case would be more complex.
- For microsatellite markers, alleles are usually interpreted as the number of tandem repeats. It would be difficult (though doable) to simulate the expansion and contraction of genome caused by the mutation of microsatellite markers.
- The infinite site and infinite allele mutation models could be simulated using either a continuous sequence of nucleotides with a simple 2-allele mutation model, or a locus with a large number of possible allelic states. It is also possible to simulate an empty region (without any locus) with loci introduced by mutation events.
- If you consider deletion as a special allelic state, you can simulate gene deletions without shrinking a chromosome. For example, a deletion mutation event can set the allelic state of one or more loci to 0, which can no longer be mutated.
- Alleles in different individuals could be interpretted differently. For example, if you would like to simulate major chromosomal mutations such as inversion, you could use a super set of markers for different types of chromosomes and use an indicator (information field) to mark the type of chromosome and which markers are valid. Using virtual subpopulations, these individuals could be handled differently during mating.
- In an implementation of an infinite-sites model, **Individual loci are used to store mutation events**. In this example (Example 4.43), 100 loci are allocated for each individual and they are used to store mutation events

(location of a mutation) that happens in a 10Mb region. Whenever a mutation event happens, its location is stored as an allele of an individual. At the end of the evolution, each individual has a list of mutation events which can be readily translated to real alleles. Similar ideas could be used to simulate the accumulation of recombination events.

In summary, the exact meaning of loci and their alleles are user defined. With appropriate mutation model and mating scheme, it is even possible to simulate phenotypic traits using this mechanism, although it is more natual to use information fields for quatitative traits.

A genotypic structure can be retrieved from *Individual* and *Population* objects. **Because a population consists of individuals of the same type, genotypic information can only be changed for all individuals at the population level**. populations in a simulator usually have the same genotypic structure because they are created by as replicates, but their structure may change during evolution. Example 3.1 demonstrates how to access genotypic structure functions at the population and individual levels. Note that `lociPos` determines the order at which loci are arranged on a chromosome. Loci positions and names will be rearranged if given `lociPos` is unordered.

Listing 3.1: Genotypic structure functions

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[2, 3], ploidy=2, loci=[5, 10],
...     lociPos=list(range(0, 5)) + list(range(0, 20, 2)), chromNames=['Chr1', 'Chr2'],
...     alleleNames=['A', 'C', 'T', 'G'])
>>> # access genotypic information from the sim.Population
>>> pop.ploidy()
2
>>> pop.ploidyName()
'diploid'
>>> pop.numChrom()
2L
>>> pop.locusPos(2)
2.0
>>> pop.alleleName(1)
'C'
>>> # access from an individual
>>> ind = pop.individual(2)
>>> ind.numLoci(1)
10L
>>> ind.chromName(0)
'Chr1'
>>> ind.locusName(1)
''
>>> # utility functions
>>> ind.chromBegin(1)
5L
>>> ind.chromByName('Chr2')
1L
>>> # loci pos can be unordered within each chromosome
>>> pop = sim.Population(loci=[2, 3], lociPos=[3, 1, 1, 3, 2],
...     lociNames=['loc%d' % x for x in range(5)])
>>> pop.lociPos()
(1.0, 3.0, 1.0, 2.0, 3.0)
>>> pop.lociNames()
('loc1', 'loc0', 'loc2', 'loc4', 'loc3')
```

Note: simuPOP does not assume any unit for loci positions. Depending on your application, it can be basepair (bp), kilo-basepair (kb), mega base pair (mb) or even using genetic-map distance such as centiMorgan. It is your

responsibility to interpret and use loci positions properly. For example, recombination rate between two adjacent markers can be specified as the product between their physical distance and a recombination intensity. The scale of this intensity will vary by the unit assumed.

Note: Names of loci, alleles and subpopulations are optional. Empty names will be used if they are not specified. Whereas `locusName`, `subPopName` and `alleleName` always return a value (empty string or specified value) for any locus, subpopulation or allele, respectively, `lociNames`, `subPopNames` and `alleleNames` only return specified values, which can be empty lists.

3.1.1 Haploid, diploid and haplodiploid populations

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

For efficiency considerations, simuPOP saves the same numbers of homologous sets of chromosomes even if some individuals have different numbers of homologous sets in a population. For example, in a haplodiploid population, because male individuals have only one set of chromosomes, their second homologous set of chromosomes are *unused*, which are labeled as `'_'`, as shown in Example 3.2.

Listing 3.2: An example of haplodiploid population

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[2,5], ploidy=sim.HAPLODIPLOID, loci=[3, 5])
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>> sim.dump(pop)
Ploidy: 2 (haplodiploid)
Chromosomes:
1:  (AUTOSOME, 3 loci)
   (1),  (2),  (3)
2:  (AUTOSOME, 5 loci)
   (1),  (2),  (3),  (4),  (5)
population size: 7 (2 subpopulations with 2, 5 Individuals)
Number of ancestral populations: 0

SubPopulation 0 (), 2 Individuals:
   0: MU 111 00001 | ___ _____
   1: MU 111 01110 | ___ _____
SubPopulation 1 (), 5 Individuals:
   2: MU 111 11110 | ___ _____
   3: MU 101 11111 | ___ _____
   4: MU 110 11111 | ___ _____
   5: MU 101 11101 | ___ _____
   6: MU 110 11001 | ___ _____
```

3.1.2 Autosomes, sex chromosomes, mitochondrial, and other types of chromosomes *

The default chromosome type is autosome, which is the *normal* chromosomes in diploid, and in haploid populations. simuPOP supports four other types of chromosomes, namely *chromosome X*, *chromosome Y, mitochondrial,* and *customized* chromosome types. Sex chromosomes are only valid in haploid populations where chromosomes X and Y are used to determine the sex of an offspring. Mitochondrial DNAs can exists in haploid or diploid populations, and are

inherited maternally. Customized chromosomes rely on user defined functions and operators to be passed from parents to offspring.

Example 3.8 shows how to specify different chromosome types, and how genotypes of these special chromosomes are arranged.

Listing 3.3: Different chromosome types

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=6, ploidy=2, loci=[3, 3, 3, 2, 2, 4, 4],
...     chromTypes=[sim.AUTOSOME]*2 + [sim.CHROMOSOME_X, sim.CHROMOSOME_Y, sim.MITOCHONDRIAL]
...         + [sim.CUSTOMIZED]*2)
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>> sim.dump(pop, structure=False) # does not display genotypic structure information
SubPopulation 0 (), 6 Individuals:
   0: MU 111 000 011 __ 11 1111 1101 | 110 000 ___ 11 __ 1111 1011
   1: MU 111 111 101 __ 11 1110 1011 | 111 011 ___ 11 __ 1110 1011
   2: MU 110 101 011 __ 11 1011 0011 | 110 100 ___ 11 __ 1010 1111
   3: MU 010 011 111 __ 11 1111 1111 | 110 010 ___ 11 __ 1111 0111
   4: MU 101 000 111 __ 01 0111 0100 | 110 111 ___ 00 __ 0111 0001
   5: MU 111 010 111 __ 10 0111 1011 | 111 111 ___ 11 __ 0111 1011
```

The evolution of sex chromosomes follow the following rules

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

The evolution of mitochonrial DNAs follow the following rules

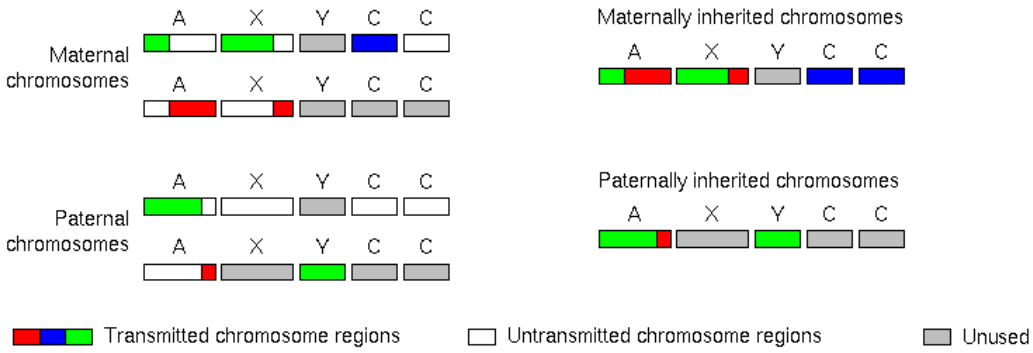
- There can be only one copy of mitochondrial DNA, exists for both males and females.
- In a non-haploid population where all chromosomes have multiple homologous copies, only the first copy is used for mitochondrial DNA.
- mtDNAs are inherited maternally

Customized chromosomes are used to model more complex types of chromosomes. They rely on customized operators for inheritence. For example, if you would like to model multiple copies of mitochondrial DNAs (cytohets with multiple organellar chromosomes) in a cell, and the process of genetic drift of somatic cytoplasmic segregation of mtDNAs, you can use multiple customized chromosomes to model multiple cytohets (see section 5.3.3 for an Example). Figure 3.1 depicts the possible chromosome structure of two diploid parents, and how offspring chromosomes are formed. It uses two customized chromosomes to model multiple copies of mitochondrial chromosomes that are passed randomly from mother to offspring. The second homologous copy of customized chromosomes are unused in this example.

3.1.3 Information fields

Different kinds of simulations require different kinds of individuals. individuals with only genotype information are sufficient to simulate the basic Wright-Fisher model. Sex is needed to simulate such a model in diploid populations

Figure 3.1: Inheritance of different types of chromosomes in a diploid population



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

with sex. individual fitness may be needed if selection is induced, and age may be needed if the population is age-structured. In addition, different types of quantitative traits or affection status may be needed to study the impact of genotype on Individual phenotype. Because it is infeasible to provide all such information to an individual, simuPOP keeps genotype, sex (`MALE` or `FEMALE`) and affection status as *built-in properties* of an individual, and all others as optional *information fields* (float numbers) attached to each individual.

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

Listing 3.4: Basic usage of information fields

```
>>> import simuPOP as sim
>>> pop = sim.Population(10, loci=[20], ancGen=1,
...     infoFields=['father_idx', 'mother_idx'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[0]*20+[1]*20)
...     ],
...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.Recombinator(rates=0.01),
...             sim.ParentsTagger()
...         ]
...     ),
...     gen = 1
... )
1L
```

```
>>> pop.indInfo('mother_idx')  # mother of all offspring
(9.0, 8.0, 8.0, 0.0, 8.0, 9.0, 8.0, 7.0, 7.0, 9.0)
>>> ind = pop.individual(0)
>>> mom = pop.ancestor(ind.mother_idx, 1)
>>> print(ind.genotype(0))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> print(mom.genotype(0))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> print(mom.genotype(1))
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Example 3.4 demonstrates the basic usage of information fields. In this example, a population with two information fields `mother_idx` and `father_idx` are created. Besides the present generation, this population keeps one ancestral generations (`ancGen=1`, see Section 3.3.7 for details). After initializing each individual with two chromosomes with all zero and all one alleles respectively, the population evolves one generation, subject to recombination at rate 0.01. Parents of each individual are recorded, by operator `ParentsTagger`, to information fields `mother_idx` and `father_idx` of each offspring.

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

Information fields can only be added or removed at the population level because all individuals need to have the same set of fields. Values of information fields could be accessed at Individual or population levels, using functions such as `Individual.info`, `Individual.setInfo`, `population.indInfo`, `Population.setIndInfo`. These functions will be introduced in their respective classes.

Note: Information fields can be located both by names and by indexes**,** the former provides better readability at a slight cost of performance because these names have to be translated into indexes each time. However, use of names are recommended in most cases for readability considerations.

3.2 Individual

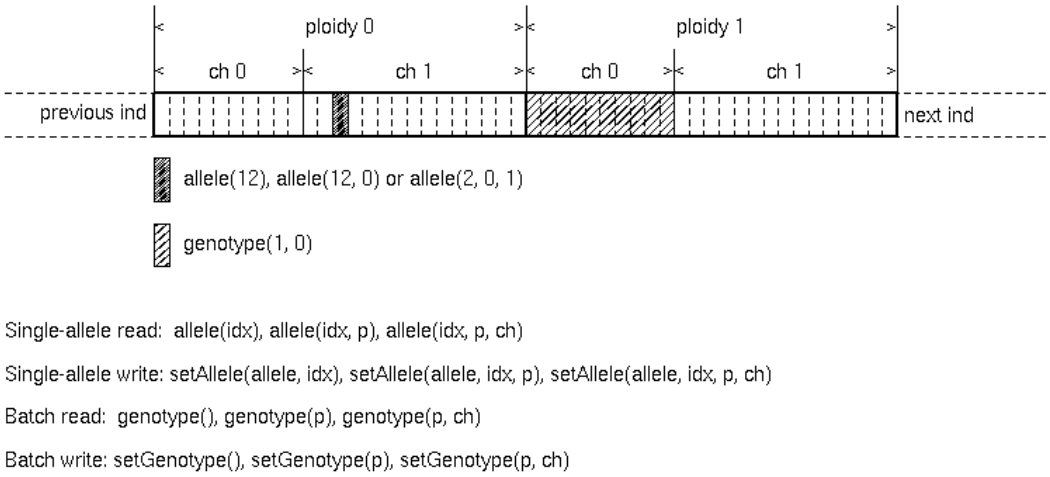
individuals are building blocks of a population. An individual object cannot be created independently, but references to inidividuals can be retrieved using member functions of a population object.

3.2.1 Access individual genotype

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 markers on the second and later chromosomes, followed by alleles on the second homologous set of the chromosomes for a diploid individual. A consequence of this memory layout is that alleles at the same locus of a non-haploid individual are separated by `Individual.totNumLoci()` loci. The memory layout of a diploid individual with two chromosomes is illustrated in Figure 3.2.

simuPOP provides several functions to read/write individual genotype. For example, `Individual.allele()` and `Individual.setAllele()` can be used to read and write single alleles. You could also access alleles in batch mode using functions `Individual.genotype()` and `Individual.setGenotype()`. It is worth noting that, instead of copying genotypes of an individual to a Python tuple or list, the return value of function `genotype([p, [ch]])` is a special python carray object that reflects the underlying genotypes. This object behaves like a regular Python list except that the underlying genotype will be changed if elements of this object are changed. Only `count(x)` and `index(x, [start, [stop]])` member functions can be used, but all comparison, assignment and slice operations are allowed. If you would like to

Figure 3.2: Memory layout of individual genotype



copy the content of this `carray` to a Python list, use the `list()` function. Example 3.5 demonstrates the use of these functions.

Listing 3.5: Access individual genotype

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

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

The same object will also be returned by function `Population.genotype()`.

3.2.2 individual sex, affection status and information fields

In addition to structural information shared by all individuals in a population, the individual class provides member functions to get and set *genotype*, *sex*, *affection status* and *information fields* of an individual. Example 3.6 demonstrates how to access and modify individual sex, affection status and information fields. Note that **information fields can be accessed as attributes of individuals**. For example, `ind.info('father_idx')` is equivalent to `ind.father_idx` and `ind.setInfo(35, 'age')` is equivalent to `ind.age = 35`.

Listing 3.6: Access Individual properties

```
>>> import simuPOP as sim
>>> pop = sim.Population([5, 4], loci=[2, 5], infoFields='x')
>>> # get an individual
>>> ind = pop.individual(3)
>>> ind.ploidy()            # access to genotypic structure
2
>>> ind.numChrom()
2L
>>> ind.affected()
False
>>> ind.setAffected(True)   # access affection sim.status,
>>> ind.sex()               # sex,
1
>>> ind.setInfo(4, 'x')     # and information fields
>>> ind.x = 5               # the same as ind.setInfo(4, 'x')
>>> ind.info('x')           # get information field x
5.0
>>> ind.x                   # the same as ind.info('x')
5.0
```

3.3 Population

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

3.3.1 Access and change individual genotype

From a user's point of view, genotypes of all individuals in a population are arranged sequentially. Similar to functions `Individual.genotype()` and `Individual.setGenotype()`, genotypes of a population can be accessed in batch using functions `Population.genotype()` and `Population.setGenotype()`. However, because it is error prone to locate an allele of a particular individual in this long array, these functions are usually used to perform population-level genotype operations such as clearing all alleles (e.g. `pop.setGenotype(0)`) or counting the number of a particular allele across all individuals (e.g. `pop.genotype().count(1)`).

Another way to change alleles across the whole population is to recode existing alleles to other numbers. This is sometimes needed if you need to change allele states to conform with a particular mutation model, assumptions of other software applications or genetic samples. For example, if your dataset uses 1, 2, 3, 4 for A, C, T, G alleles, and you would like to use alleles 0, 1, 2 and 3 for A, C, G, T (a convention for simuPOP when nucleotide mutation models are involved), you can use

```
pop.recodeAlleles([0, 0, 1, 3, 2], alleleNames=['A', 'C', 'G', 'T'])
```

to convert and rename the alleles (1 allele to 0, 2 allele to 1, etc). This operation will be applied to all subpopulations for all ancestral generations, but can be restricted to selected loci.

3.3.2 Subpopulations

A simuPOP population consists of one or more subpopulations. **If a population is not structured, it has one subpopulation that is the population itself.** Subpopulations serve as barriers of individuals in the sense that mating only happens between individuals in the same subpopulation. A number of functions are provided to merge, remove, resize subpopulations, and move individuals between subpopulations (migration).

Example 3.8 demonstrates how to use some of the subpopulation related functions.

Listing 3.7: Manipulation of subpopulations

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[3, 4, 5], ploidy=1, loci=1, infoFields='x')
>>> # individual 0, 1, 2, ... will have an allele 0, 1, 2, ...
>>> pop.setGenotype(range(pop.popSize()))
>>> #
>>> pop.subPopSize(1)
4L
>>> # merge subpopulations
>>> pop.mergeSubPops([1, 2])
1L
>>> # split subpopulations
>>> pop.splitSubPop(1, [2, 7])
(1L, 2L)
>>> pop.subPopSizes()
(3L, 2L, 7L)
>>> # remove subpopulations
>>> pop.removeSubPops(1)
>>> pop.subPopSizes()
(3L, 7L)
```

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

ulations during population operations so you can identify the ID of a subpopulation by its name. Note that simuPOP allows duplicate subpopulation names.

Listing 3.8: Use of subpopulation names

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[3, 4, 5], subPopNames=['x', 'y', 'z'])
>>> pop.removeSubPops([1])
>>> pop.subPopNames()
('x', 'z')
>>> pop.subPopByName('z')
1L
>>> pop.splitSubPop(1, [2, 3])
(1L, 2L)
>>> pop.subPopNames()
('x', 'z', 'z')
>>> pop.setSubPopName('z-1', 1)
>>> pop.subPopNames()
('x', 'z-1', 'z')
>>> pop.subPopByName('z')
2L
```

3.3.3 Virtual subpopulations and virtual splitters *

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

VSPs are defined by virtual splitters. **It is a definition for groups of individuals in each subpopulation.** A splitter defines the same number of VSPs in all subpopulations, although sizes of these VSPs vary across subpopulations due to subpopulation differences. For example, a `SexSplitter()` defines two VSPs, the first with all male individuals and the second with all female individuals, and a `InfoSplitter(field='x', values=[1, 2, 4])` defines three VSPs whose members have values `1`, `2` and `4` at information field `x`, respectively. This splitter also allows the use of cutoff values and ranges to define VSPs. If different types of VSPs are needed, a combined splitter can be used to combine VSPs defined by several splitters.

A VSP is represented by a `[sp, vsp]` pair where `sp` and `vsp` can be subpopulation indexes or names. Its name and size can be obtained using functions `subPopName()` and `subPopSize()`. Example 3.9 demonstrates how to apply virtual splitters to a population, and how to check VSP names and sizes.

Listing 3.9: Define virtual subpopulations in a population

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=[200, 400], loci=[30], infoFields='x')
>>> # assign random information fields
>>> sim.initSex(pop)
>>> sim.initInfo(pop, lambda: random.randint(0, 3), infoFields='x')
>>> # define a virtual splitter by sex
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.numVirtualSubPop()    # Number of defined VSPs
2L
>>> pop.subPopName([0, 0])    # Each VSP has a name
'Male'
```

```
>>> pop.subPopSize([0, 1])    # Size of VSP 1 in subpopulation 0
109L
>>> pop.subPopSize([0, 'Female'])    # Refer to vsp by its name
109L
>>> # 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
4L
>>> pop.subPopName([0, 0])    # Each VSP has a name
'x = 0'
>>> pop.subPopSize([0, 0])    # Size of VSP 0 in subpopulation 0
48L
>>> pop.subPopSize([1, 0])    # Size of VSP 0 in subpopulation 1
100L
```

VSP provides an easy way to access groups of individuals in a subpopulation and allows finer control of an evolutionary process. For example, mating schemes can be applied to VSPs which makes it possible to apply different mating schemes to, for example, individuals with different ages. By applying migration, mutation etc to VSPs, it is easy to implement advanced features such as sex-biased migrations, different mutation rates for individuals at different stages of a disease. Example 3.10 demonstrates how to initialize genotype and information fields to individuals in male and female VSPs.

Listing 3.10: Applications of virtual subpopulations

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(10, loci=[2, 3], infoFields='Sex')
>>> sim.initSex(pop)
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> # 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)])
>>> # set Sex information field to 0 for all males, and 1 for all females
>>> pop.setIndInfo([sim.MALE], 'Sex', [0, 0])
>>> pop.setIndInfo([sim.FEMALE], 'Sex', [0, 1])
>>> # Print individual genotypes, followed by values at information field Sex
>>> sim.dump(pop, structure=False)
SubPopulation 0 (), 10 Individuals:
   0: FU 11 111 | 11 111 |  2
   1: FU 11 111 | 11 111 |  2
   2: MU 00 000 | 00 000 |  1
   3: MU 00 000 | 00 000 |  1
   4: MU 00 000 | 00 000 |  1
   5: MU 00 000 | 00 000 |  1
   6: MU 00 000 | 00 000 |  1
   7: FU 11 111 | 11 111 |  2
   8: FU 11 111 | 11 111 |  2
   9: FU 11 111 | 11 111 |  2
```

3.3.4 Advanced virtual subpopulation splitters **

simuPOP provides a number of virtual splitters that can define VSPs using specified properties. For example, `InfoSplitter(field='a', values=[1,2,3])` defines three VSPs whose individuals have values `1`, `2`, and `3` at information field `a`, respectively; `SexSplitter()` defines two VSPs of male and female individuals, respectively; and

`RangeSplitter(ranges=[[0, 2000], [2000, 5000]])` defines two VSPs using two blocks of individuals.

A `CombinedSplitter` can be used if your simulation needs more than one sets of VSPs. For example, you may want to split your subpopulations both by sex and by affection status. In this case, you can define a combined splitter using

```
CombinedSplitter(splitters=[SexSplitter(), AffectionSplitter()])
```

This splitter simply stacks VSPs defined in `AffectionSplitter()` after `SexSplitter()` so that unaffected and affected VSPs are now VSPs 2 and 3 (0 and 1 are used for male and female VSPs).

There are also scenarios when you would like to define finer VSPs with individuals belonging to more than one VSPs. For example, you may want to have a look of frequencies of certain alleles in affected male vs affected females, or count the number of males and females with certain value at an information field. In this case, a `ProductSplitter` can be used to define VSPs using interactions of several VSPs. For example,

```
ProductSplitter(splitters=[SexSplitter(), AffectionSplitter()])
```

defines 4 subpopulations by splitting VSPs defined by `SexSplitter()` with affection status. These four VSPs will then have unaffected male, affected male, unaffected female and affected female individuals, respectively.

If you consider `ProductSplitter` as an intersection splitter that defines new VSPs as intersections of existing VSPs, you may wonder how to define unions of VSPs. For example, you can make a case where you want to consider Individuals with information field a < 0 or a > 100 together. A regular `InfoSplitter(field='a', cutoff=[0, 100])` cannot do that because it defines three VSPs with $a < 0$, $0 \leq a < 100$ and $a \geq 100$, respectively. The trick here is to use parameter `vspMap` of a `CombinedSplitter`. If this parameter is defined, multiple VSPs could be groups or reordered to define a new set of VSPs. For example,

```
CombinedSplitter(splitters=[InfoSplitter(field='a', cutoff=[0, 100])], vspMap=[[0,2], 1])
```

defines two VSPs using VSPs 0 and 2, and VSP 1 defined by the `InfoSplitter` so that the first VSP contains individuals with $a < 0$ or $a \geq 100$.

Example 3.11 demonstrates some advanced usages of virtual splitters.

Listing 3.11: Advanced virtual subpopulation usages.

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=[2000, 4000], loci=[30], infoFields='x')
>>> # assign random information fields
>>> sim.initSex(pop)
>>> sim.initInfo(pop, lambda: random.randint(0, 3), infoFields='x')
>>> #
>>> # 1, use a combined splitter
>>> pop.setVirtualSplitter(sim.CombinedSplitter(splitters = [
...     sim.SexSplitter(),
...     sim.InfoSplitter(field='x', values=[0, 1, 2, 3])
... ]))
>>> pop.numVirtualSubPop()    # Number of defined VSPs
6L
>>> pop.subPopName([0, 0])    # Each VSP has a name
'Male'
>>> pop.subPopSize([0, 0])    # sim.MALE
1011L
>>> pop.subPopSize([1, 4])    # individuals in sp 1 with value 2 at field x
1008L
>>> #
>>> # use a product splitter that defines additional VSPs by sex and info
>>> pop.setVirtualSplitter(sim.ProductSplitter(splitters = [
```

```
...     sim.SexSplitter(names=['M', 'F']),  # give a new set of names
...     sim.InfoSplitter(field='x', values=[0, 1, 2, 3])
... ]))
>>> pop.numVirtualSubPop()    # Number of defined VSPs
8L
>>> pop.subPopName([0, 0])    # Each VSP has a name
'M, x = 0'
>>> pop.subPopSize([0, 0])    # sim.MALE with value 1 in sp 0
235L
>>> pop.subPopSize([1, 5])    # sim.FEMALE with value 1 in sp 1
514L
>>> #
>>> # use a combined splitter to join VSPs defined by a
>>> # product splitter
>>> pop.setVirtualSplitter(sim.CombinedSplitter([
...     sim.ProductSplitter([
...         sim.SexSplitter(),
...         sim.InfoSplitter(field='x', values=[0, 1, 2, 3])])],
...     vspMap = [[0,1,2], [4,5,6], [7]],
...     names = ['Male x<=3', 'Female x<=3', 'Female x=4']))
>>> pop.numVirtualSubPop()    # Number of defined VSPs
3L
>>> pop.subPopName([0, 0])    # Each VSP has a name
'Male x<=3'
>>> pop.subPopSize([0, 0])    # sim.MALE with value 0, 1, 2 at field x
753L
>>> pop.subPopSize([1, 1])    # sim.FEMALE with value 0, 1 or 2 at field x
1478L
```

3.3.5 Access individuals and their properties

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

If you would like to access a group of individuals, either from a whole population, a subpopulation, or from a virtual subpopulation, `Population.individuals([subPop])` is easier to use. This function returns a Python iterator that can be used to iterate through individuals. An advantage of this function is that `subPop` can be a virtual subpopulation which makes it easy to iterate through Individuals with certain properties (such as all male Individuals). If you would like to iterate through multiple virtual subpopulations in one or more ancestral generations, you can use another function `Population.allIndividuals(subPops, ancGens)`.

If more than one generations are stored in a population, function `ancestor(idx, [subPop], gen)` can be used to access Individual from an ancestral generation (see Section 3.3.7 for details). Because there is no group access function for ancestors, it may be more convenient to use `useAncestralGen` to make an *ancestral* generation the *current* generation, and use `Population.Individuals`. Note that ancestor() function can always access individuals at a certain generation, regardless which generation the current generation is. Example 3.13 demonstrates how to use all these Individual-access functions.

If an unique ID is assigned to all individuals in a population, you can look up individuals from their IDs using function `Population.indByID()`. The information field to save individual ID is usually `ind_id` and you can use operator `IdTagger` and its function form `tagID` to set this field. Note that this function can be used to look up individuals in the present and all ancestral generations, although a parameter (`ancGen`) can be used to limit search to a specific generation if you know in advance which generation the individual locates.

Listing 3.12: Access individuals of a population

```
>>> import simuPOP as sim
>>> # create a sim.population with two generations. The current generation has values
>>> # 0-9 at information field x, the parental generation has values 10-19.
>>> pop = sim.Population(size=[5, 5], loci=[2, 3], infoFields='x', ancGen=1)
>>> pop.setIndInfo(range(10, 20), 'x')
>>> pop1 = pop.clone()
>>> pop1.setIndInfo(range(10), 'x')
>>> pop.push(pop1)
>>> #
>>> ind = pop.individual(5)       # using absolute index
>>> ind.x
5.0
>>> ind.x       # the same as ind.x
5.0
>>> # use a for loop, and relative index
>>> for idx in range(pop.subPopSize(1)):
...     print(pop.individual(idx, 1).x)
...
5.0
6.0
7.0
8.0
9.0
>>> # It is usually easier to use an iterator
>>> for ind in pop.individuals(1):
...     print(ind.x)
...
5.0
6.0
7.0
8.0
9.0
>>> # Access individuals in VSPs
>>> pop.setVirtualSplitter(sim.InfoSplitter(cutoff=[3, 7, 17], field='x'))
>>> for ind in pop.individuals([1, 1]):
...     print(ind.x)
...
5.0
6.0
>>> # Access all individuals in all ancestral generations
>>> print([ind.x for ind in pop.allIndividuals()])
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0]
>>> # or only specified subpopulations or ancestral generations
>>> print([ind.x for ind in pop.allIndividuals(subPops=[(0,2), (1,3)], ancGens=1)])
[10.0, 11.0, 12.0, 13.0, 14.0, 17.0, 18.0, 19.0]
>>>
>>> # Access individuals in ancetral generations
>>> pop.ancestor(5, 1).x        # absolute index
15.0
>>> pop.ancestor(0, 1, 1).x     # relative index
15.0
>>> # Or make ancestral generation the current generation and use 'individual'
>>> pop.useAncestralGen(1)
```

```
>>> pop.individual(5).x         # absolute index
15.0
>>> pop.individual(0, 1).x      # relative index
15.0
>>> # 'ancestor' can still access the 'present' (generation 0) generation
>>> pop.ancestor(5, 0).x
5.0
>>> # access individual by ID
>>> pop.addInfoFields('ind_id')
>>> sim.tagID(pop)
>>> [int(ind.ind_id) for ind in pop.individuals()]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> # access individual by ID. Note that individual 12 is in the parental generation
>>> pop.indByID(12).x
1.0
```

Although it is easy to access individuals in a population, it is often more efficient to access genotypes and information fields in batch mode. For example, functions `genotype()` and `setGenotype()` can read/write genotype of all individuals in a population or (virtual) subpopulation, functions `indInfo()` and `setIndInfo()` can read/write certain information fields in a population or (virtual) subpopulation. The write functions work in a circular manner in the sense that provided values are reused if they are not enough to fill all genotypes or information fields. Example 3.13 demonstrates the use of such functions.

Listing 3.13: Access Individual properties in batch mode

```
>>> 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')
>>> pop.indInfo('x')
(8.0, 0.0, 1.0, 4.0, 9.0, 5.0, 9.0, 1.0, 8.0, 4.0)
>>> pop.setGenotype([0, 1, 2, 3], 0)
>>> pop.genotype(0)
[0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
>>> pop.setVirtualSplitter(sim.InfoSplitter(cutoff=[3], field='x'))
>>> pop.setGenotype([0])    # clear all values
>>> pop.setGenotype([5, 6, 7], [1, 1])
>>> pop.indInfo('x', 1)
(9.0, 5.0, 9.0, 1.0, 8.0, 4.0)
>>> pop.genotype(1)
[5, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6, 7, 0, 0, 0, 0, 5, 6, 7, 5, 6, 7, 5, 6]
```

3.3.6 Attach arbitrary auxillary information using information fields

Information fields are usually set during population creation, using the `infoFields` parameter of the population constructor. It can also be set or added using functions `setInfoFields`, `addInfoField` and `addInfoFields`. Example 3.14 demonstrates how to read and write information fields from an individual, or from a population in batch mode. Note that functions `Population.indInfo` and `Population.setIndInfo` can be applied to (virtual) subpopulation using a optional parameter subPop.

Listing 3.14: Add and use of information fields in a population

```
>>> import simuPOP as sim
>>> pop = sim.Population(10)
>>> pop.setInfoFields(['a', 'b'])
```

```
>>> pop.addInfoFields('c')
>>> pop.addInfoFields(['d', 'e'])
>>> pop.infoFields()
('a', 'b', 'c', 'd', 'e')
>>> #
>>> # information fields can be accessed in batch mode
>>> pop.setIndInfo([1], 'c')
>>> # as well as individually.
>>> for ind in pop.individuals():
...     ind.e = ind.c + 1
...
>>> print(pop.indInfo('e'))
(2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0)
```

3.3.7 Keep track of ancestral generations

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

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

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

Several member functions can be used to manipulate ancestral generations:

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

If a population has several ancestral generations, they are referred by their indexes 0 (the latest generation), 1 (parental generation), ... and k (top-most ancestral generation) where k equals to `ancestralGens()`. In many cases, you can retrieve the properties of ancestral generations directly, using functions such as

- `popSize(ancGen=-1), subPopSizes(ancGen=-1), subPopSize(subPop, ancGen=-1)`: population and subpopulation sizes of ancestral generation `ancGen`.
- `ancestor(index, ancGen)`: Get a reference to the `index` individual of ancestral generation `ancGen`.

However, most population member functions work at the current generation so you will need to switch to an ancestral generation using function `useAncestralGen()` if you would like to manipulate an ancestral generation. For example, you can remove the second subpopulation of the parental generation using functions:

```
pop.useAncestralGen(1)
pop.removeSubPops(1)
```

A typical use of ancestral generations is demonstrated in example 3.17. In this example, a population is created and is initialized with allele frequency 0.5. Its ancestral depth is set to 2 at the beginning of generation 18 so that it can hold parental generations at generation 18 and 19. The allele frequency at each generation is calculated and displayed, both during evolution using a `Stat` operator, and after evolution using the function form this operator. Note that setting the ancestral depth at the end of an evolutionary process is a common practice because we are usually only interested in the last few generations.

Listing 3.15: Ancestral populations

```
>>> import simuPOP as sim
>>> pop = sim.Population(500, loci=1, ancGen=2)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme = sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0, begin=-3),
...         sim.PyEval(r"'%.3f\n' % alleleFreq[0][0]", begin=-3)
...     ],
...     gen = 20
... )
0.495
0.510
0.506
20L
>>> # information
>>> pop.ancestralGens()
2
>>> pop.popSize(ancGen=1)
500L
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> # number of males in the current and parental generation
>>> pop.subPopSize((0,0)), pop.subPopSize((0,0), ancGen=1)
(254L, 249L)
>>> # start from current generation
>>> for i in range(pop.ancestralGens(), -1, -1):
...   pop.useAncestralGen(i)
...   sim.stat(pop, alleleFreq=0)
...   print('%d   %.3f' % (i, pop.dvars().alleleFreq[0][0]))
...
2   0.495
1   0.510
0   0.506
>>> # restore to the current generation
>>> pop.useAncestralGen(0)
```

3.3.8 Change genotypic structure of a population

Several functions are provided to remove, add empty loci or chromosomes, and to merge loci or chromosomes from another population. They can be used to trim unneeded loci, expand existing population or merge two populations. Example 3.17 demonstrates how to use these populations. Note that function `Population.addLociFrom` by default merges chromosomes one by one according to chromosome index. If `byName` is set to True, it will try to match chromosomes by name and merge them. This example also demonstrates the use of `DBG_WARNING` flag, which will trigger a warning message when chromosomes with different names are merged.

Listing 3.16: Add and remove loci and chromosomes

```
>>> import simuOpt
>>> simuOpt.setOptions(debug='DBG_WARNING')
>>> import simuPOP as sim
Turn on debug 'DBG_WARNING'
>>> pop = sim.Population(10, loci=3, chromNames=['chr1'])
>>> # 1 1 1,
>>> pop.setGenotype([1])
>>> # 1 1 1, 0 0 0
>>> pop.addChrom(lociPos=[0.5, 1, 2], lociNames=['rs1', 'rs2', 'rs3'],
...     chromName='chr2')
>>> pop1 = sim.Population(10, loci=3, chromNames=['chr3'],
...     lociNames=['rs4', 'rs5', 'rs6'])
>>> # 2 2 2,
>>> pop1.setGenotype([2])
>>> # 1 1 1, 0 0 0, 2 2 2
>>> pop.addChromFrom(pop1)
>>> # 1 1 1, 0 0 0, 2 0 2 2 0
>>> pop.addLoci(chrom=[2, 2], pos=[1.5, 3.5], lociNames=['rs7', 'rs8'])
(7L, 10L)
>>> # 1 1 1, 0 0 0, 2 0 2 0
>>> pop.removeLoci(8)
>>> # loci names can also be used.
>>> pop.removeLoci(['rs1', 'rs7'])
>>> sim.dump(pop)
Ploidy: 2 (diploid)
Chromosomes:
1: chr1 (AUTOSOME, 3 loci)
   (1),  (2),  (3)
2: chr2 (AUTOSOME, 2 loci)
  rs2 (1), rs3 (2)
3: chr3 (AUTOSOME, 3 loci)
  rs4 (1), rs6 (3), rs8 (3.5)
population size: 10 (1 subpopulations with 10 Individuals)
Number of ancestral populations: 0

SubPopulation 0 (), 10 Individuals:
   0: MU 111 00 220 | 111 00 220
   1: MU 111 00 220 | 111 00 220
   2: MU 111 00 220 | 111 00 220
   3: MU 111 00 220 | 111 00 220
   4: MU 111 00 220 | 111 00 220
   5: MU 111 00 220 | 111 00 220
   6: MU 111 00 220 | 111 00 220
   7: MU 111 00 220 | 111 00 220
```

```
   8: MU 111 00 220 | 111 00 220
   9: MU 111 00 220 | 111 00 220

>>> # add loci from another population
>>> pop2 = sim.Population(10, loci=2, lociPos=[0.1, 2.2], chromNames='chr3')
>>> pop.addLociFrom(pop2)
WARNING: Chromosome 'chr3' is merged to chromosome 'chr1'.
>>> pop.addLociFrom(pop2, byName=2)
Traceback (most recent call last):
  File "/var/folders/ys/gnzk0qbx5wbdgm531v82xxljv5yqy8/T/tmpQTmzGW", line 1, in <module>
    #begin_file log/addRemoveLoci.py
TypeError: in method 'Population_addLociFrom', argument 3 of type 'bool'
>>> sim.dump(pop, genotype=False)
Ploidy: 2 (diploid)
Chromosomes:
1: chr1 (AUTOSOME, 5 loci)
   (0.1),  (1),  (2),  (2.2),  (3)
2: chr2 (AUTOSOME, 2 loci)
  rs2 (1), rs3 (2)
3: chr3 (AUTOSOME, 3 loci)
  rs4 (1), rs6 (3), rs8 (3.5)
population size: 10 (1 subpopulations with 10 Individuals)
Number of ancestral populations: 0
```

3.3.9 Remove or extract individuals and subpopulations from a population

Functions `Population.removeIndividuals` and `Population.removeSubPops` remove selected individuals or groups of individuals from a population. Functions `Population.extractIndividuals` and `Population.extractSubPops` extract individuals and subpopulations from an existing population and form a new one.

Functions `removeIndividauls` and `extractIndividuals` could be used to remove or extract individuals from the present generation by indexes or from all ancestral generations by IDs or a Python filter function. This function should accept parameter `ind` or one or more information fields. simuPOP will pass individual for parameter `ind`, and values at specified information fields (`age` in this example) of each individual to this function. The present population structure will be kept, even if some subpopulations are left empty. For example, you could remove the first thirty individuals of a population using

```
pop.removeIndividuals(indexes=range(30))
```

or remove all individuals at age 20 or 30 using

```
pop.removeIndividuals(IDs=(20, 30), idField='age')
```

or remove all individuals with age between 20 and 30 using

```
pop.removeIndividuals(filter=lambda age: age >=20 and age <=30)
```

. In the last example, a Python lambda function is defined to avoid the definition of a named function.

Functions `removeSubPops` or `extractSubPops` could be used to remove or extract subpopulations, or goups of individuals defined by virtual subpopulations from a population. The latter case is very interesting because it could be used to remove or extract individuals with similar properties, such as all individuals between the ages 40 and 60, as demonstrated in Example 3.17.

Listing 3.17: Extract individuals, loci and information fields from an existing population

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=[200, 200], loci=[5, 5], infoFields='age')
>>> sim.initGenotype(pop, genotype=range(10))
>>> sim.initInfo(pop, lambda: random.randint(0,75), infoFields='age')
>>> pop.setVirtualSplitter(sim.InfoSplitter(field='age', cutoff=[20, 60]))
>>> # remove individuals
>>> pop.removeIndividuals(indexes=range(0, 300, 10))
>>> print(pop.subPopSizes())
(180L, 190L)
>>> # remove individuals using IDs
>>> pop.setIndInfo([1, 2, 3, 4], field='age')
>>> pop.removeIndividuals(IDs=[2, 4], idField='age')
>>> # remove indiviuals using a filter function
>>> sim.initSex(pop)
>>> pop.removeIndividuals(filter=lambda ind: ind.sex() == sim.MALE)
>>> print([pop.individual(x).sex() for x in range(8)])
[2, 2, 2, 2, 2, 2, 2, 2]
>>> #
>>> # remove subpopulation
>>> pop.removeSubPops(1)
>>> print(pop.subPopSizes())
(56L,)
>>> # remove virtual subpopulation (people with age between 20 and 60)
>>> pop.removeSubPops([(0, 1)])
>>> print(pop.subPopSizes())
(56L,)
>>> # extract another virtual subpopulation (people with age greater than 60)
>>> pop1 = pop.extractSubPops([(0,2)])
>>> sim.dump(pop1, structure=False, max=10)
SubPopulation 0 (), 0 Individuals:
```

3.3.10 Store arbitrary population information as population variables

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

simuPOP provides two functions, namely `Population.vars()` and `Population.dvars()` to access a population dictionary. These functions return the same dictionary object but `dvars()` returns a wrapper class so that you can access this dictionary as attributes. For example, `pop.vars()['alleleFreq'][0]` is equivalent to `pop.dvars().alleleFreq[0]`. Because dictionary `subPop[spID]` is frequently used by operators to store variables related to a particular (virtual) subpopulation, function `pop.vars(subPop)` is provided as a shortcut to `pop.vars()['subPop'][spID]`. Example 3.18 demonstrates how to set and access population variables.

Listing 3.18: population variables

```
>>> import simuPOP as sim
>>> from pprint import pprint
>>> pop = sim.Population(100, loci=2)
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>> print(pop.vars())    # No variable now
{}
```

```
>>> pop.dvars().myVar = 21
>>> print(pop.vars())
{'myVar': 21}
>>> sim.stat(pop, popSize=1, alleleFreq=0)
>>> # pprint prints in a less messy format
>>> pprint(pop.vars())
{'alleleFreq': {0: {0: 0.275, 1: 0.725}},
 'alleleNum': {0: {0: 55.0, 1: 145.0}},
 'myVar': 21,
 'popSize': 100,
 'subPopSize': [100]}
>>> # print number of allele 1 at locus 0
>>> print(pop.vars()['alleleNum'][0][1])
145.0
>>> # use the dvars() function to access dictionary keys as attributes
>>> print(pop.dvars().alleleNum[0][1])
145.0
>>> print(pop.dvars().alleleFreq[0])
defdict({0: 0.275, 1: 0.725})
```

It is important to understand that this dictionary forms a **local namespace** in which Python expressions can be evaluated. This is the basis of how expression-based operators work. For example, the `PyEval` operator in example 1.1 evaluates expression "`'%.2f\t' % LD[0][1]`" in each population's local namespace when it is applied to that population. This yields different results for different population because their LD values are different. In addition to Python expressions, Python statements can also be executed in the local namespace of a population, using the `stmts` parameter of the `PyEval` or `PyExec` operator. Example 3.19 demonstrates the use of a simuPOP terminator, which terminates the evolution of a population when its expression is evaluated as `True`. Note that The `evolve()` function of this example does not specify how many generations to evolve so it will stop only after all replicates stop. The return value of this function indicates how many generations each replicate has evolved. This example also demonstrates how to run multiple replicates of an evolutionary process, which we will discuss in detail latter.

Listing 3.19: Expression evaluation in the local namespace of a population

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(100, loci=1), rep=5)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme = sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.TerminateIf('len(alleleFreq[0]) == 1')
...     ]
... )
(129L, 1540L, 180L, 247L, 242L)
```

3.3.11 Save and load a population

simuPOP populations can be saved to and loaded from disk files using `Population.save(file)` member function and global function `loadPopulation`. **Virtual splitters are not saved** because they are considered as runtime definitions. Although files in any extension can be used, extension `.pop` is recommended. Example 3.20 demonstrates how to save and load a population in the native simuPOP format.

Listing 3.20: Save and load a population

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=5, chromNames=['chrom1'])
>>> pop.dvars().name = 'my sim.Population'
>>> pop.save('sample.pop')
>>> pop1 = sim.loadPopulation('sample.pop')
>>> pop1.chromName(0)
'chrom1'
>>> pop1.dvars().name
'my sim.Population'
```

The native simuPOP format is portable across different platforms but is not human readable and is not recognized by other applications. If you need to save a simuPOP population in a format that is recognizable by a particular software, you can use functions `importPopulation`, `export`, and operator `Exporter` if you would like to export populations during evolution. These functions are defined in module `simuPOP.utils`.

3.3.12 Import and export datasets in unsupported formats *

simuPOP provides a few utility functions to import and export populations in common formats such as GENEPOP, Phylip, and STRUCTURE (see chapter utility modules for details). If you need to import data from a file in a format that is not currently supported, you generally need to first scan the file for information such as number and names of chromosomes, loci, alleles, subpopulation, and individuals. After you create a population without genotype information from these parameters, you can scan the file for the second time and fill the population with genotypes and other information. Example 3.21 demonstrates how to define a function to import from a file that is saved by function `utils.saveCSV`.

Listing 3.21: Import a population from another file format

```
>>> import simuPOP as sim
>>> def importData(filename):
...     'Read data from ``filename`` and create a population'
...     data = open(filename)
...     header = data.readline()
...     fields = header.split(',')
...     # columns 1, 3, 5, ..., without trailing '_1'
...     names = [fields[x].strip()[:-2] for x in range(1, len(fields), 2)]
...     popSize = 0
...     alleleNames = set()
...     for line in data.readlines():
...         # get all allele names
...         alleleNames |= set([x.strip() for x in line.split(',')[1:]])
...         popSize += 1
...     # create a population
...     alleleNames = list(alleleNames)
...     pop = sim.Population(size=popSize, loci=len(names), lociNames=names,
...         alleleNames=alleleNames)
...     # start from beginning of the file again
...     data.seek(0)
...     # discard the first line
...     data.readline()
...     for ind, line in zip(pop.individuals(), data.readlines()):
...         fields = [x.strip() for x in line.split(',')]
...         sex = sim.MALE if fields[0] == '1' else sim.FEMALE
...         ploidy0 = [alleleNames.index(fields[x]) for x in range(1, len(fields), 2)]
```

```
...         ploidy1 = [alleleNames.index(fields[x]) for x in range(2, len(fields), 2)]
...         ind.setGenotype(ploidy0, 0)
...         ind.setGenotype(ploidy1, 1)
...         ind.setSex(sex)
...     # close the file
...     data.close()
...     return pop
...
>>> from simuPOP.utils import saveCSV
>>> pop = sim.Population(size=[10], loci=[3, 2], lociNames=['rs1', 'rs2', 'rs3', 'rs4', 'rs5'],
...     alleleNames=['A', 'B'])
>>> sim.initSex(pop)
>>> sim.initGenotype(pop, freq=[0.5, 0.5])
>>> # output sex but not affection status.
>>> saveCSV(pop, filename='sample.csv', affectionFormatter=None,
...     sexFormatter={sim.MALE:1, sim.FEMALE:2})
>>> # have a look at the file
>>> print(open('sample.csv').read())
sex, rs1_1, rs1_2, rs2_1, rs2_2, rs3_1, rs3_2, rs4_1, rs4_2, rs5_1, rs5_2
2, B, B, B, B, B, A, A, B, B, A
2, B, A, B, A, B, A, A, A, A, B
1, B, B, B, B, B, B, B, B, B, A
1, B, A, B, A, B, B, B, A, A, A
1, B, B, B, B, B, B, A, A, B, A
1, A, B, B, A, B, B, B, A, B, B
1, B, B, B, B, B, B, B, B, A, A
2, B, B, A, A, B, A, A, A, B, A
2, A, B, B, B, A, B, B, A, A, B
2, B, A, A, B, A, A, B, B, B, A

>>> pop1 = importData('sample.csv')
>>> sim.dump(pop1)
Ploidy: 2 (diploid)
Chromosomes:
1:  (AUTOSOME, 5 loci)
  rs1 (1), rs2 (2), rs3 (3), rs4 (4), rs5 (5)
population size: 10 (1 subpopulations with 10 Individuals)
Number of ancestral populations: 0

SubPopulation 0 (), 10 Individuals:
   0: FU BBBAB | BBABA
   1: FU BBBAA | AAAAB
   2: MU BBBBB | BBBBA
   3: MU BBBBA | AABAA
   4: MU BBBAB | BBBAA
   5: MU ABBBB | BABAB
   6: MU BBBBA | BBBBA
   7: FU BABAB | BAAAA
   8: FU ABABA | BBBAB
   9: FU BAABB | ABABA
```

Unless there are specific requirements in the order and labeling of individuals, exporting a simuPOP population is usually straightforward. Functions that are useful in such occasions include structural functions `Population.numSubPop()`, `Population.subPopName()`, `Population.popSize()` and `Population.subPopSizes()`, and individual

access functions `Population.individual()` and `Population.individuals()` and individual population access functions such as `Individual.allele()` and `Individual.info()`. Function `saveFSTAT` in the cookbook module `fstatUtil` or `saveCSV` in module `simuPOP.utils` are good examples you can follow.

Chapter 4

simuPOP Operators

simuPOP is large, consisting of more than 70 operators and various functions that covers all important aspects of genetic studies. These includes mutation (*k*-allele, stepwise, generalized stepwise), migration (arbitrary, can create new subpopulation), recombination (uniform or nonuniform), gene conversion, quantitative trait, selection, penetrance (single or multi-locus, hybrid), ascertainment (case-control, affected sibpairs, random), statistics calculation (allele, genotype, haplotype, heterozygote number and frequency; expected heterozygosity; bi-allelic and multi-allelic D, D' and r^2 linkage disequilibrium measures; F_{st}, F_{it} and F_{is}); pedigree tracing, visualization (using R or other Python modules). This chapter covers the basic and some not-so-basic usages of these operators, organized roughly by genetic factors.

4.1 Introduction to operators

Operators are objects that act on populations. There are two types of operators:

- **Operators that are applied to populations**. These operators are used in the `initOps`, `preOps`, `postOps` and `finalOps` parameters of the `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. Examples of such operators include `MergeSubPops` to merge subpopulations and `StepwiseMutator` to mutate individuals using a stepwise mutation model.
- **Operators that are applied to individuals** (offspring) during mating. These operators 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 transmit parental genotype to offspring according to Mendelian laws and `ParentsTagger` that record the indexes of parents in the parental population to each offspring.

Some mutators could be applied both to populations and individuals. For example, an `IdTagger` could be applied to a whole population and assign an unique ID to all individuals, or to offspring during mating.

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

4.1.1 Apply operators to selected replicates and (virtual) subpopulations at selected 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. 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 `at` is specified. It is worth noting that, if an evolutionary process has a pre-sepcified ending generation, negative generations numbers are allowed. They are counted backward from the ending generation.

For example, if a simulator starts at generation `0`, and the `evolve` function has parameter `gen=10`, the simulator will stop at the *beginning* of generation `10`. Generation `-1` refers to generation `9`, and generation `-2` refers to generation `8`, and so on. Example 4.1 demonstrates how to set applicable generations of an operator. In this example, a population is initialized before evolution using an `InitGenotype` operator. allele frequency at locus `0` is calculated at generation `80`, `90`, but not `100` because the evolution stops at the beginning of generation `100`. A `PyEval` operator outputs generation number and allele frequency at the end of generation `80` and `90`. Another `PyEval` operator outputs similar information at generation `90` and `99`, before and after mating. Note, however, because allele frequencies are only calculated twice, the pre-mating allele frequency at generation `90` is actually calculated at generation `80`, and the allele frequencies display for generation `99` are calculated at generation `90`. At the end of the evolution, the population is saved to a file using a `SavePopulation` operator.

Listing 4.1: Applicable generations of an operator.

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=[20])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.8, 0.2])
...     ],
...     preOps=[
...         sim.PyEval(r"'At the beginning of gen %d: allele Freq: %.2f\n' % (gen, alleleFreq[0][0])",
...             at = [-10, -1])
...     ],
...     matingScheme = sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0, begin=80, step=10),
...         sim.PyEval(r"'At the end of gen %d: allele freq: %.2f\n' % (gen, alleleFreq[0][0])",
...             begin=80, step=10),
...         sim.PyEval(r"'At the end of gen %d: allele Freq: %.2f\n' % (gen, alleleFreq[0][0])",
...             at = [-10, -1])
...     ],
...     finalOps=sim.SavePopulation(output='sample.pop'),
...     gen=100
... )
At the end of gen 80: allele freq: 0.92
At the beginning of gen 90: allele Freq: 0.92
At the end of gen 90: allele freq: 0.93
At the end of gen 90: allele Freq: 0.93
At the beginning of gen 99: allele Freq: 0.93
At the end of gen 99: allele Freq: 0.93
100L
```

4.1.2 Applicable populations and (virtual) subpopulations

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

Parameter `reps` is used to control which replicate(s) an operator can be applied to. This parameter can be a list of replicate numbers or a single replicate number. Negative index is allowed where `-1` refers to the last replicate. This

technique has been widely used to produce table-like output where a `PyOutput` outputs a newline when it is applied to the last replicate of a simulator. Example 4.97 demonstrates how to use this `reps` parameter. It is worth noting that negative indexes are *dynamic* indexes relative to number of active populations. For example, `rep=-1` will refer to a previous population if the last population has stopped evolving. Use a non-negative replicate number if this is not intended.

Listing 4.2: Apply operators to a subset of populations

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(100, loci=[20]), 5)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2, 0.8])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0, step=10),
...         sim.PyEval('gen', step=10, reps=0),
...         sim.PyEval(r"'\t%.2f' % alleleFreq[0][0]", step=10, reps=(0, 2, -1)),
...         sim.PyOutput('\n', step=10, reps=-1)
...     ],
...     gen=30,
... )
0       0.23    0.22    0.29
10      0.15    0.23    0.21
20      0.04    0.07    0.10
(30L, 30L, 30L, 30L, 30L)
```

An operator can also be applied to specified (virtual) subpopulations. For example, an `initializer` can be applied to male individuals in the first subpopulation, and everyone in the second subpopulation using parameter `subPops=[(0,0)`, 1], if a virtual subpopulation is defined by individual sex. Generally speaking,

- `subPops=[]` applies the operator to all subpopulation. This is usually the default value of an operator.
- `subPops=[vsp1, vsp2,...]` applies the operator all specified (virtual) subpopulations. (e.g. `subPops=[(0,0)`, 1]).
- `subPops=sp` is an abbreviation for `subPops=[sp]`. If `sp` is virtual, it has to be written as `[sp]` because `subPops=(0, 1)` is intepreted as two non-virtual subpopulation.

However, not all operators support this parameter, and even if they do, their interpretations of parameter input may vary. Please refer to documentation for individual operators in *the simuPOP reference manual* for details.

4.1.3 Dynamically determined loci (parameter `loci`) *

Many operators accept a parameter `loci` to specify the applicable loci. This parameter can be

- `ALL_AVAIL`: all available loci of the population to which the operator is applied.
- [1, 2, 4, 5]: A list of loci indexes. When the operator is applied to a population, it will be applied to the specified loci.
- `[('chr1', 5), ('chr1', 10), ('chr2', 5)]`: A list of chromosome position pairs. That is to say, when the operator is applied to a population, it will find loci at specified position of specified chromosome. Here chromosome names are names specified by parameter `chromNames` of the `Population` constructor. That is to say, the operator can be applied to all population with such chromosomes and loci at specified locations.

- func: A function with an optional parameter `pop`. When the operator is applied to a population, it will call this function, optionally pass the population to be applied to this function, and use its output as indexes of loci.

The last usage is very interesting because it allows the determination of loci according to population property. For example, Example 4.3 shows an example with a `MaSelector` that is applied to the locus with highest frequency at each generation by calling function `mostPopular`, which calculates allele frequency and pick the locus with highest allele frequency, This example looks silly, but the technique is very useful in simulating the introduction of disease loci by, for example, adding positive selection pressure to one of the chosen loci.

Listing 4.3: Natural selection with dynamically determined loci

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=[10], infoFields='fitness')
>>>
>>> def mostPopular(pop):
...     sim.stat(pop, alleleFreq=sim.ALL_AVAIL)
...     freq = [pop.dvars().alleleFreq[x][1] for x in range(pop.totNumLoci())]
...     max_freq = max(freq)
...     pop.dvars().selLoci = (freq.index(max_freq), max_freq)
...     return [freq.index(max_freq)]
...
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.6, 0.4]),
...     ],
...     preOps=[
...         sim.MaSelector(fitness=[1, 0.9, 0.8], loci=mostPopular),
...         sim.PyEval(r"'gen=%d, select against %d with frequency %.2f\n' % (gen, selLoci[0], selLoci[1])"),
...     ],
...     matingScheme=sim.RandomMating(),
...     gen=10,
... )
gen=0, select against 6 with frequency 0.45
gen=1, select against 7 with frequency 0.46
gen=2, select against 2 with frequency 0.51
gen=3, select against 2 with frequency 0.48
gen=4, select against 2 with frequency 0.45
gen=5, select against 9 with frequency 0.45
gen=6, select against 3 with frequency 0.46
gen=7, select against 9 with frequency 0.44
gen=8, select against 7 with frequency 0.47
gen=9, select against 3 with frequency 0.44
10L
```

4.1.4 Write output of operators to one or more files

All operators we have seen, except for the `SavePopulation` operator in Example 4.1, write their output to the standard output, namely your terminal window. However, it would be much easier for bookkeeping and further analysis if these output can be redirected to disk files. Parameter `output` is designed for this purpose.

Parameter `output` can take the following values:

- `''` (an empty string): No output.

- '>': Write to standard output.
- 'filename' or '>filename': Write the output to a file named filename. If multiple operators write to the same file, or if the same operator writes to the file file several times, only the last write operation will succeed.
- '>>filename': Append the output to a file named filename. The file will be opened at the beginning of evolve function and closed at the end. An existing file will be cleared.
- '>>>filename': This is similar to the '>>' form but the file will not be cleared at the beginning of the evolve function.
- '!expr': expr is considered as a Python expression that will be evaluated at a population's local namespace whenever an output string is needed. For example, '!"%d.txt" % gen' would return 0.txt, 1.txt etc at generation 0, 1, ...
- File handle of an opened file. Actually any python object with a write function.
- A Python function that can accept a string as its only parameter (func(msg)). When an operator outputs a message, this function will be called with this message.
- A WithMode(output, 'b') object with output being the any of the allowed output string or function. This object tells simuPOP that the output is opened in binary model so that it should output bytes instead of texts to it. This is mostly designed for Python 3 because file objects in Python 2 accepts string even if they are opened in binary mode.

Because a table output such as the one in Example 4.97 is written by several operators, it is clear that all of them need to use the '>>' output format.

The SavePopulation operator in Example 4.1 write to file sample.pop. This works well if there is only one replicate but not so when the operator is applied to multiple populations. Only the last population will be saved successfully! In this case, the expression form of parameter output should be used.

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

Example 4.97 demonstrates the use of these two parameters. In this example, a table is written to file LD.txt using output='>>LD.txt'. Similar operation to output='R2.txt' fails because only the last R^2 value is written to this file. The last operator writes output for each replicate to their respective output file such as LD_0.txt, using an expression that involves variable rep.

Listing 4.4: Use the output and outputExpr parameters

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(size=1000, loci=2), rep=3)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[1, 2, 2, 1])
...     ],
...     matingScheme = sim.RandomMating(ops=sim.Recombinator(rates=0.01)),
...     postOps=[
...         sim.Stat(LD=[0, 1]),
...         sim.PyEval(r"'%.2f\t' % LD[0][1]", step=20, output='>>LD.txt'),
...         sim.PyOutput('\n', reps=-1, step=20, output='>>LD.txt'),
...         sim.PyEval(r"'%.2f\t' % R2[0][1]", output='R2.txt'),
...         sim.PyEval(r"'%.2f\t' % LD[0][1]", step=20, output="!'>>LD_%d.txt' % rep"),
...     ],
```

```
...     gen=100
... )
(100L, 100L, 100L)
>>> print(open('LD.txt').read())
0.24    0.25    0.24
0.19    0.18    0.20
0.15    0.14    0.19
0.14    0.10    0.13
0.14    0.11    0.08

>>> print(open('R2.txt').read())    # Only the last write operation succeed.
0.03
>>> print(open('LD_2.txt').read())  # Each replicate writes to a different file.
0.24    0.20    0.19    0.13    0.08
```

Example 4.5 demonstrates an advanced usage of the `output` parameter. In this example, a logging object is created to write to a logfile as well as the standard output. The `info` and `debug` functions of this object are assigned to two operators so that their outputs can be sent to both a logfile and to the console window. One of the advantages of using a logging mechanism is that debugging output could be suppressed easily by adjusting the logging level of the logging object. Note that function `logging.info()` automatically adds a new line to its input messages before it writes them to an output.

Listing 4.5: Output to a Python function

```
>>> import simuPOP as sim
>>> import logging
>>> # logging to a file simulation.log, with detailed debug information
>>> logging.basicConfig(
...     filename='simulation.log',
...     level=logging.DEBUG,
...     format='%(levelname)s: %(message)s',
...     filemode='w'
... )
>>> formatter = logging.Formatter('%(message)s')
>>> logger = logging.getLogger('')
>>> pop = sim.Population(size=1000, loci=2)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[1, 2, 2, 1])
...     ],
...     matingScheme = sim.RandomMating(ops=sim.Recombinator(rates=0.01)),
...     postOps=[
...         sim.Stat(LD=[0, 1]),
...         sim.PyEval(r"'LD: %d, %.2f' % (gen, LD[0][1])", step=20,
...             output=logger.info),   # send LD to console and a logfile
...         sim.PyEval(r"'R2: %d, %.2f' % (gen, R2[0][1])", step=20,
...             output=logger.debug),  # send R2 only to a logfile
...     ],
...     gen=100
... )
100L
>>> print(open('simulation.log').read())
INFO: LD: 0, 0.25
DEBUG: R2: 0, 0.97
```

```
INFO: LD: 20, 0.21
DEBUG: R2: 20, 0.73
INFO: LD: 40, 0.17
DEBUG: R2: 40, 0.48
INFO: LD: 60, 0.15
DEBUG: R2: 60, 0.35
INFO: LD: 80, 0.10
DEBUG: R2: 80, 0.18
```

4.1.5 During-mating operators

All operators in Examples 4.1, 4.2 and 4.4 are applied before or after mating. There is, however, a hidden during-mating operator that is called by `RandomMating()`. This operator is called `MendelianGenoTransmitter()` and is responsible for transmitting genotype from parents to offspring according to Mendel's laws. All pre-defined mating schemes (see Section 5.1) use a special kind of during-mating operator to transmit genotypes. They are called **genotype transmitters** just to show the kind of task they perform. More during mating operators could be specified by replacing the default operator used in the `ops` parameter of a mating scheme (or an offspring generator if you are defining your own mating scheme).

Operators used in a mating scheme honor applicability parameters `begin`, `step`, `end`, `at` and `reps` although they do not support negative population and replicate indexes. It is therefore possible to apply different during-mating operators at different generations. For example, a `Recombinator` is used in Example 4.6 to transmit parental genotypes to offspring after generation 30 while the `MendelianGenoTransmitter` is applied before that.

Listing 4.6: Genotype transmitters

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=10000, loci=2)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[1, 2, 2, 1])
...     ],
...     matingScheme = sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(end=29),
...         sim.Recombinator(rates=0.01, begin=30),
...     ]),
...     postOps=[
...         sim.Stat(LD=[0, 1]),
...         sim.PyEval(r"'gen %d, LD: %.2f\n' % (gen, LD[0][1])", step=20)
...     ],
...     gen=100
... )
gen 0, LD: 0.25
gen 20, LD: 0.25
gen 40, LD: 0.22
gen 60, LD: 0.20
gen 80, LD: 0.16
100L
```

During-mating operators can be applied to (virtual) subpopulations using parameter `subPops`, which **refers to (virtual) subpopulations in the offspring population**. Section 5.3.3 and 4.5 list all genotype transmitters, Section **??** demonstrates how to define your own genotype transmitter, Section 4.9.11 demonstrates the use of during-mating operator in virtual subpopulations.

4.1.6 Function form of an operator

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

```
InitGenotype(freq=[.3, .2, .5]).apply(pop)
```

Similarly, you can apply the hybrid penetrance model defined in Example 4.97 to a population by

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

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

Listing 4.7: The function form of operator `InitGenotype`

```
>>> from simuPOP import InitGenotype, Population
>>> def initGenotype(pop, *args, **kwargs):
...     InitGenotype(*args, **kwargs).apply(pop)
...
>>> pop = Population(1000, loci=[2,3])
>>> initGenotype(pop, freq=[.2, .3, .5])
```

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

```
initGenotype(pop, freq=[.3, .2, .5])
```

and

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

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

Finally, it is worth noting that, if you have a function that manipulates population, you can make it an operator by wrapping it in a `PyOperator` so that it can be called repeatedly during evolution. For example, for a function `myFunc` that works on a population, you can define a wrapper function

```
def Func(pop):
    # call myFunc
    myFunc(pop)
    return True
```

which can then use it in a `PyOperator` as follows:

```
PyOperator(func=Func)
```

The wrapper function is not needed if myFunc returns `True` by itself. It can also be simplifed to a lambda function

```
PyOperator(func=lambda pop: myFunc(pop) is None)
```

if you are certain that `myFunc` does not return any value (return `None`).

Note: Whereas output files specified by `'>'` are closed immediately after they are written, those specified by `'>>'` and `'>>>'` are not closed after the operator is applied to a population. This is not a problem when operators are used in a simulator because `Simulator.evolve` closes all files opened by operators, but can cause trouble when the operator is applied directly to a population. For example, multiple calls to `dump(pop, output='>>file')` will dump pop to `file` repeatedly but `file` will not be closed afterward. In this case, `closeOutput('file')` should be used to explicitly close the file.

4.2 Initialization

simuPOP provides three operators to initialize individual sex, information fields and genotype at the population level. A number of parameter are provided to cover most commonly used initialization scenarios. A Python operator can be used to intialize a population explicitly if none of the operators fits your need.

4.2.1 Initialize individual sex (operator `InitSex`)

Operator `InitSex()` and function `initSex()` initialize individual sex either randomly or using a given sequence. In the first case, individuals are assigned `MALE` or `FEMALE` with equal probability unless parameter *maleFreq* is used to specify the probability of having a male Individual. Alternatively, parameter *maleProp* can be used to specify exact proportions of male individuals so that you will have exactly 1000 males and 1000 females if you apply `InitSex(maleProp=0.5)` to a population of 2000 individuals.

Both parameters `maleFreq` and `maleProp` assigns individual sex randomly. If for some reason you need to specify individual sex explicitly, you could use a sequence of sex (`MALE` or `FEMALE`) to assign sex to individuals succesively. The list will be reused if needed. If a list of (virtual) subpopulations are given, this operator will only initialize individuals in these (virtual) subpopulations. Example 4.8 demonstrates how to use two `InitSex` operators to initialize two subpopulations.

Listing 4.8: Initialize individual sex

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000, 1000])
>>> sim.initSex(pop, maleFreq=0.3, subPops=0)
>>> sim.initSex(pop, sex=[sim.MALE, sim.FEMALE, sim.FEMALE], subPops=1)
>>> sim.stat(pop, numOfMales=True, vars='numOfMales_sp')
>>> print(pop.dvars(0).numOfMales)
290
>>> print(pop.dvars(1).numOfMales)
334
```

4.2.2 Initialize genotype (operator `InitGenotype`)

Operator `InitGenotype` (and its function form `initGenotype`) initializes individual genotype by allele frequency, allele proportion, haplotype frequency, haplotype proportions or a list of genotypes:

- By frequency of alleles. For example, `InitGenotype(freq=(0, 0.2, 0.4, 0.2))` will assign allele 0, 1, 2, and 3 with probability 0, 0.2, 0.4 and 0.2 respectively.
- By proportions of alleles. For example, `InitGenotype(prop=(0, 0.2, 0.4, 0.2))` will assign 400 allele 1, 800 allele 2 and 400 allele 3 to a diploid population with 800 individuals.
- By frequency of haplotypes. For example, `InitGenotype(haplotypes=[[0, 0], [1,1], [0,1],[1,1]])` will assign four haplotypes with equal probabilities. `InitGenotype(haplotypes=[[0, 0], [1,1], [0,1],[1,1]], freq=[0.2, 0.2, 0.3, 0.3])` will assign these haplotypes with different frequencies. If there are more than two loci, the haplotypes will be repeated.
- By frequency of haplotypes. For example, `InitGenotype(haplotypes=[[0, 0], [1,1], [0,1],[1,1]], prop=[0.2, 0.2, 0.3, 0.3])` will assign four haplotypes with exact proportions.
- By a list of genotype. For example, `InitGenotype(genotype=[1, 2, 2, 1])` will assign genotype `1`, `2`, `2`, `1` repeatedly to a population. If individuals in this population has two homologous copies of a chromosome with two loci, this operator will assign haplotype `1`, `2` to the first homologous copy of the chromosome, and `2`, `1` to the second copy.

- By multiple allele frequencies or proportions returned by a function passed to parameter `freq` or `prop` (new in version 1.1.7). This function can accept parameters `loc`, `subPop` or `vsp` and returns locus, subpopopulation or virtual subpopulation specific allele frequencies. For example, if you would like to initialize genotypes with random allele frequency, you can set `freq=lambda : random.random()` so that a new frequency is drawn from an uniform distribution for each new locus. Note that simuPOP expects the return value of this function to be a list of frequencies for alleles 0, 1, ..., but treats a single return value *x* as [*x, 1-x*] for simplicity.

Parameter `loci` and `ploidy` can be used to specify a subset of loci and homologous sets of chromosomes to initialize, and parameter `subPops` can be used to specify subsets of individuals to initialize. Example 4.9 demonstrates how to use these the `InitGenotype` operator, including examples on how to define and use virtual subpopulations to initialize individual genotype by sex or by proportion.

Listing 4.9: Initialize individual genotype

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[2000, 3000], loci=[5, 7])
>>> # by allele frequency
>>> def printFreq(pop, loci):
...     sim.stat(pop, alleleFreq=loci)
...     print(', '.join(['{:.3f}'.format(pop.dvars().alleleFreq[x][0]) for x in loci]))
...
>>> sim.initGenotype(pop, freq=[.4, .6])
>>> sim.dump(pop, max=6, structure=False)
SubPopulation 0 (), 2000 Individuals:
   0: MU 11000 0011111 | 11111 0101110
   1: MU 00000 1111111 | 11101 1111001
   2: MU 10111 0111100 | 01111 1011111
   3: MU 11011 1101010 | 11010 1011111
   4: MU 11011 0011010 | 10011 1001110
   5: MU 00001 1010011 | 11111 1111110
SubPopulation 1 (), 3000 Individuals:
2000: MU 10011 0010100 | 01001 0011010

>>> printFreq(pop, range(5))
0.397, 0.404, 0.400, 0.402, 0.406
>>> # by proportion
>>> sim.initGenotype(pop, prop=[0.4, 0.6])
>>> printFreq(pop, range(5))
0.400, 0.400, 0.400, 0.400, 0.400
>>> # by haplotype frequency
>>> sim.initGenotype(pop, freq=[.4, .6], haplotypes=[[1, 1, 0, 1], [0, 0, 1]])
>>> sim.dump(pop, max=6, structure=False)
SubPopulation 0 (), 2000 Individuals:
   0: MU 11011 1011101 | 00100 1001001
   1: MU 11011 1011101 | 11011 1011101
   2: MU 00100 1001001 | 00100 1001001
   3: MU 00100 1001001 | 00100 1001001
   4: MU 11011 1011101 | 11011 1011101
   5: MU 00100 1001001 | 11011 1011101
SubPopulation 1 (), 3000 Individuals:
2000: MU 00100 1001001 | 00100 1001001

>>> printFreq(pop, range(5))
0.597, 0.597, 0.403, 0.597, 0.597
>>> # by haplotype proportion
```

```
>>> sim.initGenotype(pop, prop=[0.4, 0.6], haplotypes=[[1, 1, 0], [0, 0, 1, 1]])
>>> printFreq(pop, range(5))
0.600, 0.600, 0.400, 0.000, 0.600
>>> # by genotype
>>> pop = sim.Population(size=[2, 3], loci=[5, 7])
>>> sim.initGenotype(pop, genotype=[1]*5 + [2]*7 + [3]*5 +[4]*7)
>>> sim.dump(pop, structure=False)
SubPopulation 0 (), 2 Individuals:
   0: MU 11111 2222222 | 33333 4444444
   1: MU 11111 2222222 | 33333 4444444
SubPopulation 1 (), 3 Individuals:
   2: MU 11111 2222222 | 33333 4444444
   3: MU 11111 2222222 | 33333 4444444
   4: MU 11111 2222222 | 33333 4444444

>>> #
>>> # use virtual subpopulation
>>> pop = sim.Population(size=[2000, 3000], loci=[5, 7])
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> sim.initSex(pop)
>>> sim.initGenotype(pop, genotype=range(10), loci=range(5))
>>> # initialize all males
>>> sim.initGenotype(pop, genotype=[2]*7, loci=range(5, 12),
...     subPops=[(0, 0), (1, 0)])
>>> # assign genotype by proportions
>>> pop.setVirtualSplitter(sim.ProportionSplitter([0.4, 0.6]))
>>> sim.initGenotype(pop, freq=[0.2, 0.8], subPops=[(0,0)])
>>> sim.initGenotype(pop, freq=[0.5, 0.5], subPops=[(0,1)])
>>> #
>>> # initialize by random allele frequency
>>> import random
>>> sim.initGenotype(pop, freq=lambda : random.random())
>>> printFreq(pop, range(5))
0.452, 0.649, 0.282, 0.239, 0.667
>>> # initialize with loci specific frequency. here
>>> # lambda loc: 0.01*loc is equivalent to
>>> # lambda loc: [0.01*loc, 1-0.01*loc]
>>> sim.initGenotype(pop,
...     freq=lambda loc: 0.01*loc)
>>> printFreq(pop, range(5))
0.000, 0.009, 0.018, 0.029, 0.041
>>> # initialize with VSP-specific frequency
>>> sim.initGenotype(pop,
...     freq=lambda vsp: [[0.2, 0.8], [0.5, 0.5]][vsp[1]],
...     subPops=[(0, 0), (0, 1)])
>>>
```

4.2.3 Initialize information fields (operator `InitInfo`)

Operator `InitInfo` and its function form `initInfo` initialize one or more information fields of all individuals or Individuals in selected (virtual) subpopulations using either a list of values or a Python function. If a value or a list of value is given, it will be used repeatedly to assign values of specified information fields of all applicable individuals. For example, `initInfo(pop, values=1, infoFields='x')` will assign value `1` to information field `x` of all individuals, and

```
initInfo(pop, values=[1, 2, 3], infoFields='x', subPops=[(0,1)])
```

will assign values 1, 2, 3, 1, 2, 3... to information field x of individuals in the second virtual subpopulation of subpopulation 0.

The values parameter also accepts a Python function. This feature is usually used to assign random values to an information field. For example, values=random.random would assign a random value between 0 and 1. If a function takes parameters, a lambda function can be used. For example,

```
initInfo(pop, lambda : random.randint(2, 5), infoFields=['x', 'y'])
```

assigns random integers between 2 and 5 to information fields x and y of all individuals in *pop*. Example 4.10 demonstrates these usages.

Listing 4.10: initialize information fields

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=[5], loci=[2], infoFields=['sex', 'age'])
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> sim.initSex(pop)
>>> sim.initInfo(pop, 0, subPops=[(0,0)], infoFields='sex')
>>> sim.initInfo(pop, 1, subPops=[(0,1)], infoFields='sex')
>>> sim.initInfo(pop, lambda: random.randint(20, 70), infoFields='age')
>>> sim.dump(pop, structure=False)
SubPopulation 0 (), 5 Individuals:
   0: FU 00 | 00 |  1 23
   1: FU 00 | 00 |  1 40
   2: MU 00 | 00 |  0 35
   3: MU 00 | 00 |  0 20
   4: MU 00 | 00 |  0 24
```

4.3 Expressions and statements

4.3.1 Output a Python string (operator PyOutput)

Operator PyOutput is a simple operator that prints a Python string when it is applied to a population. It is commonly used to print the progress of a simulation (e.g. PyOutput('start migration\n', at=200)) or output separators to beautify outputs from PyEval outputs (e.g. PyOutput('\n', rep=-1).

4.3.2 Execute Python statements (operator PyExec)

Operator PyExec executes Python statements in a population's local namespace when it is applied to that population. This operator is designed to execute short Python statements but multiple statements separated by newline characters are allowed.

Example 4.11 uses two PyExec operators to create and use a variable traj in each population's local namespace. The first operator initialize this variable as an empty list. During evolution, the frequency of allele 1 at locus 0 is calcuated (operator Stat) and appended to this variable (operator PyExec). The result is a trajectory of allele frequencies during evolution.

Listing 4.11: Execute Python statements during evolution

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(100, loci=1),
```

```
...     rep=2)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2, 0.8]),
...         sim.PyExec('traj=[]')
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.PyExec('traj.append(alleleFreq[0][1])'),
...     ],
...     gen=5
... )
(5L, 5L)
>>> # print Trajectory
>>> print(', '.join(['%.3f' % x for x in simu.dvars(0).traj]))
0.775, 0.790, 0.760, 0.750, 0.750
```

4.3.3 Evaluate and output Python expressions (operator `PyEval`)

Operator `PyEval` evaluate a given Python expression in a population's local namespace and output its return value. This operator has been widely used (e.g. Example 1.1, 3.15, 4.1 and 4.4) to output statistics of populations and report progress.

Two additional features of this operator may become handy from time to time. First, an optional Python statements (parameter *stmts*) can be specified which will be executed before the expression is evaluated. Second, the population being applied can be exposed in its own namespace as a variable (parameter *exposePop*). This makes it possible to access properties of a population other than its variables. Example 4.12 demonstrates both features. In this example, two statements are executed to count the number of unique parents in an offspring population and save them as variables `numFather` and `numMother`. The operator outputs these two variables alone with a generation number.

Listing 4.12: Evaluate a expression and statements in a population's local namespace.

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=1,
...     infoFields=['mother_idx', 'father_idx'])
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.ParentsTagger(),
...     ]),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.PyEval(r'"gen %d, #father %d, #mother %d\n"' \
...             ' % (gen, numFather, numMother)',
...             stmts="numFather = len(set(pop.indInfo('father_idx')))\n"
...                 "numMother = len(set(pop.indInfo('mother_idx')))",
...             exposePop='pop')
...     ],
...     gen=3
... )
gen 0, #father 439, #mother 433
```

```
gen 1, #father 433, #mother 432
gen 2, #father 449, #mother 420
3L
```

Note that the function form of this operator (`pyEval`) returns the result of the expression rather than writting it to an output.

4.3.4 Expression and statement involving individual information fields (operator `InfoEval` and `InfoExec`) *

Operators `PyEval` and `PyExec` work at the population level, using the local namespace of populations. Operator `InfoEval` and `InfoExec`, on the contraray, work at the individual level, using individual information fields (and population variables) as variables. In this case, individual information fields are copied to the population namespace one by one before expression or statements are executed for each individual. Optionally, the individual object can be exposed to these namespace using a user-specified name (parameter *exposeInd*). Individual information fields will be updated if the value of these fields are changed.

Operator `InfoEval` evaluates an expression and outputs its value. Operator `InfoExec` executes one or more statements and does not produce any output. Operator `InfoEval` is usually used to output individual information fields and properties in batch mode. It is faster and sometimes easier to use than corresponding for loop plus individual level operations. For example

- `InfoEval(r"'%.2f\t" % a')` outputs the value of information field a for all individuals, separated by tabs.
- `InfoEval('ind.sexChar()', exposeInd='ind')` outputs the sex of all individuals using an exposed individual object `ind`.
- `InfoEval('a+b**2')` outputs $a + b^2$ for information fields a and b for all individuals.

Example 4.13 demonstrates the use of this operator.

Listing 4.13: Evaluate expressions using individual information fields

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(20, loci=1, infoFields='a')
>>> pop.setVirtualSplitter(sim.InfoSplitter('a', cutoff=[3]))
>>> sim.initGenotype(pop, freq=[0.2, 0.8])
>>> pop.setIndInfo([random.randint(2, 5) for x in range(20)], 'a')
>>> sim.infoEval(pop, 'a', subPops=[(0, 0)]);print(' ')
2.02.02.02.02.02.02.0
>>> sim.infoEval(pop, 'ind.allele(0, 0)', exposeInd='ind');print(' ')
11011111111100111111
>>> # use sim.population variables
>>> pop.dvars().b = 5
>>> sim.infoEval(pop, '"%d " % (a+b)');print(' ')
8 7 7 10 10 7 8 8 10 7 9 7 7 8 10 10 10 7 9 9
```

Operator `InfoExec` is usually used to set individual information fields. For example

- `InfoExec('age += 1')` increases the age of all individuals by one.
- `InfoExec('risk = 2 if packPerYear > 10 else 1.5')` sets information field `risk` to `2` if `packPerYear` is greater than `10`, and `1.5` otherwise. Note that conditional expression is only available for Python version 2.5 or later.
- `InfoExec('a = b*c')` sets the value of information field `a` to the product of `b` and `c`.

Example 4.14 demonstrates the use of this operator, using its function form `infoExec`.

Listing 4.14: Execute statements using individual information fields

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=1, infoFields=['a', 'b', 'c'])
>>> sim.initSex(pop)
>>> sim.initGenotype(pop, freq=[0.2, 0.8])
>>> sim.infoExec(pop, 'a=1')
>>> print(pop.indInfo('a')[:10])
(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
>>> sim.infoExec(pop, 'b=ind.sex()', exposeInd='ind')
>>> print(pop.indInfo('b')[:10])
(2.0, 2.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 2.0, 2.0)
>>> sim.infoExec(pop, 'c=a+b')
>>> print(pop.indInfo('c')[:10])
(3.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 3.0, 3.0, 3.0)
>>> pop.dvars().d = 5
>>> sim.infoExec(pop, 'c+=d')
>>> print(pop.indInfo('c')[:10])
(8.0, 8.0, 7.0, 7.0, 7.0, 7.0, 7.0, 8.0, 8.0, 8.0)
>>> # the operator can update population variable as well
>>> sim.infoExec(pop, 'd+=c*c')
>>> print(pop.dvars().d)
5835.0
```

Note that a statement can also be specified for operator `InfoEval`, which will be executed before an expression is evaluated.

4.3.5 Using functions in external modules in simuPOP expressions and statements

All simuPOP expressions and statements are evaluated in a population's local namespace, which is a dictionary with no access to external modules. If you would like to use external modules (e.g. functions from the `random` module), you will have to import them to the namespace explicitly, using something like

```
exec('import random', pop.vars(), pop.vars())
```

before you evolve the population.

Example 4.15 demonstrates the application of this technique. This example imports the `time` module in the population's local namespace and set `init_time` and `last_time` before evolution. During evolution, an `IfElse` operator is used to output the status of the simulation for every 5 seconds using expression `time.time() - last_time > 5`. `last_time` is reset using the `PyExec` operator. The evolution will last 20 seconds and be terminated by the Terminator with expression `time.time() - init_time > 20`.

Listing 4.15: Write the status of an evolutionary process every 10 seconds

```
>>> import simuPOP as sim
>>> import time
>>> pop = sim.Population(1000, loci=10)
>>> pop.dvars().init_time = time.time()
>>> pop.dvars().last_time = time.time()
>>> exec('import time', pop.vars(), pop.vars())
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(),
```

```
...     postOps=[
...         sim.IfElse('time.time() - last_time > 5', [
...             sim.PyEval(r'"Gen: %d\n" % gen'),
...             sim.PyExec('last_time = time.time()')
...             ]),
...         sim.TerminateIf('time.time() - init_time > 20')
...     ]
... )
Gen: 20969
Gen: 41813
Gen: 62787
83791L
>>>
```

4.4 Demographic changes

A mating scheme controls the size of an offspring generation using parameter `subPopSize`. This parameter has been described in detail in section 5.1.1. In summary,

- The subpopulation sizes of the offspring generation will be the same as the parental generation if subPopSize is not set.
- The offspring generation will have a fixed size if `subPopSize` is set to a number (no subpopulation) or a list of subpopulation sizes.
- The subpopulation sizes of an offspring generation will be determined by the return value of a demographic function if `subPopSize` is set to such a function (a function that returns subpopulation sizes at each generation).

Note: Parameter `subPopSize` only controls subpopulation sizes of an offspring generation immediately after it is generated. population or subpopulation sizes could be changed by other operators. During mating, a mating scheme goes through each parental subpopulation and populates its corresponding offspring subpopulation. This implies that

- Parental and offspring populations should have the same number of subpopulations.
- Mating happens strictly within each subpopulation.

This section will introduce several operators that allow you to move dndividuals across the boundary of subpopulations (migration), and change the number of subpopulations during evolution (split and merge). Please refer to 5.1.1 (control the size of the offspring generation section of chapter mating scheme) for more details. For more advanced demographic models, please refer to the `simuPOP.demography` module.

4.4.1 Migration (operator `Migrator`)

Migration by probability

Operator `Migrator` (and its function form `migrate`) migrates individuals from one subpopulation to another. The key parameters are

- *from* subpopulations (parameter `subPops`). A list of subpopulations from which individuals migrate. Default to all subpopulations.
- *to* subpopulations (parameter `toSubPops`). A list of subpopulations to which individuals migrate. Default to all subpopulations. **A new subpopulation ID can be specified to create a new subpopulation from migrants.**

- A migration rate matrix (parameter `rate`). A m by n matrix (a nested list in Python) that specifies migration rate from each source to each destination subpopulation. That is to say, $\text{rate}_{i,j}$ specifies migration rate from subPops_i to toSubPops_j. Needless to say, m and n are determined by the number of *from* and *to* subpopulations.

Example 4.16 demonstrate the use of a `Migrator` to migrate individuals between three subpopulations. Note that

- Operator `Migrator` relies on an information field `migrate_to` (configurable) to record destination subpopulation of each individual so this information field needs to be added to a population befor migration.
- Migration rates to subpopulation themselves are determined automatically so they can be left unspecified.

Listing 4.16: Migration by probability

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000]*3, infoFields='migrate_to')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=sim.Migrator(rate=[
...             [0, 0.1, 0.1],
...             [0, 0, 0.1],
...             [0, 0.1, 0]
...         ]),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval('subPopSize'),
...         sim.PyOutput('\n')
...     ],
...     gen = 5
... )
[762, 1108, 1130]
[601, 1175, 1224]
[490, 1233, 1277]
[395, 1282, 1323]
[320, 1300, 1380]
5L
```

Migration by proportion and counts

Migration rate specified in the rate parameter in Example 4.16 is intepreted as probabilities. That is to say, a migration rate $r_{m,n}$ is interpreted as the probability at which any individual in subpopulation m migrates to subpopulation n. The exact number of migrants are randomly distributed.

If you would like to specify exactly how many migrants migrate from a subpopulation to another, you can specify parameter `mode` of operator `Migrator` to `BY_PROPORTION` or `BY_COUNTS`. The `BY_PROPORTION` mode interpret $r_{m,n}$ as proportion of individuals who will migrate from subpopulation m to n so the number of $m \rightarrow n$ migrant will be exactly $r_{m,n} \times$subPopSize(m). In the `BY_COUNTS` mode, $r_{m,n}$ is interpretted as number of migrants, regardless the size of subpopulation m. Example 4.17 demonstrates these two migration modes, as well as the use of parameters `subPops` and `toSubPops`.

Listing 4.17: Migration by proportion and count

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000]*3, infoFields='migrate_to')
>>> pop.evolve(
```

```
...     initOps=sim.InitSex(),
...     preOps=sim.Migrator(rate=[[0.1], [0.2]],
...             mode=sim.BY_PROPORTION,
...             subPops=[1, 2],
...             toSubPops=[3]),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval('subPopSize'),
...         sim.PyOutput('\n')
...     ],
...     gen = 5
... )
[1000, 900, 800, 300]
[1000, 810, 640, 550]
[1000, 729, 512, 759]
[1000, 657, 410, 933]
[1000, 592, 328, 1080]
5L
>>> #
>>> pop.evolve(
...     preOps=sim.Migrator(rate=[[50, 50], [100, 50]],
...             mode=sim.BY_COUNTS,
...             subPops=[3, 2],
...             toSubPops=[2, 1]),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval('subPopSize'),
...         sim.PyOutput('\n')
...     ],
...     gen = 5
... )
[1000, 692, 328, 980]
[1000, 792, 328, 880]
[1000, 892, 328, 780]
[1000, 992, 328, 680]
[1000, 1092, 328, 580]
5L
```

Theoretical migration models

To facilitate the use of widely used theoretical migration models, a few functions are defined in module `simuPOP.demography` 6.3.1. These functions generate migration matrixes that can be plugged in to the `Migrator` operator.

migrate from virtual subpopulations *

Under a realistic eco-social settings, individuals in a subpopulation rarely have the same probability to migrate. Genetic evidence has shown that female has a higher migrate rate than male in humans, perhaps due to migration patterns related to inter-population marriages. Such sex-biased migration also happens in other large migration events such as slave trade.

It is easy to simulate most of such complex migration models by migrating from virtual subpopulations. For example, if you define virtual subpopulations by sex, you can specify different migration rates for males and females and control the proportion of males among migrants, by specifying virtual subpopulations in parameter `subPops`. Parameter `toSubPops` does not accept virtual subpopulations because you cannot, for example, migrate to females in a subpopulation.

Example 4.18 demonstrate a sex-biased migration model where males dominate migrants from subpopulation 0. To avoid confusing, this example uses the proportion migration mode. At the beginning of the first generation, there are 500 males and 500 females in each subpopulation. A 10% male migration rate and 5% female migration rate leads to 50 male migrants and 25 female migrants. Subpopulation sizes and number of males in each subpopulation before mating are therefore:

- Subpopulation 0: male 500-50, female 500-25, total 925
- Subpopulation 1: male 500+50, female 500+25, total 1075

Note that the unspecified *to* subpopulations are subpopulation 0 and 1, which cannot be virtual.

Listing 4.18: Migration from virtual subpopulations

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000]*2, infoFields='migrate_to')
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.evolve(
...     # 500 males and 500 females
...     initOps=sim.InitSex(sex=[sim.MALE, sim.FEMALE]),
...     preOps=[
...         sim.Migrator(rate=[
...             [0, 0.10],
...             [0, 0.05],
...             ],
...             mode = sim.BY_PROPORTION,
...             subPops=[(0, 0), (0, 1)]),
...         sim.Stat(popSize=True, numOfMales=True, vars='numOfMales_sp'),
...         sim.PyEval(r"'%d/%d\t%d/%d\n' % (subPop[0]['numOfMales'], subPopSize[0], "
...             "subPop[1]['numOfMales'], subPopSize[1])"),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(popSize=True, numOfMales=True, vars='numOfMales_sp'),
...         sim.PyEval(r"'%d/%d\t%d/%d\n' % (subPop[0]['numOfMales'], subPopSize[0], "
...             "subPop[1]['numOfMales'], subPopSize[1])"),
...     ],
...     gen = 2
... )
450/925 550/1075
426/925 520/1075
384/859 562/1141
425/859 582/1141
2L
```

Arbitrary migration models **

If none of the described migration mothods fits your need, you can always resort to manual migration. One such example is when you need to mimick an existing evolutionary scenario so you know exactly which subpopulation each individual will migrate to.

Manual migration is actually very easy. All you need to do is specifying the destination subpopulation of all individuals in the *from* subpopulations (parameter `subPops`), using an information field (usually `migrate_to`). You can then call the `Migrator` using `mode=BY_IND_INFO`. Example 4.19 shows how to manually move individuals around. This example uses the function form of `Migrator`. You usually need to use a Python operator to set destination subpopulations if you would like to manually migrate individuals during an evolutionary process.

Listing 4.19: Manual migration

```
>>> import simuPOP as sim
>>> pop = sim.Population([10]*2, infoFields='migrate_to')
>>> pop.setIndInfo([0, 1, 2, 3]*5, 'migrate_to')
>>> sim.migrate(pop, mode=sim.BY_IND_INFO)
>>> pop.subPopSizes()
(5L, 5L, 5L, 5L)
```

Note: individuals with an invalid destination subpopulation ID (e.g. an negative number) will be discarded silently. Although not recommended, this feature can be used to remove individuals from a subpopulation.

4.4.2 Migration using backward migration matrix (operator `BackwardMigrator`)

Backward migration matrices are widely used in theoretical population genetics and coalescent based simulations. Instead of specifying the probability of migrating from another subpopulation to another (namely how migration happens), such matrices specify the probability that individuals in a subpopulation originate from others (namely the result of migration). simuPOP simulates such models by converting backward migration matrices to foward ones using the theory describes below. Due to the limit of such models, simuPOP cannot simulate migration from/to virtual subpopulatons, creation of new subpopulation, different source and destination subpopulations, and will generate an error if the conversion process fails.

To explain the differences between forward and backward migration matrices, let us assume that there are d subpopulations with population sizes $S = [S_1, S_2, ..., S_d]$, and a forward migration matrix

$$F = \begin{bmatrix} f_{11} & f_{12} & \cdots & f_{1d} \\ f_{21} & f_{22} & \cdots & f_{2d} \\ \vdots & & & \vdots \\ f_{d1} & f_{d2} & \cdots & f_{dd} \end{bmatrix}$$

where f_{ij} is the probability that an individual will migration from subpopulation i to j. After migration happens, subppulation sizes are changed to $S' = [S'_1, S'_2, ..., S'_d]$, and the origin of individuals in each subpopulation can be described by the backward migration matrix

$$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1d} \\ b_{21} & b_{22} & \cdots & b_{2d} \\ \vdots & & & \vdots \\ b_{d1} & b_{d2} & \cdots & b_{dd} \end{bmatrix}$$

where b_{ij} is the probability that an individual in subpopulation i originates from subpopulation j.

These qualities can be derived from original population sizes and the forward migration matrix. That is to say, the size of new subpopulation k is the sum of all migrants to this subpopulation

$$S'_k = \sum_{i=1}^{d} S_i f_{ik}$$

and the size of the original population k is the sum of all migrants from this subpopulation

$$S_k = \sum_{i=1}^{d} S'_i b_{ik}$$

and the composition of subpopulation k (e.g. individuals originate from subpopulation j) is

$$b_{kj} = \frac{S_j f_{jk}}{S'_k}$$

In matrix form, these formulas can be written as

$$S' = F^T S$$

$$S = B^T S'$$

and

$$B = diag(S')^{-1} F^T diag(S)$$

Therefore, given a backward migration matrix B and current population size S, we can derive a forward migration matrix using

$$S' = \left(B^T\right)^{-1} S$$

and

$$F = diag(S)^{-1} B^T diag(S')$$

Note that $F = B$ is always true if B is symmetric and $S_i = S_j$ (equal subpopulation size) so simuPOP will use B directly in this case. Also note that B might not be inversable and S' and F might be invalid (e.g. negative population size or forward migration rate) for given B and S. simuPOP will terminate with an error message in these cases.

The following example 4.20 demonstrates how to use a backward migration matrix to perform migration. It initializes all individuals with indexes of subpopulations they belong to before migration and calculates the percent of individuals from each source population using a PyOperator with function originOfInds. The so-called overseved backward migration matrix is similar to specified migration matrix despite of stochastic effects. This example also uses turnOnDebug function to let the operator print the expected subpopulation size (S') and calculate forward migration matrix (F) at each generation, which, as expected, vary from generation to generation.

Listing 4.20: Migration using a backward migration matrix

```
>>> import simuPOP as sim
>>> sim.turnOnDebug('DBG_MIGRATOR')
>>> pop = sim.Population(size=[10000, 5000, 8000], infoFields=['migrate_to', 'migrate_from'])
>>> def originOfInds(pop):
...     print('Observed backward migration matrix at generation {}'.format(pop.dvars().gen))
...     for sp in range(pop.numSubPop()):
...         # get source subpop for all individuals in subpopulation i
...         origins = pop.indInfo('migrate_from', sp)
...         spSize = pop.subPopSize(sp)
...         B_sp = [origins.count(j) * 1.0 /spSize for j in range(pop.numSubPop())]
...         print('    ' + ', '.join(['{:.3f}'.format(x) for x in B_sp]))
...     return True
...
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=
...         # mark the source subpopulation of each individual
...         [sim.InitInfo(i, subPops=i, infoFields='migrate_from') for i in range(3)] + [
...         # perform migration
...         sim.BackwardMigrator(rate=[
```

```
...             [0, 0.04, 0.02],
...             [0.05, 0, 0.02],
...             [0.02, 0.01, 0]
...         ]),
...         # calculate and print observed backward migration matrix
...         sim.PyOperator(func=originOfInds),
...         # calculate population size
...         sim.Stat(popSize=True),
...         # and print it
...         sim.PyEval(r'"Pop size after migration: {}\n".format(", ".join([str(x) for x in subPopSize]))'),
...         ],
...     matingScheme=sim.RandomMating(),
...     gen = 5
... )
Expected next population size is 10211.4, 4851.8, 7936.84
Forward migration matrix is 0.959867, 0.024259, 0.0158737, 0.0816908, 0.902435, 0.0158737, 0.0255284, 0.0121295, 0.962342
Observed backward migration matrix at generation 0
    0.939, 0.040, 0.021
    0.051, 0.927, 0.022
    0.020, 0.010, 0.969
Pop size after migration: 10218, 4859, 7923
Expected next population size is 10453.6, 4690.64, 7855.79
Forward migration matrix is 0.961671, 0.0229529, 0.0153764, 0.0860553, 0.897777, 0.0161675, 0.0263879, 0.0118406, 0.961772
Observed backward migration matrix at generation 1
    0.942, 0.038, 0.020
    0.049, 0.932, 0.020
    0.023, 0.010, 0.968
Pop size after migration: 10417, 4706, 7877
Expected next population size is 10675.5, 4517.1, 7807.37
Forward migration matrix is 0.963329, 0.0216814, 0.0149897, 0.0907397, 0.89267, 0.0165902, 0.0271056, 0.0114691, 0.961425
Observed backward migration matrix at generation 2
    0.942, 0.039, 0.020
    0.048, 0.930, 0.022
    0.020, 0.010, 0.970
Pop size after migration: 10660, 4536, 7804
Expected next population size is 10946, 4323.5, 7730.53
Forward migration matrix is 0.965217, 0.0202791, 0.0145038, 0.0965253, 0.886432, 0.0170426, 0.0280522, 0.0110802, 0.960868
Observed backward migration matrix at generation 3
    0.940, 0.040, 0.020
    0.050, 0.930, 0.020
    0.020, 0.011, 0.969
Pop size after migration: 10942, 4321, 7737
Expected next population size is 11260.4, 4079.55, 7660
Forward migration matrix is 0.967357, 0.0186417, 0.0140011, 0.104239, 0.878033, 0.0177274, 0.0291081, 0.0105456, 0.960346
Observed backward migration matrix at generation 4
    0.937, 0.043, 0.021
    0.046, 0.933, 0.021
    0.019, 0.009, 0.972
Pop size after migration: 11331, 4042, 7627
5L
```

4.4.3 Split subpopulations (operators `SplitSubPops`)

Operator `SplitSubPops` splits one or more subpopulations into finer subpopulations. It can be used to simulate populations that originate from the same founder population. For example, a population of size 1000 in Example 4.21 is split into three subpopulations of sizes 300, 300 and 400 respectively, after evolving as a single population for two generations.

Listing 4.21: Split subpopulations by size

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000)
>>> pop.evolve(
...     preOps=[
...         sim.SplitSubPops(subPops=0, sizes=[300, 300, 400], at=2),
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     matingScheme=sim.RandomSelection(),
...     gen = 4
... )
Gen 0:  [1000]
Gen 1:  [1000]
Gen 2:  [300, 300, 400]
Gen 3:  [300, 300, 400]
4L
```

Operator `SplitSubPops` splits a subpopulation by sizes of the resulting subpopulations. It is often easier to do so with proportions. In addition, if a demographic function is used, you should make sure that the number of subpopulations will be the same before and after mating at any generation. One way of doing this is to apply a `SplitSubPops` operator at the right generation. Example 4.22 demonstrates such an evolutionary scenario. However, it is often easier to split the population in the demographic function in such case (see section 5.1.2 for details).

Listing 4.22: Split subpopulations by proportion

```
>>> import simuPOP as sim
>>> def demo(gen, pop):
...     if gen < 2:
...         return 1000 + 100 * gen
...     else:
...         return [x + 50 * gen for x in pop.subPopSizes()]
...
>>> pop = sim.Population(1000)
>>> pop.evolve(
...     preOps=[
...         sim.SplitSubPops(subPops=0, proportions=[.5]*2, at=2),
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     matingScheme=sim.RandomSelection(subPopSize=demo),
...     gen = 4
... )
Gen 0:  [1000]
Gen 1:  [1000]
Gen 2:  [550, 550]
Gen 3:  [650, 650]
4L
```

Either by *sizes* or by *proportions*, individuals in a subpopulation are divided randomly. It is, however, also possible to split subpopulations according to individual information fields. In this case, individuals with different values at a given information field will be split into different subpopulations. This is demonstrated in Example 4.23 where the function form of operator `SplitSubPops` is used.

Listing 4.23: Split subpopulations by individual information field

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population([1000]*3, subPopNames=['a', 'b', 'c'], infoFields='x')
>>> pop.setIndInfo([random.randint(0, 3) for x in range(1000)], 'x')
>>> print(pop.subPopSizes())
(1000L, 1000L, 1000L)
>>> print(pop.subPopNames())
('a', 'b', 'c')
>>> sim.splitSubPops(pop, subPops=[0, 2], infoFields=['x'])
>>> print(pop.subPopSizes())
(274L, 247L, 233L, 246L, 1000L, 274L, 247L, 233L, 246L)
>>> print(pop.subPopNames())
('a', 'a', 'a', 'a', 'b', 'c', 'c', 'c', 'c')
```

4.4.4 Merge subpopulations (operator `MergeSubPops`)

Operator `MergeSubPops` merges specified subpopulations into a single subpopulation. This operator can be used to simulate admixed populations where two or more subpopulations merged into one subpopulation and continue to evolve for a few generations. Example 4.24 simulates such an evolutionary scenario. A demographic model could be added similar to Example 4.22.

Listing 4.24: Merge multiple subpopulations into a single subpopulation

```
>>> import simuPOP as sim
>>> pop = sim.Population([500]*2)
>>> pop.evolve(
...     preOps=[
...         sim.MergeSubPops(subPops=[0, 1], at=3),
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     matingScheme=sim.RandomSelection(),
...     gen = 5
... )
Gen 0:  [500, 500]
Gen 1:  [500, 500]
Gen 2:  [500, 500]
Gen 3:  [1000]
Gen 4:  [1000]
5L
```

4.4.5 Resize subpopulations (operator `ResizeSubPops`)

Whenever possible, it is recommended that subpopulation sizes are changed naturally, namely through the population of an offspring generation. However, it is sometimes desired to change the size of a population forcefully. Examples of such applications include immediate expansion of a small population before evolution, and the simulation of sudden

population size change caused by natural disaster. By default, new individuals created by such sudden population expansion get their genotype from existing individuals. Example 4.25 shows a scenario where two subpopulations expand instantly at generation 3.

Listing 4.25: Resize subpopulation sizes

```
>>> import simuPOP as sim
>>> pop = sim.Population([500]*2)
>>> pop.evolve(
...     preOps=[
...         sim.ResizeSubPops(proportions=(1.5, 2), at=3),
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     matingScheme=sim.RandomSelection(),
...     gen = 5
... )
Gen 0:  [500, 500]
Gen 1:  [500, 500]
Gen 2:  [500, 500]
Gen 3:  [750, 1000]
Gen 4:  [750, 1000]
5L
```

4.4.6 Time-dependent migration rate

In evolutionary scenarios with complex demographic models, number of subpopulations and migration rate might change from generation to generation. For example, if one of the subpopulations is split into two, the migration matrix has to be changed to accommendate increased number of subpopulations.

If there are a limited number of demographic changes and a few number of pre-determined migration matrices. You can use a number of `Migrators` that are applied at different generations. For example, you can use the following operators to apply the first migration scheme during first ten generations (0, ..., 9), and the second migration scheme during the rest of the evolutionary process:

```
preOps=[
    Migrator(rate=M1, end=9),
    Migrator(rate=M2, begin=10),
]
```

If changes of demographies are frequent or stochastic so that migration matrices can only be determined programmatically, it is easier to use a `PyOperator` to migrate populations using the function form of a `Migrator`. This is demonstrated in Example 4.26 where migration matrixes are computed dynamically due to random split of subpopulations.

Listing 4.26: Varying migration rate

```
>>> import simuPOP as sim
>>>
>>> from simuPOP.utils import migrIslandRates
>>> import random
>>>
>>> def demo(pop):
...   # this function randomly split populations
...   numSP = pop.numSubPop()
...   if random.random() > 0.3:
...       pop.splitSubPop(random.randint(0, numSP-1), [0.5, 0.5])
```

```
...   return pop.subPopSizes()
...
>>> def migr(pop):
...   numSP = pop.numSubPop()
...   sim.migrate(pop, migrIslandRates(0.01, numSP))
...   return True
...
>>> pop = sim.Population(10000, infoFields='migrate_to')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=[
...         sim.PyOperator(func=migr),
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     matingScheme=sim.RandomMating(subPopSize=demo),
...     gen = 5
... )
Gen 0:  [10000]
Gen 1:  [4982, 5018]
Gen 2:  [4980, 5020]
Gen 3:  [4978, 5022]
Gen 4:  [4925, 2528, 2547]
5L
```

4.5 Genotype transmitters

4.5.1 Generic genotype transmitters (operators `GenoTransmitter`, `CloneGenoTransmitter`, `MendelianGenoTransmitter`, `SelfingGenoTransmitter`, `HaplodiploidGenoTransmitter`, and `MitochondrialGenoTransmitter`) *

A number of during-mating operators are defined to transmit genotype from parent(s) to offspring. They are rarely used or even seen directly because they are used as genotype transmitters of mating schemes.

- `GenoTransmitter`: This genotype transmitter is usually used by customized genotype transmitters because it provides some utility functions that are more efficient than their Pythonic counterparts.
- `CloneGenoTransmitter`: Copy all genotype on non-customized chromosomes from a parent to an offspring. It also copies parental sex to the offspring because sex can be genotype determined. This genotype transmitter is used by mating scheme `CloneMating`. This genotype transmitter can be applied to populations of **any ploidy** type. If you would like to copy part of the chromosomes, or customized chromosomes, a parameter chroms could be used to specify chromosomes to copy.
- `MendelianGenoTransmitter`: Copy genotypes from two parents (a male and a female) to an offspring following Mendel's laws, used by mating scheme `RandomMating`.This genotype transmitter can only be applied to **diploid** populations.
- `SelfingGenoTransmitter`: Copy genotypes from one parent to an offspring using self-fertilization, used by mating scheme `SelfMating`. This genotype transmitter can only be applied to **diploid** populations.
- `HaplodiploidGenoTransmitter`: Set genotype to male and female offspring differently in a haplodiploid population, used by mating scheme `HaplodiploidMating`. This genotype transmitter can only be applied to **haplodiploid** populations.

- `MitochondrialGenoTransmitter`: Treat a single mitochondrial chromosome, or all customized chromosomes, or specified chromosomes as mitochondrial chromosomes and transmit maternal mitochondrial chromosomes randomly to an offspring. This genotype transmitter can be applied to populations of **any ploidy** type. It trasmits the first homologous copy of chromosomes maternally and clears alleles on other homologous copies of chromosomes of an offspring.

4.5.2 Recombination (Operator `Recombinator`)

The generic genotype transmitters do not handle genetic recombination. A genotype transmitter `Recombinator` is provided for such purposes, and can be used with `RandomMating` and `SelfMating` (replace `MendelianGenoTransmitter` and `SelfingGenoTransmitter` used in these mating schemes).

Recombination rate is implemented **between adjacent markers**. There can be only one recombination event between adjacent markers no matter how far apart they are located on a chromosome. In practise, a `Recombinator` goes along chromosomes and determine, between each adjacent loci, whether or not a recombination happens.

Recombination rates could be specified in the following ways:

1. If a single recombination rate is specified through paramter `rates`, it will be the recombination rate between all adjacent loci, regardless of loci position.

2. If recombination happens only after certain loci, you can specify these loci using parameter `loci`. For example,

   ```
   Recombinator(rates=0.1, loci=[2, 5])
   ```

   recombines a chromosome only **after** loci 2 (between 2 and 3) and 5 (between 5 and 6).

3. If parameter `loci` is given with a list of loci, different recombination rate can be given to each of them. The two lists should have the same length. For example

   ```
   Recombinator(rates=[0.1, 0.05], loci=[2, 5])
   ```

   uses two different recombination rates after loci 2 and 5.

4. If parameter `loci` is not given (default to `loci=ALL_AVAIL`) but a list of recombination rates is assigned, the rates will be assigned to each locus. The length of prameter `rates` should equal to total number of loci but the recombiantion rates for the locus at the end of each chromosome will be ignored (assumed to be 0.5). For example

   ```
   Recombinator(rates=[0.1]*5 + [0.2]*5)
   ```

   uses two different recombination rates for two chromosomes with 5 loci.

5. If recombination rates vary across your chromosomes, a long list of `rate` and `loci` may be needed to specify recombination rates one by one. An alternative method is to specify a **recombination intensity**. Recombination rate between two adjacent loci is calculated as the product of this intensity and distance between them. For example, if you apply operator

   ```
   Recombinator(intensity=0.1)
   ```

   to a population

   ```
   Population(size=100, loci=[4], lociPos=[0.1, 0.2, 0.4, 0.8])
   ```

   The recombination rates between adjacent markers will be `0.1*0.1`, `0.1*0.2` and `0.1*0.4` respectively.

Listing 4.27: Genetic recombination at all and selected loci

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(size=[1000], loci=[100]),
...     rep=2)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[0]*100 + [1]*100)
...     ],
...     matingScheme=sim.RandomMating(ops = [
...         sim.Recombinator(rates=0.01, reps=0),
...         sim.Recombinator(rates=[0.01]*10, loci=range(50, 60), reps=1),
...     ]),
...     postOps=[
...         sim.Stat(LD=[[40, 55], [60, 70]]),
...         sim.PyEval(r'"%d:\t%.3f\t%.3f\t" % (rep, LD_prime[40][55], LD_prime[60][70])'),
...         sim.PyOutput('\n', reps=-1)
...     ],
...     gen = 5
... )
0:      0.742   0.816   1:      0.898   1.000
0:      0.649   0.750   1:      0.848   1.000
0:      0.598   0.683   1:      0.816   1.000
0:      0.476   0.642   1:      0.794   1.000
0:      0.413   0.597   1:      0.782   1.000
(5L, 5L)
```

Example 4.27 demonstrates how to specify recombination rates for all loci or for specified loci. In this example, two replicates of a population are evolved, subject to two different Recombinators. The first Recombinator applies the same recombination rate between all adjacent loci, and the second Recombinator recombines only after loci 50 - 59. Because there is no recombination event between loci 60 and 70 for the second replicate, linkage disequilibrium values between these two loci does not decrease as what happens in the first replicate.

Listing 4.28: Genetic recombination rates specified by intensity

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000], loci=3, lociPos=[0, 1, 1.1])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[0]*3 + [1]*3)
...     ],
...     matingScheme=sim.RandomMating(ops=sim.Recombinator(intensity=0.01)),
...     postOps=[
...         sim.Stat(LD=[[0, 1], [1, 2]]),
...         sim.PyEval(r'"%.3f\t%.3f\n" % (LD_prime[0][1], LD_prime[1][2])', step=10)
...     ],
...     gen = 50
... )
0.988   0.998
0.912   0.996
0.836   0.991
0.896   0.982
0.814   0.991
50L
```

Example 4.28 demonstrates the use of the `intensity` parameter. In this example, the distances between the first two loci and the latter two loci are 1 and 0.1 respectively. This leads recombination rates 0.01 and 0.001 respectively with a recombination intensity 0.01. Consequently, LD between the first two loci decay much faster than the latter two.

If more advanced recombination model is desired, a customized genotype transmitter can be used. For example, Example 5.25 uses two Recombinators to implement sex-specific recombination.

Note: Both loci positions and recombination intensity are unitless. You can assume different unit for loci position and recombination intensity as long as the resulting recombination rate makes sense.

4.5.3 Gene conversion (Operator `Recombinator`) *

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

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

In practise, gene conversion can be considered as a double recombination event. That is to say, when a recombination event happens, it has certain probability to trigger a second recombination event along the chromosome. The distance between the two locations where recombination events happen is the tract length of this conversion event.

The probability at which gene conversion happens, and how tract length is determined is specify using parameter `convMode` of a Recombinator. This parameter can be

- `NoConversion` No gene conversion. (default)
- `(NUM_MARKERS, prob, N)` Convert a fixed number `N` of markers at probability `prob`.
- `(TRACT_LENGTH, prob, N)` Convert a fixed length `N` of chromosome regions at probability `prob`. This can be used when markers are not equally spaced on chromosomes.
- `(GEOMETRIC_DISTRIBUTION, prob, p)` When a conversion event happens at probability `prob`, convert a random number of markers, with a geometric distribution with parameter `p`.
- `(EXPONENTIAL_DISTRIBUTION, prob, p)` When a conversion event happens at probability `prob`, convert a random length of chromosome region, using an exponential distribution with parameter `p`.

Note that

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

Example 4.29 compares two Recombinators. The first Recombinator is a regular Recombinator that recombine between loci 50 and 51. The second Recombinator is a conversion operator because every recombination event will become a conversion event (prob=1). Because a second recombination event will surely happen between loci 60 and 61, there will be either no or double recombination events between loci 40, 70. LD between these two loci therefore does not decrease, although LD between locus 55 and these two loci will decay.

Listing 4.29: Gene conversion

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(size=[1000], loci=[100]),
...     rep=2)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[0]*100 + [1]*100)
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.Recombinator(rates=0.01, loci=50, reps=0),
...         sim.Recombinator(rates=0.01, loci=50, reps=1, convMode=(sim.NUM_MARKERS, 1, 10)),
...     ]),
...     postOps=[
...         sim.Stat(LD=[[40, 55], [40, 70]]),
...         sim.PyEval(r'"%d:\t%.3f\t%.3f\t" % (rep, LD_prime[40][55], LD_prime[40][70])'),
...         sim.PyOutput('\n', reps=-1)
...     ],
...     gen = 5
... )
0:      0.980   0.980   1:      0.978   1.000
0:      0.966   0.966   1:      0.976   1.000
0:      0.950   0.950   1:      0.968   1.000
0:      0.922   0.922   1:      0.970   1.000
0:      0.913   0.913   1:      0.969   1.000
(5L, 5L)
```

4.5.4 Tracking all recombination events **

To understand the evolutionary history of a simulated population, it is sometimes needed to track down all ancestral recombination events. In order to do that, you will first need to give an unique ID to each individual so that you could make sense of the dumped recombination events. Although this is routinely done using operator `IdTagger` (see example 4.65 for details), it is a little tricky here because you need to place the during-mating `IdTagger` before a `Recombinator` in the `ops` parameter of a mating scheme so that offspring ID could be set and outputted correctly.

After setting the name of the ID field (usually `ind_id`) to the `infoField` parameter of a `Recombinator`, it can dump a list of recombinatin events (loci after which recombinatin events happened) for each set of homologous chromosomes of an offspring. Each line is in the format of

```
offspringID parentID startingPloidy rec1 rec2 ....
```

Example 4.30 gives an example how the output looks like.

Listing 4.30: Tracking all recombination events

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=[1000, 2000], infoFields='ind_id')
>>> pop.evolve(
...     initOps=[
```

```
...         sim.InitSex(),
...         sim.IdTagger(),
...     ],
...     matingScheme=sim.RandomMating(ops = [
...         sim.IdTagger(),
...         sim.Recombinator(rates=0.001, output='>>rec.log', infoFields='ind_id')]),
...     gen = 5
... )
5L
>>> rec = open('rec.log')
>>> # print the first three lines of the log file
>>> print(''.join(rec.readlines()[:4]))
1001 642 0 381 999 1490
1001 250 1 908 999 1315 2134
1002 847 1 999
1002 91 0 975 999 1245 2546
```

4.6 Mutation

A mutator (a mutation operator) mutates alleles at certain loci from one allele to another. Because alleles are simple non-nagative numbers that can be intrepreted as nucleotides, codons, squences of nucleotides or even genetic deletions, appropriate mutation models have to be chosen for different types of loci. Please refer to Section 3.1 for a few examples.

A mutator will mutate alleles at all loci unless parameter `loci` is used to specify a subset of loci. Different mutators have different concepts and forms of mutation rates. If a mutator accepts only a single mutation rate (which can be in the form of a list or a matrix), it uses parameter `rate` and applies the same mutation rate to all loci. If a mutator accepts a list of mutation rates (each of which is a single number), it uses parameter `rates` and applies different mutation rates to different loci if multiple loci are specified. Note that parameter `rates` also accepts single form inputs (e.g. `rates=0.01`) in which case the same mutation rate will be applied to all loci.

4.6.1 Mutation models specified by rate matrixes (`MatrixMutator`)

A mutation model can be defined as a **mutation rate matrix** $(p_{ij})_{n\times n}$ where p_{ij} is the probability that an allele i mutates to j per generation per locus. Although mathematical formulation of p_{ij} are sometimes unscaled, simuPOP assumes $\sum_{j=0}^{n-1} p_{ij} = 1$ for all i and requires such rate matrixes in the specification of a mutation model. p_{ii} of such a matrix are ignored because they are automatically calculated from $p_{ii} = 1 - \sum_{j\neq i} p_{ij}$.

A `MatrixMutator` is defined to mutate between alleles 0, 1, ..., $n-1$ according to a given rate matrix. Conceptually speaking, this mutator goes through each mutable allele and mutates it to allele $0, 1, .., n-1$ according to probabilities p_{ij}, $j = 0, ..., n-1$. Most alleles will be kept intact because mutations usually happen at low probability (with p_{ii} close to 1). For example, Example 4.31 simulates a locus with 3 alleles. Because the rate at which allele 2 mutats to alleles 0 and 1 is higher than the rate alleles 0 and 2 mutate to allele 2, the frequency of allele 2 decreases over time.

Listing 4.31: General mutator specified by a mutation rate matrix

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[2000], loci=1)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2, 0.3, 0.5])
...     ],
```

```
...     preOps=sim.MatrixMutator(rate = [
...             [0, 1e-5, 1e-5],
...             [1e-4, 0, 1e-4],
...             [1e-3, 1e-3, 0]
...         ]),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0, step=100),
...         sim.PyEval(r"', '.join(['%.3f' % alleleFreq[0][x] for x in range(3)]) + '\n'",
...             step=100),
...     ],
...     gen=1000
... )
0.192, 0.302, 0.505
0.241, 0.292, 0.467
0.328, 0.273, 0.399
0.270, 0.322, 0.408
0.312, 0.412, 0.276
0.330, 0.344, 0.327
0.332, 0.424, 0.244
0.426, 0.372, 0.201
0.413, 0.384, 0.203
0.395, 0.408, 0.198
1000L
```

Note: Alleles other than 0, 1, ..., $n-1$ will not be mutated because their mutation rates are undefined. A warning message will be displayed for this case when debugging code `DBG_WARNING` is turnned on.

4.6.2 k-allele mutation model (`KAlleleMutator`)

A k-allele model assumes k alleles $(0, 1, ..., k-1)$ at a locus and mutate between them using rate matrix

$$p_{ij} = \begin{pmatrix} 1-\mu & \frac{\mu}{k-1} & \cdots & \frac{\mu}{k-1} \\ \frac{\mu}{k-1} & 1-\mu & \cdots & \frac{\mu}{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\mu}{k-1} & \frac{\mu}{k-1} & \cdots & 1-\mu \end{pmatrix}$$

The only parameter μ is the mutation rate, which is the rate at which an allele mutates to any other allele with equal probability.

This mutation model is a special case of the `MatrixMutator` but a specialized `KAlleleMutator` is recommended because it provides better performance, especially when k is large. In addition, this operator allows different mutation rates at different loci. When k is not specified, it is assumed to be the number of allowed alleles (e.g. 2 for binary modules). Example 4.32 desmonstrates the use of this operator where parameters `rate` and `loci` are used to specify different mutation rates for different loci. Because this operator treats all alleles equally, all alleles will have the same allele frequency in the long run.

Listing 4.32: A k-allele mutation model

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[2000], loci=1*3)
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(),
...     postOps=[
```

```
...         sim.KAlleleMutator(k=5, rates=[1e-2, 1e-3], loci=[0, 1]),
...         sim.Stat(alleleFreq=range(3), step=100),
...         sim.PyEval(r"', '.join(['%.3f' % alleleFreq[x][0] for x in range(3)]) + '\n'",
...             step=100),
...     ],
...     gen=500
... )
0.991, 0.999, 1.000
0.368, 0.918, 1.000
0.300, 0.815, 1.000
0.257, 0.639, 1.000
0.209, 0.573, 1.000
500L
```

Note: If alleles k and higher exist in the population, they will not be mutated because their mutation rates are undefined. A warning message will be displayed for this case when debugging code `DBG_WARNING` is turnned on.

4.6.3 Diallelic mutation models (`SNPMutator`)

`MatrixMutator` and `KAlleleMutator` are general purpose mutators in the sense that they do not assume a type for the mutated alleles. This and the following sections describe mutation models for specific types of alleles.

If there are only two alleles at a locus, a diallelic mutation model should be used. Because single nucleotide polymorphisms (SNPs) are the most widely avaiable diallelic markers, a `SNPMutator` is provided to mutate such markers using a mutate rate matrix

$$R = \begin{pmatrix} 1-u & u \\ v & 1-v \end{pmatrix}.$$

Despite of its name, this mutator can be used in many theoretical models assuming $\Pr(A \to a) = u$ and $\Pr(a \to A) = v$. If $v = 0$, mutations will be directional. Example 4.33 applies such a directional mutaton model to two loci, but with a purifying selection applied to the first locus. Because of the selection pressure, the frequency of allele 1 at the first locus does not increase indefinitely as allele 1 at the second locus.

Listing 4.33: A diallelic directional mutation model

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[2000], loci=[1, 1], lociNames=['A', 'B'],
...     infoFields='fitness')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=[
...         sim.SNPMutator(u=0.001),
...         sim.MaSelector(loci='A', fitness=[1, 0.99, 0.98]),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=['A', 'B'], step=100),
...         sim.PyEval(r"'%.3f\t%.3f\n' % (alleleFreq[0][1], alleleFreq[1][1])",
...             step=100),
...     ],
...     gen=500
... )
0.001   0.001
0.077   0.087
```

```
0.099   0.192
0.099   0.300
0.085   0.400
500L
```

4.6.4 Nucleotide mutation models (`AcgtMutator`)

Mutations in these models assume alleles 0, 1, 2, 3 as nucleotides A, C, G, and T. The operator is named `AcgtMutator` to remind you the alphabetic order of these nucleotides. This mutation model is specified by a rate matrix

$$\begin{array}{cccc} & A & C & G & T \\ A & - & x_1 & x_2 & x_3 \\ C & x_4 & - & x_5 & x_6 \\ G & x_7 & x_8 & - & x_9 \\ T & x_{10} & x_{11} & x_{12} & - \end{array}$$

which is determined by 12 parameters. However, several simpler models with fewer parameters can be used. In addition to parameters shared by all mutation operators, a nucleotide mutator is specified by a parameter list and a model name. For example:

```
AcgtMutator(rate=[1e-5, 0.5], model='K80')
```

specifies a nucleotide mutator using Kimura's 2-parameter model with $\mu = 10^{-5}$ and $\kappa = 0.5$. Because multiple parameters could be involved for a particular mutation model, **the definition of a mutation rate and other paramters are model dependent and may varying with different mathematical representation of the models**.

The names and acceptable parameters of acceptable models are listed below:

1. Jukes and Cantor 1969 model: `model='JC69'`, rate=[μ]

   The Jukes and Cantor model is similar to a 4-allele model but its definition of μ is different. More specifically, when a mutation event happens at rate μ, an allele will have equal probability to mutate to any of the 4 allelic states.

$$R = \begin{pmatrix} - & \frac{\mu}{4} & \frac{\mu}{4} & \frac{\mu}{4} \\ \frac{\mu}{4} & - & \frac{\mu}{4} & \frac{\mu}{4} \\ \frac{\mu}{4} & \frac{\mu}{4} & - & \frac{\mu}{4} \\ \frac{\mu}{4} & \frac{\mu}{4} & \frac{\mu}{4} & - \end{pmatrix}$$

2. Kimura's 2-parameter 1980 model: `model='K80'`, rate=[μ, κ]

   Kimura 's model distinguishes transitions ($A \longleftrightarrow G$, and $C \leftrightarrow T$ namely $0 \longleftrightarrow 2$ and $1 \longleftrightarrow 3$ with probability $\frac{\mu}{4}\kappa$) and transversions (others) with probability $\frac{\mu}{4}$. It would be a Jukes and Cantor model if $\kappa = 1$.

$$R = \begin{pmatrix} - & \frac{\mu}{4} & \frac{\mu}{4}\kappa & \frac{\mu}{4} \\ \frac{\mu}{4} & - & \frac{\mu}{4} & \frac{\mu}{4}\kappa \\ \frac{\mu}{4}\kappa & \frac{\mu}{4} & - & \frac{\mu}{4} \\ \frac{\mu}{4} & \frac{\mu}{4}\kappa & \frac{\mu}{4} & - \end{pmatrix}$$

3. Felsenstein 1981 model: `model='F81'`, rate=[μ, π_A, π_C, π_G].

   This model assumes different base frequencies but the same probabilities for transitions and transversions. π_T is calculated from π_A, π_C and π_G.

$$R = \begin{pmatrix} - & \mu\pi_C & \mu\pi_G & \mu\pi_T \\ \mu\pi_A & - & \mu\pi_G & \mu\pi_T \\ \mu\pi_A & \mu\pi_C & - & \mu\pi_T \\ \mu\pi_A & \mu\pi_C & \mu\pi_G & - \end{pmatrix}$$

4. Hasegawa, Kishino and Yano 1985 model: `model='HKY85'`, rate=[μ, κ, π_A, π_C, π_G]

   This model replaces 1/4 frequency used in the Kimura's 2-parameter model with nucleotide-specific frequencies.

$$R = \begin{pmatrix} - & \mu\pi_C & \mu\kappa\pi_G & \mu\pi_T \\ \mu\pi_A & - & \mu\pi_G & \mu\kappa\pi_T \\ \mu\kappa\pi_A & \mu\pi_C & - & \mu\pi_T \\ \mu\pi_A & \mu\kappa\pi_C & \mu\pi_G & - \end{pmatrix}$$

5. Tamura 1992 model: `model='T92'`, rate=[μ, π_{GC}]

   This model is a HKY85 model with $\pi_G = \pi_C = \pi_{GC}/2$ and $\pi_A = \pi_T = \pi_{AT}/2 = (1 - \pi_{GC})/2$,

$$R = \begin{pmatrix} - & \frac{1}{2}\mu\pi_{GC} & \frac{1}{2}\mu\nu\pi_{GC} & \frac{1}{2}\mu\pi_{AT} \\ \frac{1}{2}\mu\pi_{AT} & - & \frac{1}{2}\mu\pi_{GC} & \frac{1}{2}\mu\nu\pi_{AT} \\ \frac{1}{2}\mu\nu\pi_{AT} & \frac{1}{2}\mu\pi_{GC} & - & \frac{1}{2}\mu\pi_{AT} \\ \frac{1}{2}\mu\pi_{AT} & \frac{1}{2}\mu\nu\pi_{GC} & \frac{1}{2}\mu\pi_{GC} & - \end{pmatrix}$$

6. Tamura and Nei 1993 model: `model='TN93'`, rate=[μ, κ_1, κ_2, π_A, π_C, π_G]

   This model extends the HKY1985 model by distinguishing $A \longleftrightarrow G$ transitions (namely $0 \longleftrightarrow 2$) and $C \leftrightarrow T$ transitions ($1 \longleftrightarrow 3$) with different κ.

$$R = \begin{pmatrix} - & \mu\pi_C & \mu\kappa_1\pi_G & \mu\pi_T \\ \mu\pi_A & - & \mu\pi_G & \mu\kappa_2\pi_T \\ \mu\kappa_1\pi_A & \mu\pi_C & - & \mu\pi_T \\ \mu\pi_A & \mu\kappa_2\pi_C & \mu\pi_G & - \end{pmatrix}$$

7. Generalized time reversible model: `model='GTR'`, rate=[x_1, x_2, x_3, x_4, x_5, x_6, π_A, π_C, π_G]

   The generalized time reviersible model is the most general neutral, indepdendent, finite-sites, time-reversible model possible. It is specified by six parameters and base frequencies. Its rate matrix is defined as

$$R = \begin{pmatrix} - & \frac{\pi_A x_1}{\pi_C} & \frac{\pi_A x_2}{\pi_G} & \frac{\pi_A x_3}{\pi_T} \\ x_1 & - & \frac{\pi_C x_4}{\pi_G} & \frac{\pi_C x_5}{\pi_T} \\ x_2 & x_4 & - & \frac{\pi_G x_6}{\pi_T} \\ x_3 & x_5 & x_6 & - \end{pmatrix}$$

8. General model: `model='general'` (default), rate=[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}].

   This is the most general model with 12 parameters:

$$R = \begin{pmatrix} - & x_1 & x_2 & x_3 \\ x_4 & - & x_5 & x_6 \\ x_7 & x_8 & - & x_9 \\ x_{10} & x_{11} & x_{12} & - \end{pmatrix}$$

   It is not surprising that all other models are implemented as special cases of this model.

Example 4.34 applies a Kimmura's 2-parameter mutation model to a population with a single nucleotide marker.

Listing 4.34: A Kimura's 2 parameter mutation model

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[2000], loci=1,
...     alleleNames=['A', 'C', 'G', 'T'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
```

```
...         sim.InitGenotype(freq=[.1, .1, .1, .7])
...     ],
...     matingScheme=sim.RandomMating(),
...     preOps=[
...         sim.AcgtMutator(rate=[1e-4, 0.5], model='K80'),
...         sim.Stat(alleleFreq=0, step=100),
...         sim.PyEval(r"', '.join(['%.3f' % alleleFreq[0][x] for x in range(4)]) + '\n'",
...             step=100),
...     ],
...     gen=500
... )
0.093, 0.101, 0.094, 0.712
0.142, 0.073, 0.084, 0.701
0.135, 0.160, 0.083, 0.623
0.230, 0.128, 0.013, 0.628
0.293, 0.189, 0.008, 0.510
500L
```

4.6.5 Mutation model for microsatellite markers (`StepwiseMutator`)

The **stepwise mutation model** (SMM) was proposed by Ohta and Kimura [1973] to model the mutation of Variable Number Tandem Repeat (VNTR), which consists of tandem repeat of sequences. VNTR markers consisting of short sequences (e.g. 5 basepair or less) are also called microsatellite markers. A mutation event of a VNTR marker either increase of decrease the number of repeats, as a result of slipped-strand mispairing or unequal sister chromatid exchange and genetic recombination.

A `StepwiseMutator` assumes that alleles at a locus are the number of tandem repeats and mutates them by increasing or decreasing the number of repeats during a mutation event. By adjusting parameters `incProb`, `maxAllele` and `mutStep`, this operator can be used to simulate the standard neutral stepwise mutation model and a number of **generalized stepwise mutation models**. For example, Example 4.35 uses two `StepwiseMutator` to mutate two microsatellite markers, using a standard and a generalized model where a geometric distribution is used to determine the number of steps.

Listing 4.35: A standard and a generalized stepwise mutation model

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=1000, loci=[1, 1])
>>> pop.evolve(
...     # all start from allele 50
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq= [0]*50 + [1])
...     ],
...     matingScheme=sim.RandomMating(),
...     preOps=[
...         sim.StepwiseMutator(rates=1e-3, loci=0),
...         sim.StepwiseMutator(rates=1e-3, incProb=0.6, loci=1,
...             mutStep=(sim.GEOMETRIC_DISTRIBUTION, 0.2)),
...     ],
...     gen=100
... )
100L
>>> # count the average number tandem repeats at both loci
>>> cnt0 = cnt1 = 0
>>> for ind in pop.individuals():
```

```
...     cnt0 += ind.allele(0, 0) + ind.allele(0, 1)
...     cnt1 += ind.allele(1, 0) + ind.allele(1, 1)
...
>>> print('Average number of repeats at two loci are %.2f and %.2f.' % \
...     (cnt0/2000., cnt1/2000.))
Average number of repeats at two loci are 50.03 and 49.70.
```

4.6.6 Simulating arbitrary mutation models using a hybrid mutator (`PyMutator`)*

A hybrid mutator `PyMutator` mutates random alleles at selected loci (parameter `loci`), replicates (parameter `loci`), subpopulations (parameter `subPop`) with specified mutation rate (parameter `rate`). Instead of mutating the alleles by itself, it passes the alleles to a user-defined function and use it return values as the mutated alleles. Arbitrary mutation models could be implemented using this operator.

Example 4.36 applies a simple mutation model where an allele is increased by a random number between 1 and 5 when it is mutated. Two different mutation rates are used for two different loci so average alleles at these two loci are different.

Listing 4.36: A hybrid mutation model

```
>>> import simuPOP as sim
>>> import random
>>> def incAllele(allele):
...     return allele + random.randint(1, 5)
...
>>> pop = sim.Population(size=1000, loci=[20])
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(),
...     postOps=sim.PyMutator(func=incAllele, rates=[1e-4, 1e-3],
...             loci=[2, 10]),
...     gen = 1000
... )
1000L
>>> # count the average number tandem repeats at both loci
>>> def avgAllele(pop, loc):
...     ret = 0
...     for ind in pop.individuals():
...         ret += ind.allele(loc, 0) + ind.allele(loc, 1)
...     return ret / (pop.popSize() * 2.)
...
>>> print('Average number of repeats at two loci are %.2f and %.2f.' % \
...     (avgAllele(pop, 2), avgAllele(pop, 10)))
Average number of repeats at two loci are 0.01 and 3.13.
```

4.6.7 Mixed mutation models (`MixedMutator`) **

Mixed mutation models are sometimes used to model real data. For example, a k-allele model can be used to explain extremely large or small number of tandem repeats at a microsatellite marker which are hard to justify using a standard stepwise mutation model. A mixed mutation model would apply two or more mutation models at pre-specified probabilities.

A `MixedMutator` is constructed by a list of mutators and their respective probabilities. It accepts regular mutator parameters such as `rates`, `loci`, `subPops`, `mapIn and mapOut` and mutates aleles at specified rate. When a mutation event

happens, it calls one of the mutators to mutate the allele. For example, Example 4.37 applies a mixture of k-allele model and stepwise model to mutate a micosatellite model.

Listing 4.37: A mixed k-allele and stepwise mutation model

```
>>> import simuPOP as sim
>>> pop = sim.Population(5000, loci=[1, 1])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[50, 50])
...     ],
...     preOps=[
...         # the first locus uses a pure stepwise mutation model
...         sim.StepwiseMutator(rates=0.001, loci=0),
...         # the second locus uses a mixed model
...         sim.MixedMutator(rates=0.001, loci=1, mutators=[
...             sim.KAlleleMutator(rates=1, k=100),
...             sim.StepwiseMutator(rates=1)
...         ], prob=[0.1, 0.9])],
...     matingScheme=sim.RandomMating(),
...     gen = 20
... )
20L
>>> # what alleles are there?
>>> geno0 = []
>>> geno1 = []
>>> for ind in pop.individuals():
...     geno0.extend([ind.allele(0, 0), ind.allele(0, 1)])
...     geno1.extend([ind.allele(1, 0), ind.allele(1, 1)])
...
>>> print('Locus 0 has alleles', ', '.join([str(x) for x in set(geno0)]))
('Locus 0 has alleles', '49, 50, 51')
>>> print('Locus 1 has alleles', ', '.join([str(x) for x in set(geno1)]))
('Locus 1 has alleles', '88, 49, 50, 51, 67')
```

When a mutation event happens, mutators in Example 4.37 mutate the allele with probability (mutation rate) 1. If different mutation rates are specified, the overall mutation rates would be the product of mutation rate of `MixedMutator` and the passed mutators. However, it is extremely important to understand that although `MixedMutator(rates=mu)` with `StepwiseMutator(rates=1)` and `MixedMutator(rates=1)` with `StepwiseMutator(rates=mu)` mutate alleles at the same mutation rate, the former is much more efficient because it triggers far less mutation events.

4.6.8 Context-dependent mutation models (`ContextMutator`)**

All mutation models we have seen till now are context independent. That is to say, how an allele is mutated depends only on the allele itself. However, it is understood that DNA and amino acid substitution rates are highly sequence context-dependent, e.g., C $\rightarrow$ T substitutions in vertebrates may occur much more frequently at CpG sites. To simulate such models, a mutator must consider the context of a mutated allele, e.g. certain number of alleles to the left and right of this allele, and mutate the allele accordingly.

A `ContextMutator` can be used to mutate an allele depending on its surrounding loci. This mutator is constructed by a list of mutators and their respective contexts. It accepts regular mutator parameters such as `rates`, `loci`, `subPops`, `mapIn and mapOut` and mutates aleles at specified rate. When a mutation event happens, it checks the context of the mutaed allele and choose a corresponding mutator to mutate the allele. An additional mutator can be specified to mutate alleles with unknown context. Example 4.38 applies two `SNPMutator` at different rates under different contexts.

Listing 4.38: A context-dependent mutation model

```
>>> import simuPOP as sim
>>> pop = sim.Population(5000, loci=[3, 3])
>>> pop.evolve(
...     # initialize locus by 0, 0, 0, 1, 0, 1
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[1, 1], loci=[3, 5])
...     ],
...     preOps=[
...         sim.ContextMutator(mutators=[
...             sim.SNPMutator(u=0.1),
...             sim.SNPMutator(u=1),
...             ],
...             contexts=[(0, 0), (1, 1)],
...             loci=[1, 4],
...             rates=0.01
...         ),
...         sim.Stat(alleleFreq=[1, 4], step=5),
...         sim.PyEval(r"'Gen: %2d freq1: %.3f, freq2: %.3f\n'" +
...             " % (gen, alleleFreq[1][1], alleleFreq[4][1])", step=5)
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 20
... )
Gen:  0 freq1: 0.001, freq2: 0.010
Gen:  5 freq1: 0.005, freq2: 0.059
Gen: 10 freq1: 0.007, freq2: 0.108
Gen: 15 freq1: 0.015, freq2: 0.142
20L
```

Note that although

```
ContextMutator(mutators=[
    SNPMutator(u=0.1),
    SNPMutator(u=1)],
    contexts=[(0, 0), (1, 1)],
    rates=0.01
)
```

and

```
ContextMutator(mutators=[
    SNPMutator(u=0.001),
    SNPMutator(u=0.01)],
    contexts=[(0, 0), (1, 1)],
    rates=1
)
```

both apply two `SNPMutator` at mutation rates `0.001` and `0.01`, the former is more efficient because it triggers less mutation events.

Context-dependent mutator can also be implemented by a `PyMutator`. When a non-zero parameter `context` is specified, this mutator will collect `context` number of alleles to the left and right of a mutated allele and pass them as a second parameter of the user-provided mutation function. Example 4.39 applies the same mutation model as Example 4.38 using a `PyMutator`.

Listing 4.39: A hybrid context-dependent mutation model

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(5000, loci=[3, 3])
>>> def contextMut(allele, context):
...     if context == [0, 0]:
...         if allele == 0 and random.random() < 0.1:
...             return 1
...     elif context == [1, 1]:
...         if allele == 0:
...             return 1
...     # do not mutate
...     return allele
...
>>> pop.evolve(
...     # initialize locus by 0, 0, 0, 1, 0, 1
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[1, 1], loci=[3, 5])
...     ],
...     preOps=[
...         sim.PyMutator(func=contextMut, context=1,
...             loci=[1, 4],  rates=0.01
...         ),
...         #sim.SNPMutator(u=0.01, v= 0.01, loci=[1, 4]),
...         sim.Stat(alleleFreq=[1, 4], step=5),
...         sim.PyEval(r"'Gen: %2d freq1: %.3f, freq2: %.3f\n'" +
...             " % (gen, alleleFreq[1][1], alleleFreq[4][1])", step=5)
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 20
... )
Gen:  0 freq1: 0.000, freq2: 0.000
Gen:  5 freq1: 0.000, freq2: 0.000
Gen: 10 freq1: 0.000, freq2: 0.000
Gen: 15 freq1: 0.000, freq2: 0.000
20L
```

4.6.9 Manually-introduced mutations (`PointMutator`)

Operator `PointMutator` is different from all other mutators in that it mutates specified alleles of specified individuals. It is usually used to manually introduce one or more mutants to a population. Although it is not a recommended method to introduce a disease predisposing allele, the following example (Example 4.40) demonstrates an evolutionary process where mutants are repeatedly introduced and raised by positive selection until it reaches an appreciable allele frequency. This example uses two `IfElse` operators. The first one introduces a mutant when there is no mutant in the population, and the second one terminate the evolution when the frequency of the mutant reaches 0.05.

Listing 4.40: Use a point mutator to introduce a disease predisposing allele

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=1, infoFields='fitness')
>>> pop.evolve(
...     initOps=sim.PyOutput('Introducing alleles at generation'),
...     preOps=sim.MaSelector(loci=0, wildtype=0, fitness=[1, 1.05, 1.1]),
```

```
...     matingScheme=sim.RandomSelection(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.IfElse('alleleNum[0][1] == 0', ifOps=[
...             sim.PyEval(r"' %d' % gen"),
...             sim.PointMutator(inds=0, loci=0, allele=1),
...         ]),
...         sim.IfElse('alleleFreq[0][1] > 0.05', ifOps=[
...             sim.PyEval(r"'.\nTerminate at generation %d at allele freq %.3f.\n'" +
...                 " % (gen, alleleFreq[0][1])"),
...             sim.TerminateIf('True'),
...         ])
...     ],
... )
Introducing alleles at generation 0 1 2 16 17 18 22 30 32 33 34 41 81 82 83.
Terminate at generation 111 at allele freq 0.051.
112L
```

4.6.10 Apply mutation to (virtual) subpopulations *

A mutator is usually applied to all individuals in a population. However, you can restrict its use to specified subpopulations and/or virtual subpopulations using parameter `subPop`. For example, you can use `subPop=[0, 2]` to apply the mutator only to individuals in subpopulations 0 and 2.

Virtual subpopulations can also be specified in this parameter. For example, you can apply different mutation models to male and female individuals, to unaffected or affected individuals, to patients at different stages of a cancer. Example 4.41 demonstrate a mutation model where individuals with more tandem repeats at a disease predisposing locus are more likely to develop a disease (e.g. fragile-X). Affected individuals are then subject to a non-neutral mutation model at an accerlerated mutation rate.

Listing 4.41: Applying mutation to virtual subpopulations.

```
>>> import simuPOP as sim
>>> def fragileX(geno):
...     '''A disease model where an individual has increased risk of
...     affected if the number of tandem repeats exceed 75.
...     '''
...     # Alleles A1, A2.
...     maxRep = max(geno)
...     if maxRep < 50:
...         return 0
...     else:
...         # individuals with allele >= 70 will surely be affected
...         return min(1, (maxRep - 50)*0.05)
...
>>> def avgAllele(pop):
...     'Get average allele by affection sim.status.'
...     sim.stat(pop, alleleFreq=(0,1), subPops=[(0,0), (0,1)],
...         numOfAffected=True, vars=['alleleNum', 'alleleNum_sp'])
...     avg = []
...     for alleleNum in [\
...             pop.dvars((0,0)).alleleNum[0],  # first locus, unaffected
...             pop.dvars((0,1)).alleleNum[0],  # first locus, affected
...             pop.dvars().alleleNum[1],       # second locus, overall
```

```
...         ]:
...         alleleSum = numAllele = 0
...         for idx,cnt in enumerate(alleleNum):
...             alleleSum += idx * cnt
...             numAllele += cnt
...         if numAllele == 0:
...             avg.append(0)
...         else:
...             avg.append(alleleSum * 1.0 /numAllele)
...     # unaffected, affected, loc2
...     pop.dvars().avgAllele = avg
...     return True
...
>>> pop = sim.Population(10000, loci=[1, 1])
>>> pop.setVirtualSplitter(sim.AffectionSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[50, 50])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         # determine affection sim.status for each offspring (duringMating)
...         sim.PyPenetrance(func=fragileX, loci=0),
...         # unaffected offspring, mutation rate is high to save some time
...         sim.StepwiseMutator(rates=1e-3, loci=1),
...         # unaffected offspring, mutation rate is high to save some time
...         sim.StepwiseMutator(rates=1e-3, loci=0, subPops=[(0, 0)]),
...         # affected offspring have high probability of mutating upward
...         sim.StepwiseMutator(rates=1e-2, loci=0, subPops=[(0, 1)],
...            incProb=0.7, mutStep=3),
...         # number of affected
...         sim.PyOperator(func=avgAllele, step=20),
...         sim.PyEval(r"'Gen: %3d #Aff: %d AvgRepeat: %.2f (unaff), %.2f (aff), %.2f (unrelated)\n'"
...             + " % (gen, numOfAffected, avgAllele[0], avgAllele[1], avgAllele[2])",
...             step=20),
...     ],
...     gen = 101
... )
Gen:   0 #Aff: 0 AvgRepeat: 1.01 (unaff), 0.00 (aff), 1.01 (unrelated)
Gen:  20 #Aff: 6 AvgRepeat: 1.53 (unaff), 0.50 (aff), 1.52 (unrelated)
Gen:  40 #Aff: 20 AvgRepeat: 2.56 (unaff), 2.04 (aff), 1.53 (unrelated)
Gen:  60 #Aff: 46 AvgRepeat: 2.56 (unaff), 2.04 (aff), 2.04 (unrelated)
Gen:  80 #Aff: 55 AvgRepeat: 3.08 (unaff), 1.53 (aff), 2.04 (unrelated)
Gen: 100 #Aff: 48 AvgRepeat: 2.04 (unaff), 1.52 (aff), 2.04 (unrelated)
101L
```

At the beginning of a simulation, all individuals have 50 copies of a tandem repeat and the mutation follows a standard neutral stepwise mutation model. individuals with more than 50 repeats will have an increasing probability to develop a disease ($\Pr(\text{affected} \mid n) = (n - 50) * 0.05$) for $50 \leq n \leq 70$). The averge repeat number therefore increases for affected individuals. In contrast, the mean number of repeats at locus 1 on a separate chromosome oscillate around 50.

4.6.11 Allele mapping **

If alleles in your simulation do not follow the convention of a mutation model, you may want to use the `pop.recodeAlleles()` function to recode your alleles so that appropriate mutation models could be applied. If this is not possible, you can use a general mutation model with your own mutation matrix, or an advanced feature called **allele mapping**.

Allele mapping is done through two parameters *mapIn* and *mapOut*, which map alleles in your population to and from alleles assumed in a mutation model. For example, an `AcgtMutator` mutator assumes alleles `A`, `C`, `G` and `T` for alleles 0, 1, 2, and 3 respectively. If for any reason the alleles in your application does not follow this order, you will need to map these alleles to the alleles assumed in the mutator. For example, if you assumes `C`, `G`, `A`, `T` for alleles 0, 1, 2, and 3 respectively, you can use parameters

```
mapIn=[1, 2, 0, 3], mapOut=[2, 0, 1, 3]
```

to map your alleles (`C(0)->C(1)`, `G(1)->G(2)`, `A(2)->A(0)`, `T(3)->T(3)`) to alleles `AcgtMutator` assumes, and then map mutated alleles (`A(0)->A(2)`, `C(1)->C(0)`, `G(2)->G(1)`, `T(3)->T(3)`) back. Example 4.42 gives another example where alleles 4, 5, 6 and 7 are mutated using a 4-allele model.

Listing 4.42: Allele mapping for mutation operators

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[2000], loci=1)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0]*4 + [0.1, 0.2, 0.3, 0.4])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.KAlleleMutator(k=4, rates=1e-4, mapIn=[0]*4 + list(range(4)),
...             mapOut=[4, 5, 6, 7]),
...         sim.Stat(alleleFreq=0, step=100),
...         sim.PyEval(r"', '.join(['%.2f' % alleleFreq[0][x] for x in range(8)]) + '\n'",
...             step=100),
...     ],
...     gen=500
... )
0.00, 0.00, 0.00, 0.00, 0.09, 0.20, 0.30, 0.41
0.00, 0.00, 0.00, 0.00, 0.13, 0.20, 0.40, 0.26
0.00, 0.00, 0.00, 0.00, 0.17, 0.20, 0.31, 0.31
0.00, 0.00, 0.00, 0.00, 0.19, 0.18, 0.26, 0.37
0.00, 0.00, 0.00, 0.00, 0.18, 0.24, 0.23, 0.34
500L
```

These two parameters also accept Python functions which should return corresponding mapped-in or out allele for a given allele. These two functions can be used to explore very fancy mutation models. For example, you can categorize a large number of alleles into alleles assumed in a mutation model, and emit random alleles from a mutated allele.

4.6.12 Mutation rate and transition matrix of a `MatrixMutator` **

A `MatrixMutator` is specified by a mutation rate matrix. Although mutation rates of this mutator is typically allele-dependent, the `MatrixMutator` is implemented as a two-step process where mutation events are triggered independent to allelic states. This section describes these two steps which can be useful if you need to use a `maxtrixMutator` in a `MixedMutator` or `ContextMutator`, and would like to factor out an allele-independent mutation rate to the wrapper mutator.

Because alleles usually have different probabilities of mutating to other alleles, **a mutation process is usually allele dependent**. Given a mutation model (p_{ij}), it is obviously inefficient to go through all mutable alleles and determine whether or not to mutate it using $p_{ij}, j = 0, ..., 1 - n$. simuPOP uses a two step procedure to mutate a large number of alleles. More specifically, for each mutation model, we determine $\mu = \max_{i=0}^{n-1}(1 - p_{ii})$ as the overall mutation rate, and then

1. For each allele, trigger a mutation event with probability μ. Because μ is usually very small and is the same for all alleles, this step can be implemented efficiently.
2. When a mutation event happens, mutation allele i to allele j with probability

$$\Pr(i \to j) = \begin{cases} 1 - \frac{1}{\mu}(1 - p_{ii}) & \text{if } i = j \\ \frac{p_{ij}}{\mu} & \text{if } i \neq j \end{cases}$$

Because steps 1 and 2 are independent, it is easy to verify that

$$p_{ij} = \mu \Pr(i \to j)$$

if $i \neq j$ and

$$p_{ii} = (1 - \mu) + \mu \Pr(i \to i)$$

where the first and second items are probabilities of no-mutation at steps 1 and 2. μ was chosen as the smallest μ that makes $0 \leq \Pr(i \to i) \leq 1$ for all i.

For example, for a k-allele model with

$$p_{ij} = \begin{pmatrix} 1-\mu & \frac{\mu}{k-1} & \cdots & \frac{\mu}{k-1} \\ \frac{\mu}{k-1} & 1-\mu & \cdots & \frac{\mu}{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\mu}{k-1} & \frac{\mu}{k-1} & \cdots & 1-\mu \end{pmatrix}$$

μ is directly μ for the first step and

$$\Pr(i \to j) = \begin{cases} 0 & \text{if } i = j \\ \frac{1}{k-1} & \text{if } i \neq j \end{cases}$$

for the second step. Therefore, mutation rate μ in a k-allele model could be interpreted as the probability of mutation, and a mutation event would mutate an allele to any other allele with equal probability.

For a classical mutation model with $P(A \to a) = u$ and $P(a \to A) = v$,

$$p_{ij} = \begin{pmatrix} 1-u & u \\ v & 1-v \end{pmatrix}$$

if $u = 0.001$ and $v = 0.0005$, $\mu = \max(u, v) = 0.001$,

$$\Pr(i \to j) = \begin{pmatrix} 0 & 1 \\ \frac{v}{u} = 0.5 & 1 - \frac{v}{u} = 0.5 \end{pmatrix}$$

That is to say, we would mutate at a mutation rate $u = 0.001$, mutate allele A to a with probability 1 and mutate allele a to A with probability 0.5.

4.6.13 Infinite-sites model and other simulation techniques **

Infinite-sites and infinite-alleles models have some similarities. If you assume that mutation is the only force to create new mutants, you can treat a long chromosomal region as a locus and use the infinite-alleles model, actually a k-allele model with large k, to mimic the infinite-site model. This assumption is certainly wrong with the infinite-site model

when recombination is involved, because recombination creates new haplotypes (alleles) under the infinite-site model. However, for short regions where recombination can be ignored, an k-allele model can be an easy and fast way to mimic an infinite-site model. That statement basically says that you have a choice between two models if you would like to simulate the evolution of this gene, namely considering the gene as a locus and simulating variants as alleles, or considering the gene as a sequence and simulating haplotypes as alleles.

For example, the CFTR gene (for cystic fibrosis) can have many alleles (thinking in terms of infinite-allele model) which are nucleotide mutations on tens of locations (infinite-site model). In order to simulate the evolution of this gene, you have a choice between two models, namely considering the gene as a locus and simulating variants as alleles, or considering the gene as a sequence and simulating haplotypes as alleles. Because there is supposed to be only one mutant at each site, you can assign a unique *location* for each allele of an infinite-allele model and convert multi-allelic datasets simulated by an infinite-allele model to sequences of diallelic markers. Note that mutation rates are interpreted differently for these two models.

If specific location of such a mutation is needed, it is possible to record the location of mutations during an evolution and minic an infinite-sites model. For example, alleles in Example 4.43 are used to store location of a mutation event. When a mutation event happens, the location of the new allele (rather the allele itself) is recorded on the chromosome (actually list of mutation events) of an individual. The transmission of chromosomes proceed normally and effectively transmit mutants from parents to offspring. At the end of the simulation, each individual accumulates a number of mutation events and they are essentially alleles at their respective locations.

Listing 4.43: Mimicking an infinite-sites model using mutation events as alleles

```
>>> import simuOpt
>>> simuOpt.setOptions(alleleType='long')
>>> import simuPOP as sim
>>>
>>> def infSitesMutate(pop, param):
...     '''Apply an infinite mutation model'''
...     (startPos, endPos, rate) = param
...     # for each individual
...     for ind in pop.individuals():
...         # for each homologous copy of chromosomes
...         for p in range(2):
...             # using a geometric distribution to determine
...             # the first mutation location
...             loc = sim.getRNG().randGeometric(rate)
...             # if a mutation happens, record the mutated location
...             if startPos + loc < endPos:
...                 try:
...                     # find the first non-zero location
...                     idx = ind.genotype(p).index(0)
...                     # record mutation here
...                     ind.setAllele(startPos + loc, idx, ploidy=p)
...                 except:
...                     raise
...                     print('Warning: more than %d mutations have accumulated' % pop.totNumLoci())
...                     pass
...     return True
...
>>> pop = sim.Population(size=[2000], loci=[100])
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=[
...         # mutate in a 10Mb region at rate 1e-8
...         sim.PyOperator(func=infSitesMutate, param=(1, 10000000, 1e-8)),
```

```
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 100
... )
100L
>>> # now, we get a sim.Population. Let us have a look at the 'alleles'.
>>> # print the first five mutation locations
>>> print(pop.individual(0).genotype()[:5])
[1527502, 4774892, 7979220, 3671118, 395142]
>>> # how many alleles are there (does not count 0)?
>>> print(len(set(pop.genotype())) - 1)
2700
>>> # Allele count a simple count of alleles.
>>> cnt = {}
>>> for allele in pop.genotype():
...     if allele == 0:
...         continue
...     if allele in cnt:
...         cnt[allele] += 1
...     else:
...         cnt[allele] = 1
...
>>> # highest allele frequency?
>>> print(max(cnt.values()) *0.5 / pop.popSize())
0.05475
```

All mutation models in simuPOP apply to existing alleles at pre-specified loci. However, if the location of loci cannot be determined beforehand, it is sometimes desired to create new loci as a result of mutation. A customized operator can be used for this purpose (see Example 4.101), but extra attention is needed to make sure that other operators are applied to the correct loci because loci indexes will be changed with the insertion of new loci. This technique could also be used to simulate mutations over long sequences.

4.6.14 Recording and tracing individual mutants **

Mutation operators mutate alleles in place and by default do not generate any output. If you are interested in knowing the source of each mutant, you can specify an output stream and let the mutation operators dump details of each mutation event, which consists of generation number, locus index, ploidy, original allele, and mutated allele. If a list of information fields are specified through parameter `infoFields`, values at these information fields will also be outputted (if they exist in the population. The default information field is `ind_id`, which allow you to record the ID of individuals harboring the mutants.

Example 4.44 demonstrates how to use this feature to count the number of mutants at each locus. Instead of sending the output to a file (e.g. `output='>>mutants.txt'`), this example sends the output to a Python function, which parses input string and counts the number of mutants at each locus using a global dictionary variable. As we can see from the output, because the KAlleleMutator uses a higher mutation rate (0.01) at locus 1 than mutation rate (0.001) at locus 0, there are 10 times more mutants at the second locus. There are about 3/4 mutations on the locus on chromosome X and 1/4 mutations on the locus on chromosome Y, for obvious reasons.

Listing 4.44: Count number of mutants from mutator outputs

```
>>> import simuPOP as sim
>>> from collections import defaultdict
>>> # count number of mutants at each locus
>>> counter = defaultdict(int)
>>> def countMutants(mutants):
```

```
...     global counter
...     for line in mutants.split('\n'):
...         # a trailing \n will lead to an empty string
...         if not line:
...             continue
...         (gen, loc, ploidy, a1, a2, id) = line.split('\t')
...         counter[int(loc)] += 1
...
>>> pop = sim.Population([5000]*3, loci=[2,1,1], infoFields='ind_id',
...     chromTypes=[sim.AUTOSOME, sim.CHROMOSOME_X, sim.CHROMOSOME_Y])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         sim.IdTagger(),
...     ],
...     preOps=[
...         sim.KAlleleMutator(rates=[0.001] + [0.01]*3,
...             loci=range(4), k=100, output=countMutants),
...     ],
...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.IdTagger(),
...             sim.MendelianGenoTransmitter()
...         ]),
...     gen = 10
... )
10L
>>> print(counter.items())
[(0, 308), (1, 2984), (2, 2319), (3, 768)]
```

4.7 Penetrance

Penetrance is the probability for an individual to be affected with a disease conditioning on his or her genotype and other risk factors. A penetrance model calculates such a probability for an individual and assign affection status randomly according to this probability. For example, if an individual with genotype `10` has probability 0.2 to be affected according to a penetrance model, he or she will be affected with probability 0.2. Note that simuPOP supports only one affection status. If there are multiple affection outcomes involved, you can treat them as binary quantitative traits and use information fields to store them.

A penetrance operator can be applied before or after mating, to assign affection status to all individuals in the parental or offspring generation, respectively. It can also be applied during mating and assign affection status to each offspring. The latter could be used to assit natural selection through the selection of offspring. You can also assign affection status to all individuals in a population using the function form of a penetrance operator (e.g. function `mapPenetrance` for operator `MapPenetrance`). Compared the penetrance operators that assign affection status to only the current generation, **these functions by default assign affection status to all ancestral generations as well**.

A penetrance operator usually do not store the penetrance values. However, if an information field is given, penetrance values will be saved to this information field before it is used to determine individual affection status.

4.7.1 Map penetrance model (operator `MapPenetrance`)

A map penetrance opertor uses a Python dictionary to provide penetrance values for each type of genotype. For example, Example 4.45 uses a dictionary with keys `(0,0)`, `(0,1)` and `(1,1)` to specify penetrance for individuals with these genotypes at locus 0.

Listing 4.45: A penetrance model that uses pre-defined fitness value

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=2000, loci=2)
>>> sim.initGenotype(pop, freq=[.2, .8])
>>> sim.mapPenetrance(pop, loci=0,
...     penetrance={(0,0):0, (0,1):.2, (1,1):.3})
>>> sim.stat(pop, genoFreq=0, numOfAffected=1, vars='genoNum')
>>> # number of affected individuals
>>> pop.dvars().numOfAffected
531
>>> # which should be roughly (#01 + #10) * 0.2 + #11 * 0.3
>>> (pop.dvars().genoNum[0][(0,1)] + pop.dvars().genoNum[0][(1,0)]) * 0.2 \
... + pop.dvars().genoNum[0][(1,1)] * 0.3
514.2
```

The above example assumes that penetrance for individuals with genotypes `(0,1)` and `(1,0)` are the same. This assumption is usually valid but can be vialoated with impriting. In that case, you can specify fitness for both types of genotypes. The underlying mechanism is that the `MapPenetrance` looks up a genotype in the dictionary first directly, and then without phase information if a genotype is not found.

This operator supports haplodiploid populations and sex chromosomes. In these cases, only valid alleles should be listed which can lead to dictionary keys with different lengths. In addition, although less used because of potentially a large number of keys, this operator can act on multiple loci. For example,

- keys `(a1,a2)` and `(a1,)` can be used to specify fitness values for female and male individuals in a haplodiploid population, respectively
- keys `(x1,x2)` and `(x1,)` can be used to specify fitness for female and male individuals according to a locus on the X chromosome in a diploid population, respectively. Similarly, keys `()` and `(y,)` for a locus on chromosome Y.
- keys `(a1,a2,b1,b2)` can be used to specify fitness values according to genotype at two loci in a diploid population.

4.7.2 Multi-allele penetrance model (operator `MaPenetrance`)

A multi-allele penetrance model divides alleles into two groups, wildtype A and mutants a, and treat alleles within each group as the same. The penetrance model is therefore simplified to

- Two fitness values for genotype A, a in the haploid case
- Three fitness values for genotype AA, Aa and aa in the diploid single locus case. Genotype Aa and aA are assumed to have the same impact on fitness.

The default wildtype group contains allele 0 so the two allele groups are zero and non-zero alleles. Example 4.46 demonstrates the use of this operator.

Listing 4.46: A multi-allele penetrance model

```
>>> import simuPOP as sim
>>> pop = sim.Population(5000, loci=3)
>>> pop.evolve(
```

```
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.9] + [0.02]*5)
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.MaPenetrance(loci=0, penetrance=(0.01, 0.2, 0.3)),
...         sim.Stat(numOfAffected=True, vars='propOfAffected'),
...         sim.PyEval(r"'Gen: %d Prevalence: %.1f%%\n' % (gen, propOfAffected*100)"),
...     ],
...     gen = 5
... )
Gen: 0 Prevalence: 4.4%
Gen: 1 Prevalence: 4.4%
Gen: 2 Prevalence: 4.7%
Gen: 3 Prevalence: 4.4%
Gen: 4 Prevalence: 4.3%
5L
```

Operator `MaPenetrance` also supports multiple loci by specifying fitness values for all combination of genotype at specified loci. In the case of two loci, this operator requires

- Four fitness values for genotype `AB`, `Ab`, `aB` and `ab` in the haploid case,
- Nine fitness values for genotype `AABB`, `AABb`, `AAbb`, `AaBB`, `AaBb`, `Aabb`, `aaBB`, `aaBb`, and `aabb` in the haploid case.

In general, 2^n values are needed for haploid populations and 3^n values are needed for diploid populations where n is the number of loci. This operator does not yet support haplodiploid populations and sex chromosomes.

4.7.3 Multi-loci penetrance model (operator `MlPenetrance`)

Although an individual's affection status can be affected by several factors, each of which can be modeled individually, **only one penetrance value is used to determine a person's affection status** and we have to use a multi-locus penetrance model to combine single-locus models.

This multi-loci penetrance model applies several penetrance models to each Individual and computes an overall penetrance value from the penetrance values provided by these operators. Although this selector is designed to obtain multi-loci penetrance values from several single-locus penetrance models, any penetrance operator, including those obtain their penetrance values from multiple disease predisposing loci, can be used in this operator. This operator uses parameter `mode` to control how Individual penetrance values are combined. More specifically, if f_i are penetrance values obtained from individual selectors, this selector returns

- $\Pi_i f_i$ if `mode=MULTIPLICATIVE`, and
- $\sum_i f_i$ if `mode=ADDITIVE`, and
- $1 - \Pi_i (1 - f_i)$ if `mode=HETEROGENEITY`

0 or 1 will be returned if the returned fitness value is out of range of `[0,1]`.

Example 4.47 demonstrates the use of this operator using an multiplicative multi-locus model over three additive single-locus models at three diesease predisposing loci.

Listing 4.47: A multi-loci penetrance model

```
>>> import simuPOP as sim
```

```
>>> pop = sim.Population(5000, loci=3)
>>> sim.initGenotype(pop, freq=[0.2]*5)
>>> # the multi-loci penetrance
>>> sim.mlPenetrance(pop, mode=sim.MULTIPLICATIVE,
...     ops = [sim.MaPenetrance(loci=loc,
...         penetrance=[0, 0.3, 0.6]) for loc in range(3)])
>>> # count the number of affected individuals.
>>> sim.stat(pop, numOfAffected=True)
>>> pop.dvars().numOfAffected
542
```

4.7.4 Hybrid penetrance model (operator `PyPenetrance`)

When your selection model involves multiple interacting genetic and environmental factors, it might be easier to calculate a penetrance value explicitly using a Python function. A hybrid penetrance operator can be used for this purpose. If your penetrance model depends solely on genotype, you can define a function such as

```
def pfunc(geno):
    # calculate penetrance according to genotype at specified loci
    # in the order of A1,A2,B1,B2,C1,C2 for loci A,B,C (for diploid)
    return val
```

and use this function in an operator `PySelector(func=pfunc, loci=loci)`. If your penetrance model depends on genotype as well as some information fields, you can define a function in the form of

```
def pfunc(geno, fields):
    # calculate penetrance according to genotype at specified loci
    # and values at specified informaton fields.
    return val
```

and use this function in an operator `PySelector(func=pfunc, loci=loci, paramFields=fields)`. If the function you provide accepts three arguments, `PyPenetrance` will pass generation number as the third argument so that you could implement generation-specific penetrance models (e.g. `pfunc(geno, fields, gen)`).

When a `PyPenetrance` operator is used to calculate penetrance for an individual, it will collect his or her genotype at specified loci, optional values at specified information fields, and the generation number to a user-specified Python function, and take its return value as penetrance. As you can imagine, the incorporation of information fields and generation number allow the implementation of very complex penetrance scenarios such as gene environment interaction and varying selection pressures. Note that this operator does not pass sex and affection status to the user-defined function. If your selection model is sex-dependent, you can define an information field `sex`, synchronize its value with individual sex (e.g. using operator `InfoExec('sex=ind.sex()', exposeInd='ind')` and pass this information to the user-defined function (`PySelector(func=func, paramFields='sex')`).

Example 4.56 demonstrates how to use a `PyPenetrance` to specify penetrance values according to a fitness table and the smoking status of each individual. In this example, Individual risk is doubled when he or she smokes. The disease prevalence is therefore much higher in smokers than in non-smokers.

Listing 4.48: A hybrid penetrance model

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=2000, loci=[1]*2, infoFields=['p', 'smoking'])
>>> pop.setVirtualSplitter(sim.InfoSplitter(field='smoking', values=[0,1]))
>>> # the second parameter gen can be used for varying selection pressure
>>> def penet(geno, smoking):
...     #     BB     Bb      bb
```

```
...     # AA  0.01   0.01    0.01
...     # Aa  0.01   0.03    0.03
...     # aa  0.01   0.03    0.05
...     #
...     # geno is (A1 A2 B1 B2)
...     if geno[0] + geno[1] == 1 and geno[2] + geno[3] != 0:
...         v = 0.03   # case of AaBb
...     elif geno[0] + geno[1] == 2 and geno[2] + geno[3] == 1:
...         v = 0.03   # case of aaBb
...     elif geno[0] + geno[1] ==2 and geno[2] + geno[3] == 2:
...         v = 0.05   # case of aabb
...     else:
...         v = 0.01   # other cases
...     if smoking:
...         return v * 2
...     else:
...         return v
...
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5]),
...         sim.PyOutput('Calculate prevalence in smoker and non-smokers\n'),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         # set smoking status randomly
...         sim.InitInfo(lambda : random.randint(0,1), infoFields='smoking'),
...         # assign affection status
...         sim.PyPenetrance(loci=[0, 1], func=penet),
...         sim.Stat(numOfAffected=True, subPops=[(0, sim.ALL_AVAIL)],
...             vars='propOfAffected_sp', step=20),
...         sim.PyEval(r"'Non-smoker: %.2f%%\tSmoker: %.2f%%\n' % "
...             "(subPop[(0,0)]['propOfAffected']*100, subPop[(0,1)]['propOfAffected']*100)",
...             step=20)
...     ],
...     gen = 50
... )
Calculate prevalence in smoker and non-smokers
Non-smoker: 2.06%       Smoker: 4.47%
Non-smoker: 1.94%       Smoker: 4.22%
Non-smoker: 1.62%       Smoker: 5.32%
50L
>>>
```

4.8 Quantitative trait

Quantitative traits are naturally stored in information fields of each individual. A quantitative trait operator assigns quantitative trait fields according to individual genetic (genotype) and environmental (other information fields) information. Although a large number of quantitative trait models have been used in theoretical and empirical studies, no model is popular enough to deserve a specialized operator. Therefore, only one hybrid operator is currently provided in simuPOP.

4.8.1 A hybrid quantitative trait operator (operator `PyQuanTrait`)

Operator `PyQuanTrait` accepts a user defined function that returns quantitative trait values for specified information fields. This operator can comunicate with functions in one of the forms of `func(geno)`, `func(geno, field_name, ...)` or `func(geno, field_name, gen)` where `field_name` should be name of existing fields. simuPOP will pass genotype and value of specified fields according to name of the passed function. Note that geno are arrange locus by locus, namely in the order of `A1`,`A2`,`B1`,`B2` for loci `A` and `B`.

A quantitative trait operator can be applied before or after mating and assign values to the trait fields of all parents or offspring, respectively. It can also be applied during mating to assign trait values to offspring. Example 4.49 demonstrates the use of this operator, using two trait fields `trait1` and `trait2` which are determined by individual genotype and age. This example also demonstrates how to calculate statistics within virtual subpopulations (defined by age).

Listing 4.49: A hybrid quantitative trait model

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=5000, loci=2, infoFields=['qtrait1', 'qtrait2', 'age'])
>>> pop.setVirtualSplitter(sim.InfoSplitter(field='age', cutoff=[40]))
>>> def qtrait(geno, age):
...     'Return two traits that depends on genotype and age'
...     return random.normalvariate(age * sum(geno), 10), random.randint(0, 10*sum(geno))
...
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2, 0.8]),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         # use random age for simplicity
...         sim.InitInfo(lambda:random.randint(20, 75), infoFields='age'),
...         sim.PyQuanTrait(loci=(0,1), func=qtrait, infoFields=['qtrait1', 'qtrait2']),
...         sim.Stat(meanOfInfo=['qtrait1'], subPops=[(0, sim.ALL_AVAIL)],
...             vars='meanOfInfo_sp'),
...         sim.PyEval(r"'Mean of trait1: %.3f (age < 40), %.3f (age >=40)\n' % "
...             "(subPop[(0,0)]['meanOfInfo']['qtrait1'], subPop[(0,1)]['meanOfInfo']['qtrait1'])"),
...     ],
...     gen = 5
... )
Mean of trait1: 93.976 (age < 40), 183.363 (age >=40)
Mean of trait1: 93.717 (age < 40), 183.171 (age >=40)
Mean of trait1: 94.383 (age < 40), 182.793 (age >=40)
Mean of trait1: 94.645 (age < 40), 183.514 (age >=40)
Mean of trait1: 95.420 (age < 40), 183.887 (age >=40)
5L
>>>
```

4.9 Natural Selection

4.9.1 Natural selection through the selection of parents

In the simplest scenario, natural selection is implemented in two steps:

- Before mating happens, an operator (called a **selector**) goes through a population and assign each individual a fitness value. The fitness values are stored in an information field called `fitness`.
- When mating happens, parents are chosen with probabilities that are proportional to their fitness values. For example, assuming that a parental population consists of four Individuals with fitness values 1, 2, 3, and 4, respectively, the probability that they are picked to produce offspring are $1/(1+2+3+4) = 0.1$, 0.2, 0.3, and 0.4 respectively. As you can image, if the offspring population has 10 individuals, the four parents will on average parent 1, 2, 3 and 4 offspring.

Because parents with lower fitness values have less chance to be produce offspring, their genotypes have less chance to be passed to an offspring generation. If the decreased fitness is caused by the presence of certain mutant (e.g. a mutant causing a serious disease), individuals with that mutant will have less change to survive and effecitively reduce or eleminate that mutant from the population.

Example 4.50 gives an example of natural selection. In this example, a `MapSelector` is used to explicitly assign fitness value to genotypes at the first locus. The fitness values are `1`, `0.98`, `0.97` for genotypes `00`, `01` and `11` respectively. The selector set individual fitness values to information field `fitness` before mating happens. The `RandomMating` mating scheme then selects parents according to parental fitness values.

Listing 4.50: Natural selection through the selection of parents

```
>>> import simuPOP as sim
>>> pop = sim.Population(4000, loci=1, infoFields='fitness')
>>> simu = sim.Simulator(pop, rep=3)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     preOps=sim.MapSelector(loci=0, fitness={(0,0):1, (0,1):0.98, (1,1):0.97}),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0, step=10),
...         sim.PyEval("'Gen:%3d ' % gen", reps=0, step=10),
...         sim.PyEval(r"'%.3f\t' % alleleFreq[0][1]", step=10),
...         sim.PyOutput('\n', reps=-1, step=10)
...     ],
...     gen = 50
... )
Gen:  0 0.490   0.492   0.487
Gen: 10 0.433   0.430   0.431
Gen: 20 0.403   0.390   0.419
Gen: 30 0.343   0.325   0.383
Gen: 40 0.303   0.297   0.334
(50L, 50L, 50L)
```

Note: The selection algorithm used in simuPOP is called *fitness proportionate selection*, or *roulette-wheel selection*. simuPOP does not use the more efficient *stochastic universal sampling* algorithm because the number of needed offspring is unknown in advance.

4.9.2 Natural selection through the selection of offspring *

Natural selection can also be implemented as selection of offspring. Remember that an individual will be discarded if one of the during-mating operators fails (return `False`), **a during-mating selector discards offspring according to fitness values of offspring**. Instead of relative fitness that will be compared against other individuals during

the selection of parents, **fitness values of a during-mating selector are considered as absolute fitness which are probabilities to survive** and have to be between 0 and 1.

A during-mating selector works as follows:

1. During evolution, parents are chosen randomly to produce one or more offspring. (Nothing prevents you from choosing parents according to their fitness values, but it is rarely justifiable to apply natural selection to both parents and offspring.)
2. A selection operator is applied to each offspring during mating and determines his or her fitness value. The fitness value is considered as probability to survive so an offspring will be discarded (operator returns `False`) if the fitnessvalue is larger than an uniform random number.
3. Repeat steps 1 and 2 until the offspring generation is populated.

Because many offspring will be generated and discarded, especially when offspring fitness values are low, selection through offspring is less efficient than selection through parents. In addition, absolute fitness is usually more difficult to estimate than relative fitness. So, unless there are compelling reasons (e.g. simulating realistic scenarios of survival competition among offspring), selection through parents are recommended.

Example 4.51 gives an example of natural selection through the selection of offspring. This example looks almost identical to Example 4.50 but the underlying selection mechanism is quite different. Note that selection through offspring does not save fitness values to an information field so you do not need to add information field fitness to the population.

Listing 4.51: Natural selection through the selection of offspring

```
>>> import simuPOP as sim
>>> pop = sim.Population(10000, loci=1)
>>> simu = sim.Simulator(pop, rep=3)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.MapSelector(loci=0, fitness={(0,0):1, (0,1):0.98, (1,1):0.97}),
...     ]),
...     postOps=[
...         sim.Stat(alleleFreq=0, step=10),
...         sim.PyEval("'Gen:%3d ' % gen", reps=0, step=10),
...         sim.PyEval(r"'%.3f\t' % alleleFreq[0][1]", step=10),
...         sim.PyOutput('\n', reps=-1, step=10)
...     ],
...     gen = 50
... )
Gen:  0 0.493   0.493   0.496
Gen: 10 0.461   0.464   0.465
Gen: 20 0.436   0.445   0.442
Gen: 30 0.389   0.386   0.385
Gen: 40 0.370   0.345   0.348
(50L, 50L, 50L)
```

4.9.3 Are two selection scenarios equivalent? **

If you look closely at Examples 4.50 and 4.51, you will notice that their results are quite similar. This is actually what you should expect in most cases. Let us look at the theoretical consequence of selection through parents or offspring in a simple case with asexual mating.

Assuming a diallelic marker with three genotypes g_{AA}, g_{Aa} and g_{aa}, with frequencies P_{AA}, P_{Aa} and P_{aa}, and relative fitness values w_{AA}, w_{Aa}, and w_{22} respectively. If we select through offspring, the proportion of genotype g_{AA} etc., should be

$$P'_{AA} = \frac{P_{AA}w_{AA}}{P_{AA}w_{AA} + P_{Aa}w_{Aa} + P_{aa}w_{aa}}$$
$$P'_{Aa} = \frac{P_{Aa}w_{Aa}}{P_{AA}w_{AA} + P_{Aa}w_{Aa} + P_{aa}w_{aa}}$$
$$P'_{aa} = \frac{P_{aa}w_{aa}}{P_{AA}w_{AA} + P_{Aa}w_{Aa} + P_{aa}w_{aa}}$$

because offspring genotypes are randomly drawn from the parental generation, and each offspring has certain probability to survive.

Now, if we select through parents, the proportion of parents with genotype AA will be the number of AA individuals times its probability to be chosen:

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

This is, however, exactly

$$n_{AA}\frac{w_{AA}}{\sum_{n=1}^{N} w_n} = \frac{n_{AA}w_{AA}}{n_{AA}w_{AA} + n_{Aa}w_{Aa} + n_{aa}w_{aa}} = \frac{P_{AA}w_{AA}}{P_{AA}w_{AA} + P_{Aa}w_{Aa} + P_{aa}w_{aa}} = P'_{AA}$$

which corresponds to the proportion of offspring with such genotype. That is to say, **in this simple case, two types of selection scenarios yield identical results**.

These two types of selection scenarios do not have to always yield identical results. Exceptions exist in cases with more than one offspring or sexual mating with sex-specific survival rate. simuPOP provides both selection implementations and you should choose one of them for your particular simulation.

4.9.4 Map selector (operator `MapSelector`)

A map selector uses a Python dictionary to provide fitness values for each type of genotype. For example, Example 4.52 uses a dictionary with keys `(0,0)`, `(0,1)` and `(1,1)` to specify fitness values for individuals with these genotypes at locus 0. This example is a typical example of heterozygote advantage. When $w_{11} < w_{12} > w_{22}$, the genotype frequencies will go to an equilibrium state. Theoretically, if $s_1 = w_{12} - w_{11}$ and $s_2 = w_{12} - w_{22}$, the stable allele frequency of allele 0 is

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

which is $\frac{2}{3}$ in the example ($s_1 = .1$, $s_2 = .2$).

Listing 4.52: A selector that uses pre-defined fitness value

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=1000, loci=1, infoFields='fitness')
>>> s1 = .1
>>> s2 = .2
>>> pop.evolve(
...     initOps=[
```

```
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.2, .8])
...     ],
...     preOps=sim.MapSelector(loci=0, fitness={(0,0):1-s1, (0,1):1, (1,1):1-s2}),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.PyEval(r"'%.4f\n' % alleleFreq[0][0]", step=100)
...     ],
...     gen=301
... )
0.2250
0.6605
0.6530
0.6870
301L
>>>
```

The above example assumes that the fitness value for individuals with genotypes `(0,1)` and `(1,0)` are the same. This assumption is usually valid but can be vialoated with impriting. In that case, you can specify fitness for both types of genotypes. The underlying mechanism is that the `MapSelector` looks up a genotype in the dictionary first directly, and then without phase information if a genotype is not found.

This operator supports haplodiploid populations and sex chromosomes. In these cases, only valid alleles should be listed which can lead to dictionary keys with different lengths. In addition, although less used because of potentially a large number of keys, this operator can act on multiple loci. Please refer to `MapPenetrance` for details.

4.9.5 Multi-allele selector (operator `MaSelector`)

A multi-allele selector divides alleles into two groups, wildtype A and mutants a, and treat alleles within each group as the same. The fitness model is therefore simplified to

- Two fitness values for genotype A, a in the haploid case
- Three fitness values for genotype AA, Aa and aa in the diploid single locus case. Genotype Aa and aA are assumed to have the same impact on fitness.

The default wildtype group contains allele 0 so the two allele groups are zero and non-zero alleles. Example 4.53 demonstrates the use of this operator. This example is identical to Example 4.52 except that there are five alleles at locus 0 and alleles 1, 2, 3, 4 are treated as a single non-widetype group.

Listing 4.53: A multi-allele selector

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=1000, loci=1, infoFields='fitness')
>>> s1 = .1
>>> s2 = .2
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.2] * 5)
...     ],
...     preOps=sim.MaSelector(loci=0, fitness=[1-s1, 1, 1-s2]),
...     matingScheme=sim.RandomMating(),
...     postOps=[
```

```
...         sim.Stat(alleleFreq=0),
...         sim.PyEval(r"'%.4f\n' % alleleFreq[0][0]", step=100)
...     ],
...     gen = 301)
0.2250
0.6605
0.6530
0.6870
301L
```

Operator `MaSelector` also supports multiple loci by specifying fitness values for all combination of genotype at specified loci. In the case of two loci, this operator requires

- Four fitness values for genotype `AB`, `Ab`, `aB` and `ab` in the haploid case,
- Nine fitness values for genotype `AABB`, `AABb`, `AAbb`, `AaBB`, `AaBb`, `Aabb`, `aaBB`, `aaBb`, and `aabb` in the haploid case.

In general, 2^n values are needed for haploid populations and 3^n values are needed for diploid populations where n is the number of loci. This operator does not yet support haplodiploid populations and sex chromosomes. Example 4.54 demonstrates the use of a multi-locus model in a haploid population.

Listing 4.54: A multi-locus multi-allele selection model in a haploid population

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=10000, ploidy=1, loci=[1,1], infoFields='fitness')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])
...     ],
...     # fitness values for AB, Ab, aB and ab
...     preOps=sim.MaSelector(loci=[0,1], fitness=[1, 1, 1, 0.95]),
...     matingScheme=sim.RandomSelection(),
...     postOps=[
...         sim.Stat(haploFreq=[0, 1], step=25),
...         sim.PyEval(r"'%.3f\t%.3f\t%.3f\t%.3f\n' % (haploFreq[(0,1)][(0,0)],"
...                 "haploFreq[(0,1)][(0,1)], haploFreq[(0,1)][(1,0)],"
...                 "haploFreq[(0,1)][(1,1)])", step=25)
...     ],
...     gen = 100
... )
0.264   0.243   0.252   0.240
0.292   0.294   0.321   0.093
0.339   0.330   0.303   0.027
0.310   0.383   0.297   0.009
100L
```

4.9.6 Multi-locus selection models (operator `MlSelector`)

Although an individual's fitness can be affected by several factors, each of which can be modeled individually, **only one fitness value is used to determine a person's ability to pass all these factors to his or her offspring**. Although in theory we sometimes assume independent evolution of disease predisposing loci (mostly for mathematical reasons), in practise we have to use a multi-locus selection model to combine single-locus models.

This multi-loci selector applies several selectors to each individual and computes an overall fitness value from the fitness values provided by these selectors. Although this selector is designed to obtain multi-loci fitness values from several single-locus fitness models, any selector, including those obtain their fitness values from multiple disease predisposing loci, can be used in this selector. This selector uses parameter `mode` to control how individual fitness values are combined. More specifically, if f_i are fitness values obtained from individual selectors, this selector returns

- $\Pi_i f_i$ if `mode=MULTIPLICATIVE`, and
- $1 - \sum_i (1 - f_i)$ if `mode=ADDITIVE`, and
- $1 - \Pi_i (1 - f_i)$ if `mode=HETEROGENEITY`

0 will be returned if the returned fitness value is less than 0.

This operator simply combines individual fitness values and it is your responsibility to apply and interpret these models. For example, if relative fitness values are greater than one, the heterogeneity model hardly makes sense. Example 4.55 demonstrates the use of this operator using an additive multi-locus model over an additive and a recessive single-locus model at two diesease predisposing loci. For comparison, we simulate two additional replicates with selection only applying to one of the two loci. It would be interesting to see if these two loci evolve more or less independently by comparing allele freqency trajectories of these two replicates to those in the first replicate.

Listing 4.55: A multi-loci selector

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=10000, loci=2, infoFields='fitness')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])
...     ],
...     preOps=[
...         sim.MlSelector([
...             sim.MapSelector(loci=0, fitness={(0,0):1, (0,1):1, (1,1):.8}),
...             sim.MapSelector(loci=1, fitness={(0,0):1, (0,1):0.9, (1,1):.8}),
...             ], mode = sim.ADDITIVE, reps=0),
...         sim.MapSelector(loci=0, fitness={(0,0):1, (0,1):1, (1,1):.8}, reps=1),
...         sim.MapSelector(loci=1, fitness={(0,0):1, (0,1):0.9, (1,1):.8}, reps=2)
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...          sim.Stat(alleleFreq=[0,1]),
...          sim.PyEval(r"'REP %d:\t%.3f\t%.3f\t' % (rep, alleleFreq[0][1], alleleFreq[1][1])"),
...          sim.PyOutput('\n', reps=-1),
...     ],
...     gen = 5
... )
REP 0:  0.472   0.465
REP 0:  0.452   0.429
REP 0:  0.429   0.397
REP 0:  0.405   0.378
REP 0:  0.382   0.355
5L
```

4.9.7 A hybrid selector (operator `PySelector`)

When your selection model involves multiple interacting genetic and environmental factors, it might be easier to calculate a fitness value explicitly using a Python function. A hybrid selector can be used for this purpose. If your selection model depends solely on genotype, you can define a function such as

```
def fitness_func(geno):
    # calculate fitness according to genotype at specified loci
    # genotypes are arrange locus by locus, namely A1,A2,B1,B2 for loci A and B
    return val
```

and use this function in an operator `PySelector(func=fitness_func, loci=loci)`. If your selection model depends on genotype as well as some information fields, you can define a function in the form of

```
def fitness_func(geno, field1, field2):
    # calculate fitness according to genotype at specified loci
    # and values at specified informaton fields.
    return val
```

where `field1`, `field2` are names of information fields. simuPOP will pass genotype and value of specified fields according to name of the passed function. Note that genotypes are arrange locus by locus, namely in the order of `A1`,`A2`,`B1`,`B2` for loci `A` and `B`. Other parameters such as `gen`, `ind`, and `pop` are also allowed. Please check the reference manual for details.

When a `PySelector` is used to calculate fitness for an individual (parents if applied pre-mating, offspring if applied during-mating), it will collect his or her genotype at specified loci, optional values at specified information fields, generation number, or individual to a user-specified Python function, and take its return value as fitness. As you can imagine, the incorporation of information fields and generation number allow the implementation of very complex selection scenarios such as gene environment interaction and varying selection pressures.

Example 4.56 demonstrates how to use a `PySelector` to specify fitness values according to a fitness table and the smoking status of each individual.

Listing 4.56: A hybrid selector

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=2000, loci=[1]*2, infoFields=['fitness', 'smoking'])
>>> s1 = .02
>>> s2 = .03
>>> # the second parameter gen can be used for varying selection pressure
>>> def sel(geno, smoking):
...     #     BB  Bb   bb
...     # AA  1   1    1
...     # Aa  1   1-s1 1-s2
...     # aa  1   1    1-s2
...     #
...     # geno is (A1 A2 B1 B2)
...     if geno[0] + geno[1] == 1 and geno[2] + geno[3] == 1:
...         v = 1 - s1  # case of AaBb
...     elif geno[2] + geno[3] == 2:
...         v = 1 - s2  # case of ??bb
...     else:
...         v = 1       # other cases
...     if smoking:
...         return v * 0.9
...     else:
```

```
...         return v
...
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])
...     ],
...     preOps=sim.PySelector(loci=[0, 1], func=sel),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         # set smoking status randomly
...         sim.InitInfo(lambda : random.randint(0,1), infoFields='smoking'),
...         sim.Stat(alleleFreq=[0, 1], step=20),
...         sim.PyEval(r"'%.4f\t%.4f\n' % (alleleFreq[0][1], alleleFreq[1][1])", step=20)
...     ],
...     gen = 50
... )
0.4943  0.4890
0.5032  0.4260
0.4305  0.3780
50L
```

4.9.8 Multi-locus random fitness effects (operator `PyMlSelector`)

If the fitness of individuals is determined by fitness effects over a large number of loci, both `MlSelector` and `PySelector` are difficult to use because the former requires a large number of single-locus selectors, and the latter requires the processing long genome sequences. If the overall fitness can be determined by fitness effects of mutants, a `PyMlSelector` can be used. This operator

- Calls a user-provided call-back function for each locus with at least a mutant (non-zero allele). The function can accept location and genotype so the fitness can be location and genotype dependent. The return value is cached so the function will be called only once for each locus-genotype pair.
- The fitness of each individual is determined by fitness values of loci with at least one mutant, using the same methods as operator `MlSelector`. This implicitly assumes that loci without any mutant have fitness value 1 and will not contribute to the final fitness value.

Example 4.56 demonstrates how to use a `PyMlSelector` to implement a fitness model where each mutant has a random fitness drawn from a Gamma distribution. An additive model is used so a homozygote will have a fitness penalty that doubles that of a heterozygote. Because the fitness values of heterozygote and homozygote at each locus are requested separately, a class is used to store locus-specific s values.

The fitness value of each locus-genotype pair is outputted to a file, and it should be interesting to plot the distribution of allele frequency at each locus against the fitness values, because mutants that suffer from stronger negative natural selection are supposed to be rarer.

Listing 4.57: Random fitness effect

```
>>> import simuOpt
>>> simuOpt.setOptions(quiet=True, alleleType='mutant')
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=2000, loci=[10000], infoFields=['fitness'])
>>>
>>> class GammaDistributedFitness:
```

```
...     def __init__(self, alpha, beta):
...         self.coefMap = {}
...         self.alpha = alpha
...         self.beta = beta
...
...     def __call__(self, loc, alleles):
...         # because s is assigned for each locus, we need to make sure the
...         # same s is used for fitness of genotypes 01 (1-s) and 11 (1-2s)
...         # at each locus
...         if loc in self.coefMap:
...             s = self.coefMap[loc]
...         else:
...             s = random.gammavariate(self.alpha, self.beta)
...             self.coefMap[loc] = s
...         #
...         if 0 in alleles:
...             return 1. - s
...         else:
...             return 1. - 2.*s
...
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=[
...         sim.AcgtMutator(rate=[0.00001], model='JC69'),
...         sim.PyMlSelector(GammaDistributedFitness(0.23, 0.185),
...             output='>>sel.txt'),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(numOfSegSites=sim.ALL_AVAIL, step=50),
...         sim.PyEval(r"'Gen: %2d #seg sites: %d\n' % (gen, numOfSegSites)",
...             step=50)
...     ],
...     gen = 201
... )
Gen:  0 #seg sites: 182
Gen: 50 #seg sites: 1343
Gen: 100 #seg sites: 1490
Gen: 150 #seg sites: 1588
Gen: 200 #seg sites: 1675
201L
>>> print(''.join(open('sel.txt').readlines()[:5]))
5855    1       0       0.999984
1085    2       0       0.999972
2907    0       1       0.926761
7773    0       1       0.995197
1835    0       2       0.963785
```

4.9.9 Alternative implementations of natural selection

If you know how natural selection works in simuPOP, you do not have to use a selector to perform natural selection. For example,

- If you choose to use fitness values of parents to perform probabilistic natural selection during mating, you just need to set individual fitness in some way before mating. (You do not even have to use information field `fitness` because you can specify which information field to use in a mating scheme using parameter `selectionField`). This can be done through a penetrance model (as shown in the following example) where affected individuals are selected against during mating, a quantitative trait model (where a trait is defined to control individual fitness), or by setting information field fitness manually through a Python operator.
- If you would like to perform deterministic selection on certain phenotype, you can explicitly remove individuals before or during mating. More explicitly, you can use an operator `DiscardIf` to remove parents before mating or remove offspring during mating according to certain status (disease status or quantitative trait), provided that the trait status is defined before this operator is applied.

Example 4.58 demonstrates a commonly used case where parents who are affected with certain disease are excluded from producing offspring. In this example, a penetrance model (operator `MaPenetrance`) is applied to the parental generation to determine who will be affected. An `InfoExec` operator is used to set individual fitness to 1 if he or she is unaffected, and 0 if he or she is affected. Due to the way parents are selected, affected parents will not be able to produce offspring as long as there is any unaffected individual. Because individual affection status is determined by his or her genotype, this genotype - affection status - fitness relationship could be implemented using an equivalent `MaSelector`. This method could be extended to `InfoExec('fitness = 1 - 0.01*ind.affected()', exposeInd='ind')` to select against, but not remove, affected parents, and similarly `InfoExec('fitness = 1 - 0.01*(LDL > 250)')` to select against individuals according to a quantitative trait. For this particular example, a `DiscardIf` operator could be used, although it can be slower because of the explicit removal of parents.

Listing 4.58: Natural selection according to individual affection status

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=2000, loci=1, infoFields='fitness')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])
...     ],
...     preOps=[
...         sim.MaPenetrance(loci=0, penetrance=[0.01, 0.1, 0.2]),
...         sim.Stat(numOfAffected=True, step=25, vars='propOfAffected'),
...         sim.PyEval(r"'Percent of affected: %.3f\t' % propOfAffected", step=50),
...         sim.InfoExec('fitness = not ind.affected()', exposeInd='ind')
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.PyEval(r"'%.4f\n' % alleleFreq[0][1]", step=50)
...     ],
...     gen=151
... )
Percent of affected: 0.110      0.4713
Percent of affected: 0.009      0.0095
Percent of affected: 0.013      0.0000
Percent of affected: 0.008      0.0000
151L
```

4.9.10 Frequency dependent or dynamic selection pressure *

If individual fitness depends on individual information fields and/or population variables, you will have to calculate individual fitness using expressions or functions. In order to access individual information fields and population

variable and calculate individual fitness, you have the option to

- Use a `PySelector` and pass genotype, values of information fields, references to individual and population to a user-provided function, which returns fitness value for each individual.
- Use of `PyOperator` to obtain information of the population (e.g. variables) and all individuals. Determine individual fitness and set information field `fitness` of all individuals.
- Use an operator `InfoExec` to calculate individual fitness using expressions. This method can be more efficient than others because simuPOP does not have to call a user-provided function.

Example 4.59 demonstrates an example where the fitness values of individuals are calculated from allele frequencies calculated using a `Stat` operator. Because the fitness values of individuals are 1, $1-(p-0.5)*0.1$, $1-(p-0.5)*0.2$ for genotype 00, 01 and 11 where p is the frequency of allele 1, this allele will be under purifying selection if its frequency is over 0.5, and positive selection if its frequency is less than 0.5. Consequently, the frequency of this allele will oscillate around 0.5 during evolution, as shown in the result of this example.

Listing 4.59: Frequency dependent selection

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=2000, loci=1, infoFields='fitness')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])
...     ],
...     preOps=[
...         sim.Stat(alleleFreq=0),
...         sim.InfoExec('''fitness = {
...             0: 1,
...             1: 1 - (alleleFreq[0][1] - 0.5)*0.1,
...             2: 1 - (alleleFreq[0][1] - 0.5)*0.2}[ind.allele(0,0)+ind.allele(0,1)]''',
...             exposeInd='ind'),
...         sim.Stat(meanOfInfo='fitness'),
...         sim.PyEval(r"'alleleFreq=%.3f, mean fitness=%.5f\n' % (alleleFreq[0][1], meanOfInfo['fitness'])",
...             step=25),
...     ],
...     matingScheme=sim.RandomMating(),
...     gen=151
... )
alleleFreq=0.495, mean fitness=1.00045
alleleFreq=0.504, mean fitness=0.99955
alleleFreq=0.484, mean fitness=1.00150
alleleFreq=0.492, mean fitness=1.00076
alleleFreq=0.499, mean fitness=1.00005
alleleFreq=0.526, mean fitness=0.99726
alleleFreq=0.514, mean fitness=0.99856
151L
```

4.9.11 Support for virtual subpopulations *

Support for virtual subpopulations allows you to use different selectors for different (virtual) subpopulations. Because virtual subpopulations may overlap, and they do not have to cover all individuals in a subpopulation, it is important to remember that

- If virtual subpopulations overlap, the fitness value set by the last selector will be used.
- If an individual is not included in any of the virtual subpopulation, its fitness value will be zero which will prevent them from producing any offspring.

Example 4.60 demonstrates how to apply selectors to virtual subpopulations. This example has two subpopulations, each having two virtual subpopulations defined by sex. Natural selection is applied to male individuals in the first subpopulation, and female individuals in the second subpopulation. However, because the sex of offspring is randomly determined, the selection actually decreases the disease allele frequency for all inviduals.

Listing 4.60: Selector in virtual subpopulations

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[5000, 5000], loci=1, infoFields='fitness')
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])
...     ],
...     preOps=[
...         sim.MaSelector(loci=0, fitness=[1, 1, 0.98], subPops=[(0,0), (1,1)]),
...         sim.MaSelector(loci=0, fitness=[1, 0.99, 0.98], subPops=[(0,1), (1,0)]),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=[0], subPops=[(sim.ALL_AVAIL, sim.ALL_AVAIL)],
...             vars='alleleFreq_sp', step=50),
...         sim.PyEval(r"'%.4f\t%.4f\t%.4f\t%.4f\n' % "
...             "tuple([subPop[x]['alleleFreq'][0][1] for x in ((0,0),(0,1),(1,0),(1,1))])",
...             step=50)
...     ],
...     gen=151
... )
0.5022  0.5083  0.4970  0.5020
0.4086  0.4054  0.3849  0.3817
0.3275  0.3259  0.2435  0.2532
0.2715  0.2662  0.1305  0.1338
151L
```

Selecting through offspring can also be applied to virtual subpopulations. For example, Example 4.61 moves the selectors to the `ops` parameter of `RandomMating`. In this way, male and female offspring will have different survival probabilities according to their genotype.

Listing 4.61: Selection against offspring in virtual subpopulations

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[5000, 5000], loci=1, infoFields='fitness')
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.MaSelector(loci=0, fitness=[1, 1, 0.98], subPops=[(0,0), (1,1)]),
```

```
...         sim.MaSelector(loci=0, fitness=[1, 0.99, 0.98], subPops=[(0,1), (1,0)]),
...         ]),
...     postOps=[
...         sim.Stat(alleleFreq=[0], subPops=[(sim.ALL_AVAIL, sim.ALL_AVAIL)],
...             vars='alleleFreq_sp', step=50),
...         sim.PyEval(r"'%.4f\t%.4f\t%.4f\t%.4f\n' % "
...             "tuple([subPop[x]['alleleFreq'][0][1] for x in ((0,0),(0,1),(1,0),(1,1))])",
...             step=50)
...     ],
...     gen=151
... )
0.5018  0.5034  0.4941  0.4853
0.3652  0.3728  0.3820  0.3766
0.2882  0.2920  0.2590  0.2667
0.2083  0.1994  0.2378  0.2356
151L
```

4.9.12 Natural selection in heterogeneous mating schemes **

Multiple mating schemes could be applied to the same subpopulation in a heterogeneous mating scheme (`HeteroMating`). These mating schemes may or may not support natural selection, may be applied to different virtual subpopulations of population, and they may see Individuals differently in terms of individual fitness. Parameter `fitnessField` of a mating scheme could be used to handle such cases. More specifically,

- You can turn off the natural selection support of a mating scheme by setting `fitnessField=''`.
- If a mating scheme uses a different set of fitness values, you can add an information field (e.g. `fitness1`), setting individual fitness to this information field using a selector (with parameter `infoFields='fitness1'`) and tells a mating scheme to look in this information field for fitness values (using parameter `fitnessField='fitness1'`).

4.10 Tagging operators

In simuPOP, tagging refers to the action of setting various information fields of offspring, usually using various parental information during the production of offspring. simuPOP provides a number of tagging operators (called taggers) for various purposes. Because tagging operators are during-mating operators, parameter `subPops` can be used to tag only offspring that belong to specified virtual subpopulation. (e.g. all male offspring)

4.10.1 Inheritance tagger (operator `InheritTagger`)

An inheritance tagger passes values of parental information field(s) to the corresponding offspring information field(s). Depending on the parameters, an InheritTagger can

- For asexual mating schemes, pass one or more information fields from parent to offspring.
- Pass one or more information fields from father to offspring (`mode=PATERNAL`).
- Pass one or more information fields from mother to offspring (`mode=MATERNAL`).
- Pass the maximal, minimal, sum, multiplcation or average of values of one or more information fields of both parents (`mode=MAXIMUM`, `MINIMUM`, `ADDITION, MULTIPLICATION` or `MEAN`).

This can be used to track the spread of certain information during evolution. For example, Example4.62 tags the first individuals of ten subpopulations of size 1000. individuals in the offspring generation inherits the maximum value of field `x` from his/her parents so `x` is inherited regardless of the sex of parents. A Stat operator is used to calculate the number of offspring having this tag in each subpopulation. The results show that some tagged ancestors have many offspring, and some have none. If you run this simulation long enough, you can see that all ancestors become the ancestor of either none or all individuals in a population. Note that this simulation only considers genealogical inheritance and ancestors do not have to pass any genotype to the last generation.

Listing 4.62: Use an inherit tagger to track offspring of individuals

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000]*10, loci=1, infoFields='x')
>>> # tag the first individual of each subpopulation.
>>> for sp in range(pop.numSubPop()):
...     pop.individual(0, sp).x = 1
...
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.InheritTagger(mode=sim.MAXIMUM, infoFields='x'),
...     ]),
...     postOps=[
...         sim.Stat(sumOfInfo='x', vars=['sumOfInfo_sp']),
...         sim.PyEval(r'", ".join(["%3d" % subPop[i]["sumOfInfo"]["x"] for i in range(10)])+"\n"'),
...     ],
...     gen = 5
... )
  2,   1,   0,   1,   1,   2,   3,   3,   1,   1
  5,   1,   0,   1,   1,   3,   3,   5,   3,   0
  9,   2,   0,   2,   2,   7,   9,   5,  13,   0
 21,   4,   0,   2,   5,  18,  11,   9,  27,   0
 39,   5,   0,   6,   8,  36,  23,  20,  67,   0
5L
```

4.10.2 Summarize parental informatin fields (operator `SummaryTagger`)

A `SummaryTagger` summarize values of one or more parental information fields and place the result in an offspring information field. If mating is sexual, two sets of values will be involved. Summarization methods include `MEAN`, `MINIMUM`, `MAXIMUM`, `SUMMATION` and `MULTIPLICATION`. The operator is usually used to summarize certain characteristic of parents of each offspring. For example, a `SummaryTagger` is used in Example 4.63 to calculate the mean fitness of parents during each mating event. The results are saved in the `avgFitness` field of offspring. Because allele 1 at locus 0 is under purifying selection, the allele frequency of this allele decreases. In the mean time, fitness of parents increases because less and less parents have this allele.

Listing 4.63: Using a summary tagger to calculate mean fitness of parents.

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=1, infoFields=['fitness', 'avgFitness'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...     ],
...     preOps=sim.MaSelector(loci=0, wildtype=0, fitness=[1, 0.99, 0.95]),
```

```
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.SummaryTagger(mode=sim.MEAN, infoFields=['fitness', 'avgFitness']),
...     ]),
...     postOps=[
...         sim.Stat(alleleFreq=0, meanOfInfo='avgFitness', step=10),
...         sim.PyEval(r"'gen %d: allele freq: %.3f, average fitness of parents: %.3f\n' % "
...             "(gen, alleleFreq[0][1], meanOfInfo['avgFitness'])", step=10)
...     ],
...     gen = 50,
... )
gen 0: allele freq: 0.473, average fitness of parents: 0.984
gen 10: allele freq: 0.421, average fitness of parents: 0.986
gen 20: allele freq: 0.388, average fitness of parents: 0.988
gen 30: allele freq: 0.288, average fitness of parents: 0.991
gen 40: allele freq: 0.256, average fitness of parents: 0.993
50L
```

4.10.3 Tracking parents (operator `ParentsTagger`)

A parents tagger is used to record the indexes of parents (in the parental population) in the information fields (default to `father_idx`, `mother_idx`) of their offspring. These indexes provide a way to track down an individuals parents, offspring and consequently all relatives in a multi-generation population. Because this operator has been extensively used in this guide, please refer to other sections for an Example (e.g. Example 3.4).

As long as parental generations do not change after the offspring generation is created, recorded parental indexes can be used to locate parents of an individual. However, in certain applications when parental generations change (e.g. to draw a pedigree from a large population), or when individuals can not be looked up easily using indexes (e.g. after individuals are saved to a file), giving every Individual an unique ID and refer to them using ID will be a better choice.

4.10.4 Tracking index of offspring within families (operator `OffspringTagger`)

An offspring tagger is used to record the index of offspring within each family in an information field (default to `offspring_idx`) of offspring. Because the index is reset for each mating event, the index will be reset even if two adjacent families share the same parents. In addition, this operator records the relative index of an offspring so the index will not change if an offspring is re-generated when the previous offspring is discarded for any reason.

Because during-mating selection operator discards offspring according their genotypes, a mating scheme can produce families with varying sizes even if `numOffspring` is set to a constant number. On the other hand, if we would like to ensure equal family size N in the presence of natural selection, we will have to produce more offspring so that there can be at least N offspring in each family after selection. Once N offspring have been generated, excessive offspring can be discarded according to `offspring_idx`. The following example demonstrates such a simulation scenario:

Listing 4.64: Keeping constant family size in the presence of natural selection against offspring

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=1, infoFields='offspring_idx')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
```

```
...         # lethal recessive alleles
...         sim.MaSelector(loci=0, wildtype=0, fitness=[1, 0.90, 0.5]),
...         sim.OffspringTagger(),
...         sim.DiscardIf('offspring_idx > 4'),
...     ], numOffspring=10),
...     postOps=[
...         sim.Stat(alleleFreq=0, step=10),
...         sim.PyEval(r"'gen %d: allele freq: %.3f\n' % "
...             "(gen, alleleFreq[0][1])", step=10)
...     ],
...     gen = 50,
... )
gen 0: allele freq: 0.445
gen 10: allele freq: 0.187
gen 20: allele freq: 0.089
gen 30: allele freq: 0.087
gen 40: allele freq: 0.059
50L
```

Because families with lethal alleles produce the same number of offspring as families without such alleles, natural selection happens within each families and is weaker than the case when natural selection is used to all offspring. This phenomena is generally referred to as reproductive compensation.

4.10.5 Assign unique IDs to individuals (operator `IdTagger`)

Although it is possible to use generation number and individual indexes to locate individuals in an evolving population, an unique I D makes it much easier to identify individuals when migration is involved, and to analyze an evolutionary process outside of simuPOP. An operator `IdTagger` (and its function form `tagID`) is provided by simuPOP to assign an unique ID to all individuals during evolution.

The IDs of individuals are usually stored in an information field named `ind_id`. To ensure uniqueness across populations, a single source of ID is used for this operator. individual IDs are assigned consecutively starting from 0. If you would like to reset the sequence or start from a different number, you can call the `reset(startID)` function of any `IdTagger`.

An `IdTagger` is usually used during-mating to assign ID to each offspring. However, if it is applied directly to a population, it will assign unique IDs to all individuals in this population. This property is usually used in the `preOps` parameter of function `Simulator.evolve` to assign initial ID to a population. For example, two `IdTagger` operators are used in Example 4.65 to assign IDs to all individuals. Although different operators are used, different IDs are assigned to individuals.

Listing 4.65: Assign unique IDs to individuals

```
>>> import simuPOP as sim
>>> pop = sim.Population(10, infoFields='ind_id', ancGen=1)
>>> pop.evolve(
...     initOps=sim.IdTagger(),
...     matingScheme=sim.RandomSelection(ops=[
...         sim.CloneGenoTransmitter(),
...         sim.IdTagger(),
...     ]),
...     gen = 1
... )
1L
>>> print([int(ind.ind_id) for ind in pop.individuals()])
```

```
[11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> pop.useAncestralGen(1)
>>> print([int(ind.ind_id) for ind in pop.individuals()])
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> sim.tagID(pop) # re-assign ID
>>> print([int(ind.ind_id) for ind in pop.individuals()])
[21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
```

4.10.6 Tracking Pedigrees (operator `PedigreeTagger`)

A `PedigreeTagger` is similar to a `ParentsTagger` in that it records parental information in offspring's information fields. However, instead of indexes of parents, this operator records an unique ID of each parent to make it easier to study and reconstruct a complete pedigree of a whole evolutionary process. The default information fields are `father_id` and `mother_id`.

By default, the `PedigreeTagger` does not produce any output. However, if a valid output string (or function) is specified, it will output the ID of offspring and their parents, sex and affection status of offspring, and optionally values at specified information fields (parameter `outputFields`) and genotype at specified loci (parameter `outputLoci`). Because this operator only outputs offspring, the saved file does not have detailed information of individuals in the top-most ancestral generation. If you would like to record complete pedigree information, you can apply `PedigreeTagger` in the `initOps` operator of function `Simulator.evolve` or `Population.evolve` to output information of the initial population. Although this operator is primarily used to output pedigree information, values at specified information fields and genotypes at specified loci could also be outputed.

Example 4.66 demonstrates how to output the complete pedigree of an evolutionary process. Note that `IdTagger` has to be applied before `PedigreeTagger` so that IDs of offspring could be assigned before they are outputted.

Listing 4.66: Output a complete pedigree of an evolutionary process

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, infoFields=['ind_id', 'father_id', 'mother_id'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.IdTagger(),
...         sim.PedigreeTagger(output='>>pedigree.txt'),
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.IdTagger(),
...         sim.PedigreeTagger(output='>>pedigree.txt'),
...         sim.MendelianGenoTransmitter()]
...     ),
...     gen = 100
... )
100L
>>> ped = open('pedigree.txt')
>>> lines = ped.readlines()
>>> ped.close()
>>> # first few lines, saved by the first PedigreeTagger
>>> print(''.join(lines[:3]))
1 0 0 F U
2 0 0 F U
3 0 0 M U

>>> # last several lines, saved by the second PedigreeTagger
```

```
>>> print(''.join(lines[-3:]))
10098 9974 9915 F U
10099 9967 9997 M U
10100 9945 9936 M U

>>> # load this file
>>> ped = sim.loadPedigree('pedigree.txt')
>>> # should have 100 ancestral generations (plus one present generation)
>>> ped.ancestralGens()
100
```

4.10.7 A hybrid tagger (operator `PyTagger`)

A `PyTagger` uses a user-defined function to pass parental information fields to offspring. When a mating event happens, this operator collect values of specified information fields of parents, pass them to a user-provided function, and use the return values to set corresponding offspring information fields. A typical usage of this operator is to set random environmental factors that are affected by parental values. Example 4.67 demonstrates such an example where the location of each offspring (x, y) is randomly assigned around the middle position of his or her parents.

Listing 4.67: Use of a hybrid tagger to pass parental information to offspring

```
>>> import simuPOP as sim
>>> import random
>>> def randomMove(x, y):
...     '''Pass parental information fields to offspring'''
...     # shift right with high concentration of alleles...
...     off_x = random.normalvariate((x[0]+x[1])/2., 0.1)
...     off_y = random.normalvariate((y[0]+y[1])/2., 0.1)
...     return off_x, off_y
...
>>> pop = sim.Population(1000, loci=[1], infoFields=['x', 'y'])
>>> pop.setVirtualSplitter(sim.GenotypeSplitter(loci=0, alleles=[[0, 0], [0,1], [1, 1]]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         sim.InitInfo(random.random, infoFields=['x', 'y'])
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.PyTagger(func=randomMove),
...     ]),
...     postOps=[
...         sim.Stat(minOfInfo='x', maxOfInfo='x'),
...         sim.PyEval(r"'Range of x: %.2f, %.2f\n' % (minOfInfo['x'], maxOfInfo['x'])")
...     ],
...     gen = 5
... )
Range of x: -0.14, 1.18
Range of x: -0.03, 1.03
Range of x: -0.00, 1.03
Range of x: 0.04, 0.98
Range of x: 0.03, 0.94
5L
```

```
>>>
```

4.10.8 Tagging that involves other parental information

If the way how parental information fields pass to their offspring is affected by parental genotype, sex, or affection status, you could use a Python operator (`PyOperator`) during mating to explicitly obtain parental information and set offspring information fields.

Alternatively, you can add another information field, translate needed information to this field and pass the genotype information in the form of information field. Operator `InfoExec` could be helpful in this case. Example 4.68 demonstrates such an example where the number of affected parents are recorded in an information field. Before mating happens, a penetrance operator is used to assign affection status to parents. The affection status is then copied to an information field affected so that operator `SummaryTagger` could be used to count the number of affected parents. Two `MaPenetrance` operators are used both before and after mating to assign affection status to both parental and offspring generations. This helps dividing the offspring generation into affected and unaffected virtual subpopulations. Not surprisingly, the average number of affected parents is larger for affected individuals than unaffected individuals.

Listing 4.68: Tagging that involves other parental information

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=[1], infoFields=['aff', 'numOfAff'])
>>> # define virtual subpopulations by affection sim.status
>>> pop.setVirtualSplitter(sim.AffectionSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...     ],
...     preOps=[
...         # get affection sim.status for parents
...         sim.MaPenetrance(loci=0, wildtype=0, penetrance=[0.1, 0.2, 0.4]),
...         # set 'aff' of parents
...         sim.InfoExec('aff = ind.affected()', exposeInd='ind'),
...     ],
...         # get number of affected parents for each offspring and store in numOfAff
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.SummaryTagger(mode=sim.SUMMATION, infoFields=['aff', 'numOfAff'])]),
...     postOps=[
...         # get affection sim.status for offspring
...         sim.MaPenetrance(loci=0, wildtype=0, penetrance=[0.1, 0.2, 0.4]),
...         # calculate mean 'numOfAff' of offspring, for unaffected and affected subpopulations.
...         sim.Stat(meanOfInfo='numOfAff', subPops=[(0,0), (0,1)], vars=['meanOfInfo_sp']),
...         # print mean number of affected parents for unaffected and affected offspring.
...         sim.PyEval(r"'Mean number of affected parents: %.2f (unaff), %.2f (aff)\n' % "
...             "(subPop[(0,0)]['meanOfInfo']['numOfAff'], subPop[(0,1)]['meanOfInfo']['numOfAff'])")
...     ],
...     gen = 5
... )
Mean number of affected parents: 0.41 (unaff), 0.44 (aff)
Mean number of affected parents: 0.41 (unaff), 0.54 (aff)
Mean number of affected parents: 0.47 (unaff), 0.55 (aff)
Mean number of affected parents: 0.47 (unaff), 0.55 (aff)
Mean number of affected parents: 0.42 (unaff), 0.45 (aff)
```

```
5L
>>>
```

4.11 Statistics calculation (operator `Stat`)

4.11.1 How statistics calculation works

A `Stat` operator calculates specified statistics of a population when it is applied to this population. This operator can be applied to specified replicates (parameter *rep*) at specified generations (parameter *begin*, *end*, *step*, and *at*). This operator does not produce any output (ignore parameter *output*) after statistics are calculated. Instead, it stores results in the local namespace of the population being applied. Other operators can retrieve these variables or evalulate expression directly in this local namespace.

The `Stat` operator is usually used in conjunction with a `PyEval` or `PyExec` operator which execute Python statements and/or expressions in a population's local namespace. For example, operators

```
ops = [
    Stat(alleleFreq=[0]),
    PyEval("'%.2f' % alleleFreq[0][0]")
]
```

in the `ops` parameter of the `Simulator.evolve` function will be applied to populations during evolution. The first operator calculates allele frequency at the first locus and store the results in each population's local namespace. The second operator formats and outputs one of the variables. Because of the flexiblity of the `PyEval` operator, you can output statistics, even simple derived statistics, in any format. For example, you can output expected heterozygosity ($1-\sum p_i^2$) using calculated allele frequencies as follows:

```
PyEval("'H_exp=%.2f' % (1-sum([x*x for x in alleleFreq[0].values()]))")
```

Note that `alleleFreq[0]` is a dictionary.

You can also retrieve variables in a population directly using functions `Population.vars()` or `Population.dvars()`. The only difference between these functions is that `vars()` returns a dictionary and `dvars()` returns a Python object that uses variable names as attributes (`vars()['alleleFreq']` is equivalent to `dvars().alleleFreq`). This method is usually used when the function form of the `Stat` operator is used. For example,

```
stat(pop, alleleFreq=[0])
H_exp = 1 - sum([x*x for x in pop.dvars().alleleFreq[0].values()])
```

uses the `stat` function (note the capital S) to count frequencies of alleles for a given population and calculates expected heterozygosity using these variables.

4.11.2 `defdict` datatype

simuPOP uses dictionaries to save statistics such as allele frequencies. For example, `alleleFreq[5]` can be `{0:0.2, 3:0.8}` meaning there are 20% allele 0 and 80% allele 3 at locus 5 in a population. However, because it is sometimes unclear whether or not a particular allele exists in a population, `alleleFreq[5][allele]` can fail with a `KeyError` exception if `alleleFreq[5]` does not have key `allele`.

To address this problem, a special default dictionary type `defdict` is used for dictionaries with keys determined from a population. This derived dictionary type works just like a regular dictionay, but it returns 0, instead of raising a `KeyError` exception, when an invalid key is used. For example, subpopulations in Example 4.69 have different alleles. Although `pop.dvars(sp).alleleFreq[0]` have only two keys for `sp=0` or `1`, `pop.dvars(sp).alleleFreq[0][x]` are used to print frequencies of alleles `0`, `1` and `2`.

Listing 4.69: The defdict datatype

```
>>> import simuPOP as sim
>>> pop = sim.Population([100]*2, loci=1)
>>> sim.initGenotype(pop, freq=[0, 0.2, 0.8], subPops=0)
>>> sim.initGenotype(pop, freq=[0.2, 0.8], subPops=1)
>>> sim.stat(pop, alleleFreq=0, vars=['alleleFreq_sp'])
>>> for sp in range(2):
...     print('Subpop %d (with %d alleles): ' % (sp, len(pop.dvars(sp).alleleFreq[0])))
...     for a in range(3):
...         print('%.2f ' % pop.dvars(sp).alleleFreq[0][a])
...
Subpop 0 (with 2 alleles):
0.00
0.21
0.79
Subpop 1 (with 2 alleles):
0.21
0.79
0.00
```

Note: The standard `collections` module of Python has a `defaultdict` type that accepts a default factory function that will be used when an invalid key is encountered. The `defdict` type is similar to `defaultdict(int)` but with an important difference: when an invalid key is encountered, `d[key]` with a default value will be inserted to a `defaultdict(int)`, but will not be inserted to a `defdict`. That is to say, it is safe to use `alleleFreq[loc].keys()` to get available alleles after non-assignment `alleleFreq[loc][allele]` operations.

4.11.3 Support for virtual subpopulations

The `Stat` operator supports parameter *subPops* and can calculate statistics in specified subpopulations. For example

```
Stat(alleleFreq=[0], subPops=[(0, 0), (1, 0)])
```

will calculate the frequencies of alleles at locus 0, among Individuals in two virtual subpopulations. If the virtual subpopulation is defined by sex (using a `SexSplitter`), the above operator will calculate allele frequency among all males in the first and second subpopulations (not separately!). If `subPops` is not specified, allele frequency of the whole population (all subpopulations) will be calculated.

Although many statistics could be calculated and outputted, the `Stat` operator by default outputs a selected number of variables for each statisic calculated. Other statistics could be calculated and outputted if their names are specified in parameter `vars`. Variable names ending with `_sp` is interpreted as variables that will be calculated and outputted in all or specified (virtual) subpopulations. For example, parameter `vars` in

```
Stat(alleleFreq=[0], subPops=[0, (1, 0)], vars=['alleleFreq_sp', 'alleleNum_sp'])
```

tells this operator to output numbers and frequencies of alleles at locus `0` in subpopulation `0` and virtual subpopulation `(1,0)`. These variables will be saved in dictionaries `subPop[sp]` of the local namespace. For example, the above operator will write variables such as `subPop[0]['alleleFreq'], subPop[(1,0)]['alleleFreq'] and subPop[(1,0)]['alleleNum']`. Functions `Population.vars(sp)` and `Population.dvars(sp)` are provided as shortcuts to access these variables but the full variable names have to be specified if these variables are used in expressions.

By default, the same variables will be set for a statistic, regardless of the values of the `loci` and `subPops` parameter. This can be a problem if multiple `Stat` operators are used to calculate the same statistics for different sets of loci (e.g. for each chromosome) or subpopulations. To avoid name conflict, you can use parameter *suffix* to add a suffix to all variables outputted by a Stat operator. For example, Example 4.70 uses 4 `Stat` operators to calculate overall and pairwise F_{ST} values for three subpopulations. Different suffixes are used for pairwise F_{ST} estimators so that variables set by these operators will not override each other.

Listing 4.70: Add suffixes to variables set by multiple Stat operators

```
>>> import simuPOP as sim
>>> pop = sim.Population([5000]*3, loci=5)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(structure=range(5), subPops=(0, 1), suffix='_01', step=40),
...         sim.Stat(structure=range(5), subPops=(1, 2), suffix='_12', step=40),
...         sim.Stat(structure=range(5), subPops=(0, 2), suffix='_02', step=40),
...         sim.Stat(structure=range(5), step=40),
...         sim.PyEval(r"'Fst=%.3f (pairwise: %.3f %.3f %.3f)\n' % (F_st, F_st_01, F_st_12, F_st_02)",
...             step=40),
...     ],
...     gen = 200
... )
Fst=0.000 (pairwise: 0.000 0.000 0.000)
Fst=0.004 (pairwise: 0.006 0.003 0.004)
Fst=0.012 (pairwise: 0.017 0.015 0.004)
Fst=0.008 (pairwise: 0.012 0.010 0.001)
Fst=0.008 (pairwise: 0.007 0.009 0.007)
200L
```

Note: The `Stat` opeartor accepts overlapping or even duplicate virtual subpopulations. During the calculation of summary statistics, these subpopulations are treated as separate subpopulations so some individuals can be counted more than once. For example, individuals in virtual subpopulation (0, 1) will be counted twice during the calculation of allele frequency and population size in operator

```
Stat(alleleFreq=[0], popSize=True, subPops=[0, (0, 1)])
```

4.11.4 Counting individuals by sex and affection status

Parameters *popSize*, *numOfMales* and *numOfAffected* provide basic Individual counting statistics. They count the number of all, male/female, affected/unaffected individuals in all or specified (virtual) subpopulations, and set variables such as `popSize`, `numOfMales`, `numOfFemales`, `numOfAffected`, `numOfUnaffected`. Proportions and statistics for subpopulations are available if variables such as `propOfMales`, `numOfAffected_sp` are specified in parameter vars. Another variable `subPopSize` is defined for parameter `popSize=True`. It is a list of sizes of all or specified subpopulations and is easier to use than referring to variable `popSize` from individual subpopulations.

Example 4.71 demonstrates how to use these parameters in operator `Stat`. It defines four VSPs by sex and affection status (using a `stackedSplitter`) and count individuals by sex and affection status. It is worth noting that `pop.dvars().popSize` in the first example is the total number of individuals in two virtual subpopulations `(0,0)` and `(0,2)`, which are all male indiviudals, and all unaffected individuals. Because these two VSPs overlap, this variable can be larger than actual population size.

Listing 4.71: Count individuals by sex and/or affection status

```
>>> import simuPOP as sim
>>> pop = sim.Population(10000, loci=1)
>>> pop.setVirtualSplitter(sim.CombinedSplitter(
...     [sim.SexSplitter(), sim.AffectionSplitter()]))
>>> sim.initSex(pop)
```

```
>>> sim.initGenotype(pop, freq=[0.2, 0.8])
>>> sim.maPenetrance(pop, loci=0, penetrance=[0.1, 0.2, 0.5])
>>> # Count sim.population size
>>> sim.stat(pop, popSize=True, subPops=[(0, 0), (0, 2)])
>>> # popSize is the size of two VSPs, does not equal to total sim.population size.
>>> # Because two VSPs overlap (all males and all unaffected), popSize can be
>>> # greater than real sim.population size.
>>> print(pop.dvars().subPopSize, pop.dvars().popSize)
([5052, 6080], 11132)
>>> # print popSize of each virtual subpopulation.
>>> sim.stat(pop, popSize=True, subPops=[(0, 0), (0, 2)], vars='popSize_sp')
>>> # Note the two ways to access variable in (virtual) subpopulations.
>>> print(pop.dvars((0,0)).popSize, pop.dvars().subPop[(0,2)]['popSize'])
(5052, 6080)
>>> # Count number of male (should be the same as the size of VSP (0,0).
>>> sim.stat(pop, numOfMales=True)
>>> print(pop.dvars().numOfMales)
5052
>>> # Count the number of affected and unaffected male individual
>>> sim.stat(pop, numOfMales=True, subPops=[(0, 2), (0, 3)], vars='numOfMales_sp')
>>> print(pop.dvars((0,2)).numOfMales, pop.dvars((0,3)).numOfMales)
(3056, 1996)
>>> # or number of affected male and females
>>> sim.stat(pop, numOfAffected=True, subPops=[(0, 0), (0, 1)], vars='numOfAffected_sp')
>>> print(pop.dvars((0,0)).numOfAffected, pop.dvars((0,1)).numOfAffected)
(1996, 1924)
>>> # These can also be done using a sim.ProductSplitter...
>>> pop.setVirtualSplitter(sim.ProductSplitter(
...     [sim.SexSplitter(), sim.AffectionSplitter()]))
>>> sim.stat(pop, popSize=True, subPops=[(0, x) for x in range(4)])
>>> # counts for male unaffected, male affected, female unaffected and female affected
>>> print(pop.dvars().subPopSize)
[3056, 1996, 3024, 1924]
```

4.11.5 Number of segregating and fixed sites

Parameter *numOfSegSites* counts the number of segregating sites for specified or all loci, for all individuals or individuals in specified (virtual) subpopulations. It can also be used to count the number of fixed sites . This parameter sets variables `numOfSegSites` and `numOfFixedSites`. Here we defined fixed sites as loci with only one non-zero allele (e.g. fixed to a non-zero allele). Other numbers, such as all loci with only one allele (including zero), or loci with all wildtype alleles (only zero), can be derived from these two counts. Starting from version 1.1.3, variables `segSites` and `fixedSites` can be used to return a list of segregating and fixed sites.

For example, Example 4.72 demonstrates how to use this operator to calculate the number of segregating sites (sites with alleles 0 and 1), number of fixed sites (sites with only allele 1), and number of loci with only wildtype alleles (loci with only allele 0). As you can see, the population starts with 100 segregating sites. During evolution, alleles at some loci get lost and some get fixed, and there should be no segregating site if we evolve the population for long enough.

Listing 4.72: Count number of segregating and fixed sites

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=[1]*100)
>>> pop.evolve(
```

```
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.3, 0.7]),
...         sim.PyOutput('#all 0\t#seg sites\t#all 1\n'),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(numOfSegSites=sim.ALL_AVAIL,
...             vars=['numOfSegSites', 'numOfFixedSites']),
...         sim.PyEval(r'"%d\t%d\t%d\n" % (100-numOfSegSites-numOfFixedSites,'
...             'numOfSegSites, numOfFixedSites)',
...             step=50)
...         ],
...     gen=500
... )
#all 0  #seg sites      #all 1
0       100     0
0       93      7
3       76      21
7       55      38
12      40      48
17      31      52
19      23      58
22      19      59
26      14      60
28      10      62
500L
>>> # output a list of segregating sites
>>> sim.stat(pop, numOfSegSites=sim.ALL_AVAIL, vars='segSites')
>>> print(pop.dvars().segSites)
[11, 15, 20, 32, 39, 43, 44, 51, 86, 95]
```

4.11.6 Allele count and frequency

Parameter *alleleFreq* accepts a list of markers at which allele frequencies in all or specified (virtual) subpopulations will be calculated. This statistic sets variables `alleleFreq[loc][allele]` and `alleleNum[loc][allele]` which are frequencies and numbers of allele `allele` at locus `loc`, respectively. If variables `alleleFreq_sp` and `alleleNum_sp` are specified in parameter *vars*, these variables will be set for all or specified (virtual) subpopulations. **At the Python level, these variables are dictionaries of default dictionaries.** That is to say, `alleleFreq[loc]` at a unspecified locus will raise a `KeyError` exception, and `alleleFreq[loc][allele]` of an invalid allele will return 0.

Example 4.73 demonstrates an advanced usage of allele counting statistic. In this example, two virtual subpopulations are defined by individual affection status. During evolution, a multi-allele penetrance operator is used to determine individual affection status and a `Stat` operator is used to calculate allele frequencies in these two virtual subpopulations, and in the whole population. Because the simulated disease is largely caused by the existence of allele 1 at the first locus, it is expected that the frequency of allele 1 is higher in the case group than in the control group. It is worth noting that `alleleFreq[0][1]` in this example is the frequency of allele 1 in the whole population because these two virtual subpopulations add up to the whole population.

Listing 4.73: Calculate allele frequency in affected and unaffected individuals

```
>>> import simuPOP as sim
>>> pop = sim.Population(10000, loci=1)
>>> pop.setVirtualSplitter(sim.AffectionSplitter())
```

```
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(loci=0, freq=[0.8, 0.2])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.MaPenetrance(penetrance=[0.1, 0.4, 0.6], loci=0),
...         sim.Stat(alleleFreq=0, subPops=[(0, 0), (0, 1)],
...             vars=['alleleFreq', 'alleleFreq_sp']),
...         sim.PyEval(r"'Gen: %d, freq: %.2f, freq (aff): %.2f, freq (unaff): %.2f\n' % " + \
...             "(gen, alleleFreq[0][1], subPop[(0,1)]['alleleFreq'][0][1]," + \
...             "subPop[(0,0)]['alleleFreq'][0][1])"),
...     ],
...     gen = 5
... )
Gen: 0, freq: 0.20, freq (aff): 0.41, freq (unaff): 0.14
Gen: 1, freq: 0.20, freq (aff): 0.40, freq (unaff): 0.14
Gen: 2, freq: 0.20, freq (aff): 0.41, freq (unaff): 0.14
Gen: 3, freq: 0.20, freq (aff): 0.41, freq (unaff): 0.14
Gen: 4, freq: 0.19, freq (aff): 0.41, freq (unaff): 0.14
5L
```

4.11.7 Genotype count and frequency

Parameter *genoFreq* accepts a list of loci at which genotype counts and frequencies are calculated and outputted. A genotype is represented as a tuple of alleles at a locus. The length of the tupples is determined by the number of homologous copy of chromosomes in a population. For example, genotypes in a diploid population are ordered pairs such as `(1, 2)` where 1 and 2 are alleles at a locus on, respectively, the first and second homologous copies of chromosomes. `(1, 2)` and `(2, 1)` are different genotypes. This statistic sets dictionaries (with locus indexes as keys) of default dictionaries (with genotypes as keys) `genoFreq` and `genoNum`.

Example 4.74 creates a small population and initializes a locus with rare alleles 0, 1 and a common allele 2. A function `stat` (the function form of operator `Stat`) is used to count the available genotypes. Note that `pop.dvars().genoFreq[0][(i,j)]` can be used to print frequencies of all genotypes even when not all genotypes are available in the population.

Listing 4.74: Counting genotypes in a population

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=[1, 1, 1], lociNames=['A', 'X', 'Y'],
...     chromTypes=[sim.AUTOSOME, sim.CHROMOSOME_X, sim.CHROMOSOME_Y])
>>> sim.initGenotype(pop, freq=[0.01, 0.05, 0.94])
>>> sim.stat(pop, genoFreq=['A', 'X']) # both loci indexes and names can be used.
>>> print('Available genotypes on autosome:', list(pop.dvars().genoFreq[0].keys()))
('Available genotypes on autosome:', [(1, 2), (2, 1), (1, 1), (2, 0), (2, 2), (0, 2)])
>>> for i in range(3):
...     for j in range(3):
...         print('%d-%d: %.3f' % (i, j, pop.dvars().genoFreq[0][(i,j)]))
...
0-0: 0.000
0-1: 0.000
0-2: 0.020
1-0: 0.000
```

```
1-1: 0.030
1-2: 0.070
2-0: 0.010
2-1: 0.040
2-2: 0.830
>>> print('Genotype frequency on chromosome X:\n', \
...     '\n'.join(['%s: %.3f' % (x,y) for x,y in pop.dvars().genoFreq[1].items()]))
('Genotype frequency on chromosome X:\n', '(2,): 0.950\n(0,): 0.020\n(1,): 0.030')
```

4.11.8 Homozygote and heterozygote count and frequency

In a diploid population, a heterozygote is a genotype with two different alleles and a homozygote is a genotype with two identical alleles. Parameter `heteroFreq` accepts a list of loci and outputs variables `heteroFreq` which is a dictionary of heterozygote frequencies at specfied loci. Optional variables `heteroNum`, `homoFreq` and `homoNum` can be outputted for all and each (virtual) subpopulations. Example 4.75 demonstrates the decay of heterozygosity of a locus due to genetic drift.

Listing 4.75: Counting homozygotes and heterozygotes in a population

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=1)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(heteroFreq=0, step=10),
...         sim.PyEval(r"'Gen: %d, HeteroFreq: %.2f\n' % (gen, heteroFreq[0])", step=20)
...     ],
...     gen = 100
... )
Gen: 0, HeteroFreq: 0.45
Gen: 20, HeteroFreq: 0.44
Gen: 40, HeteroFreq: 0.55
Gen: 60, HeteroFreq: 0.46
Gen: 80, HeteroFreq: 0.40
100L
```

4.11.9 Haplotype count and frequency

Haplotypes refer to alleles on the same homologous copy of a chromosome at specified loci. For example, an diploid individual can have haplotypes `(0, 2, 1)` and `(0, 1, 1)` at loci `(2, 3, 5)` if he or she has genotype `(0, 0)`, `(2, 1)` and `(1,1)` at loci 2, 3 and 5 respectively. Parameter *haploFreq* accept one or more lists of loci specifying one or more haplotype sites (e.g. `haploFreq=[(0,1,2), (2,3)]` specifies two haplotype sites). The results are saved to dictionaries (with haplotype site as keys) of default dictionaries (with haplotype as keys). For example, `haploFreq[(0,1,2)][(0,1,1)]` will be the frequency of haplotype `(0, 1, 1)` at loci `(0, 1, 2)`. Example 4.76 prints the numbers of genotypes and haplotypes at loci 0, 1 and 2 of a small population. Note that the `viewVars` function defined in module `simuUtil` can make use of a wxPython window to view all variables if it is called in GUI mode.

Listing 4.76: Counting haplotypes in a population

```
>>> import simuPOP as sim
>>> from simuPOP.utils import viewVars
>>> pop = sim.Population(100, loci=3)
>>> sim.initGenotype(pop, freq=[0.2, 0.4, 0.4], loci=0)
>>> sim.initGenotype(pop, freq=[0.2, 0.8], loci=2)
>>> sim.stat(pop, genoFreq=[0, 1, 2], haploFreq=[0, 1, 2],
...     vars=['genoNum', 'haploFreq'])
>>> viewVars(pop.vars(), gui=False)
{'genoNum': {0: {(0, 0): 3.0,
                 (0, 1): 7.0,
                 (0, 2): 5.0,
                 (1, 0): 9.0,
                 (1, 1): 14.0,
                 (1, 2): 16.0,
                 (2, 0): 8.0,
                 (2, 1): 14.0,
                 (2, 2): 24.0},
             1: {(0, 0): 100.0},
             2: {(0, 0): 4.0, (0, 1): 19.0, (1, 0): 15.0, (1, 1): 62.0}},
 'haploFreq': {(0, 1, 2): {(0, 0, 0): 0.03,
                           (0, 0, 1): 0.145,
                           (1, 0, 0): 0.055,
                           (1, 0, 1): 0.315,
                           (2, 0, 0): 0.125,
                           (2, 0, 1): 0.33}}}
```

Note: *haploFreq* does not check if loci in a haplotype site belong to the same chromosome, or if loci are duplicated or in order. It faithfully assemble alleles at specified loci as haplotypes although these haplotypes might not be biologically meaningful.

Note: Counting a large number of haplotypes on long haplotype sites may exhaust the RAM of your computer.

4.11.10 Summary statistics of information fields

Parameter `sumOfInfo`, `meanOfInfo`, `varOfInfo`, `maxOfInfo` and `minOfInfo` are used to calculate the sum, mean, sample variance ($\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2$), max and min of specified information fields of individuals in all or specified (virtual) subpopulations. The results are saved in dictionaries `sumOfInfo`, `meanOfInfo`, `varOfInfo`, `maxOfInfo` and `minOfInfo` with information fields as keys. For example, parameter `meanOfInfo='age'` calculates the mean age of all individuals and set variable `meanOfInfo['age']`.

Example 4.77 demonstrates a mixing process of two populations. The population starts with two types of individuals with ancestry values 0 or 1 (information field `anc`). During the evolution, parents mate randomly and the ancestry of offspring is the mean of parental ancestry values. A `Stat` operator is used to calculate the mean and variance of individual ancestry values, and the number of individuals in five ancestry groups. It is not surprising that whereas population mean ancestry does not change, more and more people have about the same number of ancestors from each group and have an ancestry value around 0.5. The variance of ancestry values therefore decreases gradually.

Listing 4.77: Calculate summary statistics of information fields

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population([500], infoFields='anc')
>>> # Defines VSP 0, 1, 2, 3, 4 by anc.
>>> pop.setVirtualSplitter(sim.InfoSplitter('anc', cutoff=[0.2, 0.4, 0.6, 0.8]))
>>> #
```

```
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         # anc is 0 or 1
...         sim.InitInfo(lambda : random.randint(0, 1), infoFields='anc')
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.InheritTagger(mode=sim.MEAN, infoFields='anc')
...     ]),
...     postOps=[
...         sim.Stat(popSize=True, meanOfInfo='anc', varOfInfo='anc',
...             subPops=[(0, sim.ALL_AVAIL)]),
...         sim.PyEval(r"'Anc: %.2f (%.2f), #inds: %s\n' %" + \
...             "(meanOfInfo['anc'], varOfInfo['anc'], " + \
...             "', '.join(['%4d' % x for x in subPopSize]))")
...     ],
...     gen = 5,
... )
Anc: 0.54 (0.12), #inds:  106,    0,  253,    0,  141
Anc: 0.54 (0.06), #inds:   23,   97,  197,  137,   46
Anc: 0.55 (0.03), #inds:    7,  125,  128,  204,   36
Anc: 0.56 (0.02), #inds:    1,   55,  247,  178,   19
Anc: 0.55 (0.01), #inds:    0,   13,  354,  131,    2
5L
```

4.11.11 Linkage disequilibrium

Parameter *LD* accepts a list of loci-pairs (e.g. `LD=[(0,1),(2,3)]`) with optional primary alleles at two loci (e.g. `LD=[(0,1,0,0),(2,3)]`). For each pair of loci, this operator calculates linkage disequilibrium and optional association measures between them.

Assuming that two loci are both diallelic, one with alleles A and a, and the other with alleles B and b. If we denote P_x, P_{xy} as allele and haplotype frequencies for allele x and haplotype xy, respectively, the linkage disequilibrium measures **with respect to primaries alleles** *A* and *B* are

- Basic LD measure D:

$$D = P_{AB} - P_A P_B$$

  D ranges from -0.25 to 0.25. The sign depends on the choice of alleles (*A* and *B*) at two loci.

- Lewontin's $D' = D/D_{max}$ where

$$D_{max} = \begin{cases} \min\left(P_A\left(1-P_B\right), \left(1-P_A\right)P_B\right) & \text{if } D > 0 \\ \min\left(P_A P_B, \left(1-P_A\right)\left(1-P_B\right)\right) & \text{if } D < 0 \end{cases}$$

  D' ranges from -1 to 1. The sign depends on the choice of alleles (*A* and *B*) at two loci.

- r^2 (Δ^2 in Devlin and Risch [1995])

$$r^2 = \frac{D^2}{P_A\left(1-P_A\right)P_B\left(1-P_B\right)}$$

If one or both loci have more than 2 alleles, or if no primary allele is specified, the LD measures are calculated as follows:

- If primary alleles are specified, all other alleles are considered as minor alleles with combined frequency (e.g. $1 - P_A$). The same formulas apply which lead to signed D and D' measures.
- If primary alleles are not specified, these LD measures are calculated as the average of the absolute value of diallelic measures of all allele pairs. For example, the multi-allele version of r^2 is

$$r^2 = \sum_i \sum_j P_i P_j \left| r_{ij}^2 \right| = \sum_i \sum_j \frac{D_{ij}^2}{(1 - P_i)(1 - P_j)}$$

  where i and j iterate through all alleles at the two loci. **In the diallelic case, LD measures will be the absolute value of the single measures** because D_{ij} and D'_{ij} only differ by signs.

In another word,

- `LD=[loc1, loc2]` will yield positive D and D' measures.
- `LD=[loc1, loc2, allele1, allele2]` will yield signed D and D' measures.
- In the diallelic case, both cases yield identical results except for signs of D and D'.
- In the multi-allelic case, the results can be different because `LD=[loc1, loc2, allele1, allele2]` combines non-primary alleles and gives a single diallelic measure.

Note: A large number of linkage disequilibrium measures have been used in different disciplines but not all of them are well-accepted. Requests of adding a particular LD measure will be considered when a reliable reference is provided. Association tests between specified loci could also be calculated using a m by n table of haplotype frequencies. If primary alleles are specified, non-primary alleles are combined to form a 2 by 2 table ($m = n = 2$). Otherwise, m and n are respective numbers of alleles at two loci.

- χ^2 and its p-value (variable `LD_ChiSq` and `LD_ChiSq_p`, respectively). A one-side χ^2 test with $(m-1) \times (n-1)$ degrees of freedom will be used.
- Cramer V statistic (variable `CramerV`):

$$V = \sqrt{\frac{\chi^2}{N \times \min(m-1, n-1)}}$$

  where N equals the total number of haplotypes ($2 \times$ popSize for autosomes in diploid populations).

This statistic sets variables `LD`, `LD_prime`, `R2`, and optionally `ChiSq`, `ChiSq_p` and `CramerV`. SubPopulation specific variables can be calculated by specifying variables such as `LD_sp` and `R2_sp`. Example 4.78 demonstrates how to calculate various LD measures and output selected variables. Note that the significant overall LD between two loci is an artifact of population structure because loci are in linkage equilibrium in each subpopulation.

Listing 4.78: Linkage disequilibrium measures

```
>>> import simuPOP as sim
>>> pop = sim.Population([1000]*2, loci=3)
>>> sim.initGenotype(pop, freq=[0.2, 0.8], subPops=0)
>>> sim.initGenotype(pop, freq=[0.8, 0.2], subPops=1)
>>> sim.stat(pop, LD=[[0, 1, 0, 0], [1, 2]],
...     vars=['LD', 'LD_prime', 'R2', 'LD_ChiSq', 'LD_ChiSq_p', 'CramerV',
...         'LD_prime_sp', 'LD_ChiSq_p_sp'])
>>> from pprint import pprint
>>> pprint(pop.vars())
{'CramerV': {0: {1: 0.3355834766347789}, 1: {2: 0.39144946095755695}},
```

```
'LD': {0: {1: 0.08387987499999999}, 1: {2: 0.09783043749999992}},
'LD_ChiSq': {0: {1: 450.4650791611408}, 1: {2: 612.9307219358476}},
'LD_ChiSq_p': {0: {1: 0.0}, 1: {2: 0.0}},
'LD_prime': {0: {1: 0.3425347836362625}, 1: {2: 0.4057999832524774}},
'R2': {0: {1: 0.1126162697902852}, 1: {2: 0.15323268048396166}},
'subPop': {0: {'LD_ChiSq_p': {0: {1: 0.03843990070970382},
                              1: {2: 0.5110492462003573}},
               'LD_prime': {0: {1: -0.17661111690962444},
                            1: {2: 0.016760924318107204}}},
           1: {'LD_ChiSq_p': {0: {1: 0.8024214035646771},
                              1: {2: 0.11685510935577492}},
               'LD_prime': {0: {1: -0.02259456714902688},
                            1: {2: 0.035632559660018596}}}}}
```

4.11.12 Genetic association

Genetic association refers to association between individual genotype (alleles or genotype) and phenotype (affection status). There are a large number of statistics tests based on different study designs (e.g. case-control, Pedigree, longitudinal) with different covariate variables. Although specialized software applications should be used for sophisticated statistical analysis, simuPOP provides a number of simple genetic association tests for convenience. These tests

- Are single-locus tests that test specified loci separately.
- Are based on individual affection status. Associations between genotype and quantitative traits are currently unsupported.
- Apply to all individuals in specified (virtual) subpopulations. Because a population usually has much more unaffected individuals than affected ones, it is a common practice to draw certain types of samples (e.g. a case-control sample with the same number of cases and controls) before statistical tests are applied.

simuPOP currently supports the following tests:

- **Allele-based Chi-square test**: This is the basic allele-based χ^2 test that can be applied to diploid as well as haploid populations. Basically, a 2 by n contigency table is set up for each locus with n_{ij} being the number of alleles j in cases $(i = 0)$ and controls $(i = 1)$. A χ^2 test is applied to each locus and set variables `Allele_ChiSq` and `Allele_ChiSq_p` to the χ^2 statistic and its two-sided p value (with degrees freedom $n-1$). Note that genotype information is not preserved in such a test.
- **Genotype-based Chi-square test**: This is the genotype-based χ^2 test for diploid populations. Basically, a 2 by n contigency table is set up for each locus with n_{ij} being the number of genotype j (unordered pairs of alleles) in cases $(i = 0)$ and controls $(i = 1)$. A χ^2 test is applied to each locus and set variables `Geno_ChiSq` and `Geno_ChiSq_p` to the χ^2 statistic and its two-sided p value (with degrees freedom $n - 1$). This test is usually applied to diallelic loci with 3 genotypes (*AA*, *Aa* and *aa*) but it can be applied to loci with more than two alleles as well.
- **Genotype-based trend test**: This Cochran-Armitage test can only be applied to diallelic loci in diploid populations. For each locus, a 2 by 3 contigency table is set up with n_{ij} being the number of genotype j (*AA*, *Aa* and *aa* with *A* being the wildtype allele) in cases $(i = 0)$ and controls $(i = 1)$. A Cochran-Armitage trend test is applied to each locus and set variables `Armitage_p` to its two-sided p value.

Example 4.79 demonstrates how to apply a penetrance model, draw a case-control sample and apply genetic association tests to an evolving population. In this example, a penetrance model is applied to a locus (locus 3). A Python operator is then used to draw a case-control sample from the population and test genetic association at two surrounding loci. Because these two loci are tightly linked to the disease predisposing locus, they are in strong association with the

disease initially. However, because of recombination, such association decays with time at rates depending on their genetic distances to the disease predisposing locus.

Listing 4.79: Genetic association tests

```
>>> import simuPOP as sim
>>> from simuPOP.utils import *
>>> from simuPOP.sampling import drawCaseControlSample
>>> def assoTest(pop):
...     'Draw case-control sample and apply association tests'
...     sample = drawCaseControlSample(pop, cases=500, controls=500)
...     sim.stat(sample, association=(0, 2), vars=['Allele_ChiSq_p', 'Geno_ChiSq_p', 'Armitage_p'])
...     print('Allele test: %.2e, %.2e, Geno test: %.2e, %.2e, Trend test: %.2e, %.2e' \
...         % (sample.dvars().Allele_ChiSq_p[0], sample.dvars().Allele_ChiSq_p[2],
...         sample.dvars().Geno_ChiSq_p[0], sample.dvars().Geno_ChiSq_p[2],
...         sample.dvars().Armitage_p[0], sample.dvars().Armitage_p[2]))
...     return True
...
>>> pop = sim.Population(size=100000, loci=3)
>>> pop.setVirtualSplitter(sim.ProportionSplitter([0.5, 0.5]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[0]*3, subPops=[(0,0)]),
...         sim.InitGenotype(genotype=[1]*3, subPops=[(0,1)]),
...     ],
...     matingScheme=sim.RandomMating(ops=sim.Recombinator(loci=[0, 1], rates=[0.01, 0.005])),
...     postOps=[
...         sim.MaPenetrance(loci=1, penetrance=[0.1, 0.2, 0.4]),
...         sim.PyOperator(func=assoTest, step=20),
...     ],
...     gen = 100
... )
Allele test: 0.00e+00, 0.00e+00, Geno test: 0.00e+00, 0.00e+00, Trend test: 0.00e+00, 0.00e+00
Allele test: 1.14e-13, 4.44e-16, Geno test: 3.09e-13, 2.66e-15, Trend test: 7.66e-14, 2.22e-16
Allele test: 1.71e-08, 8.55e-15, Geno test: 4.95e-08, 3.45e-13, Trend test: 1.62e-08, 7.36e-14
Allele test: 8.57e-09, 7.99e-15, Geno test: 3.09e-08, 2.18e-14, Trend test: 7.05e-09, 2.66e-15
Allele test: 3.12e-06, 9.05e-09, Geno test: 5.95e-06, 8.83e-08, Trend test: 2.12e-06, 1.26e-08
100L
```

4.11.13 population structure

Parameter `structure` measures the structure of a population using the following statistics:

- The G_{ST} statistic developed by Nei [1973]. This statistic is equivalent to Wright's fixation index F_{ST} in the diallelic case so it can be considered as the multi-allele and multi-locus extension of Wright's F_{ST}. It assumes known genotype frequency so it can be used to calculate true F_{ST} of a population when all genotype information is available. This statistic sets a dictionary of locus level G_{ST} (variable `g_st`) and a summary statistics for all loci (variable `G_st`).
- Wright's fixation index F_{ST} calculated using an algorithm developed by Weir and Cockerham [1984]. This statistic considers existing populations as random samples from an infinite pool of populations with the same ancestral population so it is best to be applied to random samples where true genotype frequencies are unknown. This statistic sets dictionaries of locus level F_{ST}, F_{IT} and F_{IS} (variables `f_st`, `f_is` and `f_it`), and summary

statistics for all loci (variables `F_st`, `F_is` and `F_it`) . When hetergozygote count is unavailable (non-diploid population, loci on sex chromosomes and mitochondrial chromosomes), simuPOP uses expected heterozygosity to estimate this quantity.

These statistics by default uses all existing subpopulations, but it can also be applied to a subset of subpopulations, or even virtual subpopulations using parameter *subPops*. That is to say, you can measure the genetic difference between males and females using `subPops=[(0,0), (0,1)]` if a SexSplitter is used to define two virtual subpopulations with male and female individuals respectively.

Example 4.80 demonstrate a simulation with two replicates. In the first replicate, three subpopulations evolve separately without migration and become more and more genetically distinct. In the second replicate, a low level migration is applied between subpopulations so the population structure is kept at a low level.

Listing 4.80: Measure of population structure

```
>>> import simuPOP as sim
>>> from simuPOP.utils import migrIslandRates
>>> simu = sim.Simulator(sim.Population([5000]*3, loci=10, infoFields='migrate_to'),
...     rep=2)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     preOps=sim.Migrator(rate=migrIslandRates(0.01, 3), reps=1),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(structure=range(10), step=40),
...         sim.PyEval("'Fst=%.3f (rep=%d without migration) ' % (F_st, rep)", step=40, reps=0),
...         sim.PyEval("'Fst=%.3f (rep=%d with migration) ' % (F_st, rep)", step=40, reps=1),
...         sim.PyOutput('\n', reps=-1, step=40)
...     ],
...     gen = 200
... )
Fst=0.000 (rep=0 without migration) Fst=0.000 (rep=1 with migration)
Fst=0.003 (rep=0 without migration) Fst=0.002 (rep=1 with migration)
Fst=0.006 (rep=0 without migration) Fst=0.002 (rep=1 with migration)
Fst=0.008 (rep=0 without migration) Fst=0.003 (rep=1 with migration)
Fst=0.010 (rep=0 without migration) Fst=0.001 (rep=1 with migration)
(200L, 200L)
```

4.11.14 Hardy-Weinberg equilibrium test

Parameter `HWE` accepts a list of loci at which exact Hardy Weinberg equilibrium tests are applied. The p-values of the tests are assigned to a dictionary `HWE`. Example 4.81 demonstrates how Hardy Weinberg equilibrium is reached in one generation.

Listing 4.81: Hardy Weinberg Equilibrium test

```
>>> import simuPOP as sim
>>> pop = sim.Population([1000], loci=1)
>>> pop.setVirtualSplitter(sim.ProportionSplitter([0.4, 0.4, 0.2]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
```

```
...         sim.InitGenotype(genotype=[0,0], subPops=[(0,0)]),
...         sim.InitGenotype(genotype=[0,1], subPops=[(0,1)]),
...         sim.InitGenotype(genotype=[1,1], subPops=[(0,2)]),
...     ],
...     preOps=[
...         sim.Stat(HWE=0, genoFreq=0),
...         sim.PyEval(r'"HWE p-value: %.5f (AA: %.2f, Aa: %.2f, aa: %.2f)\n" % (HWE[0], '
...             'genoFreq[0][(0,0)], genoFreq[0][(0,1)] + genoFreq[0][(1,0)], genoFreq[0][(1,1)])'),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(HWE=0, genoFreq=0),
...         sim.PyEval(r'"HWE p-value: %.5f (AA: %.2f, Aa: %.2f, aa: %.2f)\n" % (HWE[0], '
...             'genoFreq[0][(0,0)], genoFreq[0][(0,1)] + genoFreq[0][(1,0)], genoFreq[0][(1,1)])'),
...     ],
...     gen = 1
... )
HWE p-value: 0.00000 (AA: 0.40, Aa: 0.40, aa: 0.20)
HWE p-value: 0.93636 (AA: 0.38, Aa: 0.48, aa: 0.15)
1L
```

4.11.15 Measure of Inbreeding

Inbreeding coefficient at a generation is defined as the probability that the two alleles in a given individual are identical by decent (IBD). Although it is usually very difficult to estimate this quantity, it is easy to observe it directly during evolution if the ancestors of alleles are tracked. This can be done using the lineage module of simuPOP where allelic lineage is tracked during evolution. For example, Example 4.82 output the frequency of IBD loci in a population of size 500. It also outputs the frequency of IBS (Identical by State), which should always be larger than IBD frequency, and theoretical estimate of the decay of inbreeding coefficient.

Listing 4.82: Frequency of IBD as a measure of inbreeding coefficient

```
>>> import simuOpt
>>> simuOpt.setOptions(alleleType='lineage')
>>> import simuPOP as sim
>>> pop = sim.Population([500], loci=[1]*100)
>>> pop.evolve(
...     initOps=[
...         sim.InitLineage(),
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2]*5),
...     ],
...     preOps=[
...         sim.Stat(inbreeding=sim.ALL_AVAIL, popSize=True, step=10),
...         sim.PyEval(r'"gen %d: IBD freq %.4f, IBS freq %.4f, est: %.4f\n" % '
...             '(gen, sum(IBD_freq.values()) /len(IBD_freq), '
...             ' sum(IBS_freq.values()) /len(IBS_freq), '
...             ' 1 - (1-1/(2.*popSize))**gen)', step=10)
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 100
... )
gen 0: IBD freq 0.0000, IBS freq 0.1958, est: 0.0000
gen 10: IBD freq 0.0097, IBS freq 0.2058, est: 0.0100
```

```
gen 20: IBD freq 0.0192, IBS freq 0.2165, est: 0.0198
gen 30: IBD freq 0.0290, IBS freq 0.2203, est: 0.0296
gen 40: IBD freq 0.0383, IBS freq 0.2285, est: 0.0392
gen 50: IBD freq 0.0525, IBS freq 0.2431, est: 0.0488
gen 60: IBD freq 0.0594, IBS freq 0.2503, est: 0.0583
gen 70: IBD freq 0.0706, IBS freq 0.2583, est: 0.0676
gen 80: IBD freq 0.0776, IBS freq 0.2609, est: 0.0769
gen 90: IBD freq 0.0875, IBS freq 0.2658, est: 0.0861
100L
```

4.11.16 Effective population size

Effective population size is an important, yet complicated concept in population genetics. Simply put, the effective population size is determined by a mating scheme, namely how parents are selected and how offsprings are generated. In the context of forward-time simulation, if we populate an offspring population from a parental population, a true effective population size can be calculated, under certain assumptions, as

$$Ne = \frac{kN - 1}{k - 1 + V_k/k}$$

where k and V_k are the mean and variance of the number of gametes each parent transmits to the offspring generation. Naturally, the number of sex chromosomes transmitted will be different for males and females. This effective size is independent of genotypes and is called the demographic effective size.

Because the calculation of demographic effective size needs to track which alleles are transmitted from parental to offspring population, it has to collect information from both parental and offspring populations, and can only be calculated using the lineage modules of simuPOP. As shown in Example 4.83, a `Stat` operator is applied before mating to mark lineage of alleles of each locus with an individual index, and save the IDs of parents in a variable `Ne_demo_base`. After mating, another `Stat` operator is used to count how many alleles each parent has contributed to the offspring generation, and calculate demographic effective size accordingly. This example uses three virtual subpopulations, a whole subpopulation, all male individuals, and all female individuals, and calculated effective size for loci on an autosome, an X chromosome, and a Y chromosome. As we can imagine, the effective size is 0 at the Y chromosome for all females, because no such chromsome is transmitted from the parental population.

Listing 4.83: Demographic effective population size

```
>>> import simuOpt
>>> simuOpt.setOptions(alleleType='lineage', quiet=True)
>>> import simuPOP as sim
>>> pop = sim.Population([2000], loci=[1]*3,
...     chromTypes=[sim.AUTOSOME, sim.CHROMOSOME_X, sim.CHROMOSOME_Y])
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.3, 0.7]),
...     ],
...     preOps=[
...         sim.Stat(effectiveSize=range(3), subPops=[0, (0,0), (0,1)],
...             vars='Ne_demo_base_sp'),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(effectiveSize=range(3), subPops=[0, (0,0), (0,1)],
...             vars='Ne_demo_sp'),
```

```
...         sim.PyEval(r'"Demographic Ne: %.1f (auto), %.1f (X), %.1f (Y), '
...             r'Males: %.1f, %.1f, %.1f, Females: %.1f, %.1f, %.1f\n"'
...             '% tuple([subPop[0]["Ne_demo"][x] for x in (0, 1, 2)] + '
...             '[subPop[(0,0)]["Ne_demo"][x] for x in (0, 1, 2)] + '
...             '[subPop[(0,1)]["Ne_demo"][x] for x in (0, 1, 2)])')
...     ],
...     gen = 5
... )
Demographic Ne: 2021.2 (auto), 1808.8 (X), 1056.1 (Y), Males: 1038.4, 1049.4, 1056.1, Females: 983.8, 983.8, nan
Demographic Ne: 2024.8 (auto), 1886.4 (X), 918.2 (Y), Males: 965.7, 1014.2, 918.2, Females: 1063.3, 1063.3, nan
Demographic Ne: 2048.7 (auto), 1858.5 (X), 969.2 (Y), Males: 1023.0, 1037.4, 969.2, Females: 1025.1, 1025.1, nan
Demographic Ne: 1955.0 (auto), 1790.6 (X), 956.8 (Y), Males: 958.8, 985.2, 956.8, Females: 996.5, 996.5, nan
Demographic Ne: 2000.5 (auto), 1811.7 (X), 955.1 (Y), Males: 983.8, 966.2, 955.1, Females: 1016.8, 1016.8, nan
5L
```

Effective population sizes could also be estimated from genotypes because changes of genotypes reflects properties of the mating scheme. However, it is important to realize that ***evolving a population for one generation is only one realization of many possible realizations of the same mating scheme*** (effective size). If we consider the demographic effective size as the average effective size of all realizations, estimating effective size from genotypes will be inaccurate unless a large number of unlinked loci are used. The temporal methods essentially try to get better estimate by averaging such realizations across multiple generations, although the demographic effective size might vary due to change of population size.

simuPOP currently provides two temporal methods proposed by Waples (1989) and Jorde & Ryman's (2007). Because these methods estimate effective population size using changes of allele frequencies of samples at two generations, it is necessary to set a baseline generation before any temporal method could be applied.

The baseline information is saved to variable `Ne_temporal_base` when this variable is specified in the `vars` parameter of the `Stat` operator. After the baseline is set, for example, at generation 0, if the operator `Stat` is applied at generations 0, 20, and 40, it will set variable `Ne_waples89_P1`, `Ne_waples89_P2` (for Waples 1989) and `Ne_tempoFS_P1`, `Ne_tempoFS_P2` (for Jorde & Ryman 2007, as implemented in a package `TempoFS`) as the census population size at generation 0, estimated effective population sizes between generation 0 and 20 at generation 20, and estimates between 0 and 40 at generation 40. The variables are lists of three elements: the estimated Ne and lower and upper boundaries of the 95% confidence interval.

Sampling plan 1 assumes that samples are drawn with replacement at the first time point so that some of the individuals sampled in the first time period could have contributed genes to subsequent generations (see Nei and Tajima, 1981 Genetics and other papers). simuPOP uses census population (or subpopulation if the statistics are calcuated for each subpopulations) size as N and consider the sample being a subset of the population (or subpopulation), it should be applied to a virtual subpopulation (e.g. a subset of individuals defined by a `RangeSplitter`) of the whole population. Sample plan 2 treats the sample as a sample from an infinitely-sized population, and should be applied to a population (sample) that is actually extracted from a larger population. Results under both assumptions are calculated and provided so you should choose the ones that match your sampling plan.

Example 4.84 demonstrates how to calculate temporal effective population sizes at a 20 generation interval during evolution, using a fixed baseline generation at generation 0. The statistics are estimated from genotypes at 50 unlinked loci from 500 random samples from a population of size 2000. Instead of drawing random samples explicitly, this example defines a virtual subpopulation that consists of the first 500 individuals in the population. The Stat operator is applied at generations 0, 20, 40, ..., 100 to this virtual subpopulation, with the first output being the census size (of the sample). Because a standard Wright-Fisher random mating scheme is used, the true effective population size should be around 2000. It would be interesting to adjust this evolutionary process (with population expansion, with varying number of offspring etc) and the method of estimation (sample size, generations between estimates) to see how well this statistic estimate effective population size under different scenarios.

Listing 4.84: Temporal effective population size using a fixed baseline sample

```
>>> import simuPOP as sim
```

```
>>> pop = sim.Population([2000], loci=[1]*50)
>>> pop.setVirtualSplitter(sim.RangeSplitter([0, 500]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.3, 0.7]),
...         sim.Stat(effectiveSize=range(50), subPops=[(0,0)],
...             vars='Ne_temporal_base'),
...     ],
...     preOps=[
...         sim.Stat(effectiveSize=range(50), subPops=[(0,0)],
...             vars=['Ne_waples89_P1', 'Ne_tempoFS_P1'], step=20),
...         sim.PyEval(r'"Waples Ne: %.1f (%.1f - %.1f), TempoFS: '
...             r'%.1f (%.1f - %.1f), at generation %d\n" % '
...             'tuple(Ne_waples89_P1 + Ne_tempoFS_P1 + [gen])', step=20)
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 101
... )
Waples Ne: 500.0 (500.0 - 500.0), TempoFS: 500.0 (500.0 - 500.0), at generation 0
Waples Ne: 1853.1 (1155.2 - 3536.1), TempoFS: 1843.2 (1255.1 - 3467.7), at generation 20
Waples Ne: 1537.9 (979.7 - 2452.6), TempoFS: 1565.7 (1117.0 - 2617.2), at generation 40
Waples Ne: 1843.3 (1178.0 - 2872.4), TempoFS: 1963.4 (1332.2 - 3730.9), at generation 60
Waples Ne: 1783.0 (1143.4 - 2710.7), TempoFS: 1807.2 (1291.5 - 3008.7), at generation 80
Waples Ne: 1572.7 (1011.2 - 2346.6), TempoFS: 1639.5 (1205.1 - 2563.6), at generation 100
101L
```

Instead of using a fixed baseline generation, it is also possible to reset baseline generation during evolution. For example, Example 4.85 demonstrates how to calculate temporal effective population sizes at a 20 generation interval during evolution. This example sets variable `Ne_temporal_base` with `Ne_waples89_P1` whenever the Stat operator is applied. This effectively resets the baseline generation to the present generation at generations 0, 20, 40, etc, so baseline generations 0, 20, 40, ... are used at generations 20, 40, ... This example also demonstrates how to use the suffix parameter to apply the same statistics with different parameters.

Listing 4.85: Temporal effective population size between consecutive samples

```
>>> import simuPOP as sim
>>> pop = sim.Population([2000], loci=[1]*50)
>>> pop.setVirtualSplitter(sim.RangeSplitter([0, 500]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.3, 0.7]),
...         sim.Stat(effectiveSize=range(50), subPops=[(0,0)],
...             vars='Ne_temporal_base'),
...     ],
...     preOps=[
...         sim.Stat(effectiveSize=range(50), subPops=[(0,0)],
...             vars='Ne_waples89_P1', step=20),
...         sim.Stat(effectiveSize=range(50), subPops=[(0,0)], step=20,
...             suffix='_i', vars=['Ne_temporal_base', 'Ne_waples89_P1']),
...         sim.PyEval(r'"Waples Ne (till %d): %.1f (%.1f - %.1f), '
...             r'(interval) %.1f (%.1f - %.1f)\n" % '
...             'tuple([gen] + Ne_waples89_P1 + Ne_waples89_P1_i)',
...             step=20)
```

```
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 101
... )
Waples Ne (till 0): 500.0 (500.0 - 500.0), (interval) 500.0 (500.0 - 500.0)
Waples Ne (till 20): 1853.1 (1155.2 - 3536.1), (interval) 1853.1 (1155.2 - 3536.1)
Waples Ne (till 40): 1537.9 (979.7 - 2452.6), (interval) 2063.7 (1281.1 - 4094.1)
Waples Ne (till 60): 1843.3 (1178.0 - 2872.4), (interval) 1681.9 (1052.1 - 3112.9)
Waples Ne (till 80): 1783.0 (1143.4 - 2710.7), (interval) 1872.7 (1167.0 - 3586.3)
Waples Ne (till 100): 1572.7 (1011.2 - 2346.6), (interval) 2056.1 (1276.6 - 4073.3)
101L
```

Linkage disequilibrium method is another popular method to estimate effective population size. Compared to temporal methods, it has the distinct advantage that it requires only one sample. simuPOP provides a method that is developed by Waples in his 2006 paper. To use this method, you will need to specify variable `Ne_LD` for a random mating scheme, or `Ne_LD_mono` for a monogamous mating scheme. 4.86 demonstrates this usage. Note that because the LDNe mehod is sensitive to rare alleles (which can lead to inflated measure of LD), simuPOP provides estimates that ignores alleles with frequencies less than 0 (all alleles are kept), 0.01, 0.02 and 0.05. The results are saved in variable `Ne_LD` as a dictionary with keys 0, 0.01, 0.02, 0.05, and values as lists of estimated effective population sizes and their 95% confidence intervals. Because of the existence of many rare alleles, the example gives quite different estimates with and without rare alleles (using cutoff=0.02).

Listing 4.86: Effective population size estimated using a LD based method

```
>>> import simuPOP as sim
>>> pop = sim.Population([2000], loci=[1]*50)
>>> pop.setVirtualSplitter(sim.RangeSplitter([0, 500]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.005]*4 + [0.015]*2 + [0.25, 0.7]),
...     ],
...     preOps=[
...         sim.Stat(effectiveSize=sim.ALL_AVAIL, subPops=[(0,0)],
...             vars='Ne_LD', step=20),
...         sim.PyEval(r'"LD Ne (gen %d): %.1f (%.1f - %.1f)'
...             r', %.1f (%.1f - %.1f, adjusted)\n" % '
...             'tuple([gen] + Ne_LD[0.] + Ne_LD[0.02])',
...             step=20)
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 101
... )
LD Ne (gen 0): 30623.2 (5220.9 - inf), inf (8071.2 - inf, adjusted)
LD Ne (gen 20): 6297.4 (2574.4 - inf), 1900.0 (1160.3 - 4647.8, adjusted)
LD Ne (gen 40): 2187.6 (1554.1 - 3589.2), 2535.5 (1459.2 - 8173.8, adjusted)
LD Ne (gen 60): 2757.8 (1799.2 - 5619.3), 3510.9 (1801.6 - 32066.7, adjusted)
LD Ne (gen 80): 2574.0 (1729.7 - 4828.9), 1813.2 (1197.7 - 3501.7, adjusted)
LD Ne (gen 100): 3234.6 (1819.5 - 12210.9), 2834.8 (1603.4 - 10168.4, adjusted)
101L
```

simuPOP allows you to estimate effective population size using genotypes at selected loci from selected individuals. It is up to you, however, to decide when to apply the operator (pre- or post-mating), how to draw samples, and select the right method for your data. For example, the temporal methods assume discrete generations and no (or slight) selection, migration, and mutation. The LD method assumes that markers are selectively neutral and independent;

population has discrete generations and is closed to immigration; and sampling is random. In addition, to keep the interface simple, simuPOP does not provide many options as dedicated programs do (e.g. TempoFS). Please export your samples in other formats (e.g. use operator `Export(format="GENEPOP")` or function `export(pop, format="GENEPOP")` from module `simuPOP.utiles`) and use these programs if you need such flexibilities.

4.11.17 Other statistics

If you need other statistics, a popular approach is to define them using Python operators. If your statistics is based on existing statistics such as allele frequency, it is a good idea to calculate existing statistics using a `stat` function and derive your statistics from population variables. Please refer to the last chapter of this guide on an example.

If you would like to calculate some summary statistics that involves individual information fields but cannot be calculated using parameters such as minOfInfo, you can try to use operators such as InfoExec to process individuals one by one and collect result. For example, you can use operators

```
PyExec('s=0')
InfoExec('s+=x*x')
PyEval('s')
```

to calculate and report $s = \sum x^2$ where x is an information field during evolution. This makes use of the fact that operator `InfoExec` goes through all individuals and evaluate the statement.

If performance becomes a problem, you might want to have a look at the source code of simuPOP and implement your statistics at the C++ level. If you believe that your statistics are popular enough, please send your implementation to the simuPOP mailinglist for possible inclusion of your statistics into simuPOP.

4.11.18 Support for sex and customized chromosome types

simuPOP supports statistics calculation for loci on sex chromosomes. For example, when pair-wise difference between haplotypes is calculated using parameter `neutrality`, it will pick the right haplotypes for X, and Y chromosomes. However, because neutrality is calculated based on a group of haplotypes of all specified loci, even if the loci are collected across chromosomes, you can not use operator

```
Stat(neutrality=ALL_AVAIL)
```

if the loci are selected from chromosomes of different types, because different numbers of haplotypes exists on these chromosomes. To calculate `Pi` for these chromosomes, you would have to calculate them separately, using operators such as

```
Stat(neutrality=range(30,40), suffix='_X')
Stat(neutrality=range(40,50), suffix='_Y')
```

so that all specified loci are on the same type of chromosomes. Here we use parameter `suffix` to avoid conflict of variable names because both operator would produce the same variable `Pi` without this parameter.

The case with customized chromosomes are more complex because the meaning of these chromosomes are defined by users. If these chromosomes are mitochondrial DNAs, only chromosomes from the females are carrying useful information. If you would like to calculate, for example, the `Pi` statistics for these chromosomes, you will have to explicitly selected females for calculation. This can be done by operator

```
Stat(neutrality=range(50,60), vsps=[(ALL_AVAIL, 'FEMALE')], suffix='_mt')
```

if VSPs have been created by a `SexSplitter`.

Example 4.87 demonstrates the use of these operators. This example intentionally initializes all individuals with the same haplotypes on all chromosomes (the `InitGenotype` operator ignores chromosome types). Because of different

chromosome types, four `Stat` operators are used to get the `Pi` statistics for them. These operators return different results because different sets of haplotypes are picked for the calculation of this statistics.

Listing 4.87: Statistics for sex and customized chromosome types

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=[5]*4,
...     chromTypes=[sim.AUTOSOME, sim.CHROMOSOME_X, sim.CHROMOSOME_Y, sim.MITOCHONDRIAL])
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(haplotypes=[ [0, 1, 2, 0, 1]*4, [2, 1, 0, 2, 3]*4 ],
...             prop=[0.4, 0.6]),
...     ],
...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.MitochondrialGenoTransmitter()]),
...     preOps=[
...         sim.Stat(neutrality=range(5)),
...         sim.Stat(neutrality=range(5, 10), suffix='_X'),
...         sim.Stat(neutrality=range(10, 15), suffix='_Y'),
...         sim.Stat(neutrality=range(15, 20), suffix='_mt'),
...         sim.PyEval(r'"%.3f %.3f %.3f %.3f\n" % (Pi, Pi_X, Pi_Y, Pi_mt)'),
...     ],
...     gen = 2
... )
1.921 1.900 1.973 1.914
1.931 1.921 1.957 1.945
2L
```

4.12 Conditional operators

4.12.1 Conditional operator (operator `IfElse`) *

Operator `IfElse` provides a simple way to conditionally apply an operator. The condition can be a fixed condition, a expression (a string) that will be evaluated in a population's local namespace or a user-defined function when it is applied to the population.

The first case is used to control the execution of certain operators depending on user input. For example, Example 4.88 determines whether or not some outputs should be given depending on a variable `verbose`. Note that the applicability of the conditional operators are determined by the `IfElse` operator and individual opearators. That is to say, the parameters `begin`, `step`, `end`, `at`, and `reps` of operators in `ifOps` and `elseOps` are only honored when operator `IfElse` is applied.

Listing 4.88: A conditional opeartor with fixed condition

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=1000, loci=1)
>>> verbose = True
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
```

```
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=sim.IfElse(verbose,
...         ifOps=[
...             sim.Stat(alleleFreq=0),
...             sim.PyEval(r"'Gen: %3d, allele freq: %.3f\n' % (gen, alleleFreq[0][1])",
...                 step=5)
...         ],
...         begin=10),
...     gen = 30
... )
Gen:  10, allele freq: 0.483
Gen:  15, allele freq: 0.455
Gen:  20, allele freq: 0.481
Gen:  25, allele freq: 0.481
30L
```

When a string is specified, it will be considered as an expression and be evaluated in a population's namespace. The return value will be used to determine if an operator should be executed. For example, you can re-introduce a mutant if it gets lost in the population, output a warning when certain condition is met, or record the occurance of certain events in a population. For example, Example 4.89 records the number of generations the frequency of an allele goes below 0.4 and beyong 0.6 before it gets lost or fixed in the population. Note that a list of else-operators can also be executed when the condition is not met.

Listing 4.89: A conditional opeartor with dynamic condition

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(
...     sim.Population(size=1000, loci=1),
...     rep=4)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         sim.PyExec('below40, above60 = 0, 0')
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.IfElse('alleleFreq[0][1] < 0.4',
...             sim.PyExec('below40 += 1')),
...         sim.IfElse('alleleFreq[0][1] > 0.6',
...             sim.PyExec('above60 += 1')),
...         sim.IfElse('len(alleleFreq[0]) == 1',
...             sim.PyExec('stoppedAt = gen')),
...         sim.TerminateIf('len(alleleFreq[0]) == 1')
...     ]
... )
(892L, 1898L, 4001L, 2946L)
>>> for pop in simu.populations():
...     print('Overall: %4d, below 40%%: %4d, above 60%%: %4d' % \
...         (pop.dvars().stoppedAt, pop.dvars().below40, pop.dvars().above60))
...
Overall:  891, below 40%:   20, above 60%:  515
Overall: 1897, below 40%: 1039, above 60%:   51
```

```
Overall: 4000, below 40%: 2878, above 60%:    0
Overall: 2945, below 40%:  198, above 60%: 1731
```

In the last case, a user-defined function can be specified. This function should accept parameter `pop` when the operator is applied to a population, and one or more parameters `pop`, `off`, `dad` and `mom` when it is applied during-mating. The later could be used to apply different during-mating operators for different types of parents or offspring. For example, Example 5.40 in Chapter 6 uses a `CloneGenoTransmitter` when only one parent is available (when parameter `mom` is `None`), and a `MendelianGenoTransmitter` when two parents are available.

4.12.2 Conditionally terminate an evolutionary process (operator `TerminateIf`)

Operator `TerminateIf` has been described and used in several examples such as Example 5.20, 3.19 and 4.89. This operator accept an Python expression and terminate the evolution of the population being applied if the expression is evaluated to be `True`. This operator is well suited for situations where the number of generations to evolve cannot be determined in advance.

If a `TerminateIf` operator is applied to the offspring generation, the evolutionary cycle is considered to be completed. If the evolution is terminated before mating, the evolutionary cycle is condered to be incomplete. Such a difference can be important if the number of generations that have been involved is important for your analysis.

A less-known feature of operator `TerminateIf` is its ability to terminate the evolution of all replicates, using parameter `stopAll=True`. For example, Example 4.90 terminates the evolution of all populations when one of the populations gets fixed. The return value of `simu.evolve` shows that some populations have evolved one generation less than the population being fixed.

Listing 4.90: Terminate the evolution of all populations in a simulator

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(
...     sim.Population(size=100, loci=1),
...     rep=10)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.TerminateIf('len(alleleFreq[0]) == 1', stopAll=True)
...     ]
... )
(88L, 88L, 88L, 88L, 87L, 87L, 87L, 87L, 87L, 87L)
>>>
```

4.12.3 Conditionally revert an evolutionary process to a saved state (operator `RevertIf`)

Operator `RevertIf` is a very interesting operator. It accepts a condition and a saved population (`.pop` file) and will revert the current evolving population to the saved population if the condition is met.

For example, one of the biggest problem with introducing a disease allele to an evolving population (using an operator `PointMutator`) is that the disease allele will very likely get lost because of genetic drift. It is possible to simulate the allele frequency trajectory backward in time and follow the trajectory during the forward-time simulation phase (`simuPOP.utils.simulateBackwardTrajectory`, `simuPOP.utils.simulateForwardTrajectory`, and a `ControlledRandomMating`

mating scheme with a `ControlledOffspringGenerator`). However, that method is applicable only to evolutionary processes with a small number of loci under selection, and has a number of limitations (e.g. unlinked disease predisposing loci).

A natural way to simulate the introduction of disease alleles is therefore to terminate and restart the simulation whenever the disease allele gets lost (or fixed). This could be done by splitting the evolutionary process into two stages. The disease allele is introduced at the second stage and the simulation will be terminated as soon as the introduced allele is lost (using a `TerminateIf` operator). The second stage would be repeated until the simulation succeeds. Alternatively, you can save the population before the introduction of the disease allele, and revert to the saved population when the introduced allele gets lost. The latter can be done using operator `RevertIf`.

Example 4.91 shows an example of such an evolutionary process. This example saves an evolving population at the beginning of generation 4 and introduces a disease allele to the population. Starting from the fifth generation, a RevertIf opeartor checks if the disease allele still exists in the population, and revert to the saved population if the allele has been lost. When you read the example, it is important to remind yourself that after the `RevertIf` operator is triggered and applied, the population is at generation 4 before the disease allele is introduced, This is why the `SavePopulation` and `RevertIf` operators are usually put together.

Listing 4.91: Revert an evolutionary process to a previous saved state when an introduced allele is lost

```
>>> import simuPOP as sim
>>>
>>> pop = sim.Population(1000, loci=1)
>>> evolved = pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=[
...         sim.SavePopulation('init.pop', at=4),
...         sim.RevertIf('alleleFreq[0][1] == 0', "init.pop", begin=5),
...         sim.PointMutator(at=4, inds=0, allele=1, loci=0),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.PyEval(r"'%d %.4f\n' % (gen, alleleFreq[0][1])"),
...         ],
...     gen=20
... )
0 0.0000
1 0.0000
2 0.0000
3 0.0000
4 0.0000
4 0.0000
4 0.0005
5 0.0010
6 0.0005
7 0.0010
8 0.0015
9 0.0010
10 0.0010
11 0.0005
12 0.0010
13 0.0010
14 0.0030
15 0.0015
16 0.0010
```

```
17 0.0000
4 0.0000
4 0.0005
5 0.0000
4 0.0005
5 0.0005
6 0.0005
7 0.0010
8 0.0005
9 0.0010
10 0.0000
4 0.0005
5 0.0005
6 0.0020
7 0.0040
8 0.0020
9 0.0015
10 0.0025
11 0.0015
12 0.0030
13 0.0015
14 0.0040
15 0.0045
16 0.0040
17 0.0065
18 0.0070
19 0.0055
>>> print('Evolved {} generations'.format(evolved))
Evolved 46 generations
```

Although operator `RevertIf` can also be used to fast-forward an evolutionary process (skip to a previously saved state directly), it is tricky to make it work with complex demographic models and therefore not recommended.

4.12.4 Conditional during mating operator (operator `DiscardIf`)

Operator `DiscardIf` accepts a condition or a Python function. When it is applied during mating, it will evaluate the condition or call the function for each offspring, and discard the offspring if the return value of the expression or function is True. The python expression accepts information fields as variables so operator `DiscardIf('age > 80')` will discard all individuals with age > 80. Optionally, the offspring itself can be used in the expression if parameter exposeInd is used to set the variable name of the offspring.

Alternatively, a Python function can be passed to this operator. This function should be defined with parameters `pop`, `off`, `mom`, `dad` or names of information fields. For example, `DiscardIf(lambda age:  age > 80)` will remove individuals with age > 80.

A constant expression is also allowed in this operator. Although it does not make sense to use `DiscardIf(True)` because all offspring will be discarded, it is quite useful to use this operator in the context of `DiscardIf(True, subPops=[(0, 0)])` to remove all individuals in a virtual subpopulation. If virtual subpopulation `(0, 0)` is defined as all individuals with age > 80, the last method achieves the same effect as the first two methods.

Example 4.92 demonstrates an interesting application of this operator. This example evolves a population for one generation. Instead of keeping all offspring, it keeps only 500 affected and 500 unaffected offspring. This is achieved by defining virtual subpopulations by affection status and range, and discard the first 500 offspring if they are unaffected, and the last 500 offspring if they are affected.

Listing 4.92: Use operator DiscardIf to generate case control samples

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=500, loci=1)
>>> pop.setVirtualSplitter(sim.ProductSplitter([
...     sim.AffectionSplitter(),
...     sim.RangeSplitter([[0,500], [500, 1000]]),
...     ])
... )
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...     ],
...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.MaPenetrance(loci=0, penetrance=[0, 0.01, 0.1]),
...             sim.DiscardIf(True, subPops=[
...                 (0, 'Unaffected, Range [0, 500)'),
...                 (0, 'Affected, Range [500, 1000)')])
...         ],
...         subPopSize=1000,
...     ),
...     gen = 1
... )
1L
>>> sim.stat(pop, numOfAffected=True)
>>> print(pop.dvars().numOfAffected, pop.dvars().numOfUnaffected)
(500, 500)
```

4.13 Miscellaneous operators

4.13.1 An operator that does nothing (operator `NoneOp`)

Operator `NoneOp` does nothing when it is applied to a population. It provides a placeholder when an operator is needed but no action is required. Example 4.13.1 demonstrates a typical usage of this operator

```
if hasSelection:
    sel = MapSelector(loci=[0], fitness=[1, 0.99, 0.98])
else:
    sel = NoneOp()
#
simu.evolve(
    preOps=[sel], # and other operators
    matingScheme=RandomMating(),
    gen=10
)
```

4.13.2 dump the content of a population (operator `Dumper`)

Operator `Dumper` and its function form `dump` has been used extensively in this guide. They are prefect for demonstration and debugging purposes because they display all properties of a population in a human readable format. They are, however, rarely used in realistic settings because outputting a large population to your terminal can be disastrous.

Even with modestly-sized populations, it is a good idea to dump only parts of the population that you are interested. For example, you can use parameter `genotype=False` to stop outputting individual genotype, `structure=False` to stop outtputing genotypic and population structure information, `loci=range(5)` to output genotype only at the first five loci, `max=N` to output only the first `N` individuals (default to `100`), `subPops=[(0, 0)]` to output, for example, only the first virtual subpopulation in subpopulation 0. Multiple virtual subpopulations are allowed and you can even use `subPops=[(ALL_AVAIL, 0)]` to go through a specific virtual subpopulation of all subpopulations. This operator by default only dump the present generation but you can set `ancGens` to a list of generation numbers or `ALL_AVAIL` to dump part or all ancestral generations. Finally, if there are more than 10 alleles, you can set the `width` at which each allele will be printed. The following example (Example 4.93) presents a rather complicated usage of this operator.

Listing 4.93: dump the content of a population

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[10, 10], loci=[20, 30], infoFields='gen',
...     ancGen=-1)
>>> sim.initSex(pop)
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop1 = pop.clone()
>>> sim.initGenotype(pop, freq=[0]*20 + [0.1]*10)
>>> pop.setIndInfo(1, 'gen')
>>> sim.initGenotype(pop1, freq=[0]*50 + [0.1]*10)
>>> pop1.setIndInfo(2, 'gen')
>>> pop.push(pop1)
>>> sim.dump(pop, width=3, loci=[5, 6, 30], subPops=([0, 0], [1, 1]),
...     max=10, structure=False)
SubPopulation 0,0 (Male), 5 Individuals:
   2: MU  56 54 52 |  58 54 51 |  2
   3: MU  52 50 51 |  56 51 50 |  2
   4: MU  50 53 52 |  52 59 56 |  2
   5: MU  57 54 56 |  57 57 53 |  2
   6: MU  59 54 54 |  57 51 50 |  2
SubPopulation 1,1 (Female), 7 Individuals:
  10: FU  54 53 57 |  59 59 59 |  2
  11: FU  55 59 51 |  59 51 58 |  2
  12: FU  55 58 58 |  57 54 58 |  2
  14: FU  53 57 52 |  51 54 58 |  2
  15: FU  51 58 59 |  54 52 54 |  2

>>> # list all male individuals in all subpopulations
>>> sim.dump(pop, width=3, loci=[5, 6, 30], subPops=[(sim.ALL_AVAIL, 0)],
...     max=10, structure=False)
SubPopulation 0,0 (Male), 5 Individuals:
   2: MU  56 54 52 |  58 54 51 |  2
   3: MU  52 50 51 |  56 51 50 |  2
   4: MU  50 53 52 |  52 59 56 |  2
   5: MU  57 54 56 |  57 57 53 |  2
   6: MU  59 54 54 |  57 51 50 |  2
SubPopulation 1,0 (Male), 3 Individuals:
  13: MU  55 52 53 |  57 56 52 |  2
  17: MU  55 51 51 |  57 55 51 |  2
```

```
 19: MU  56 54 53 |  58 58 56 |  2
```

4.13.3 Save a population during evolution (operator `SavePopulation`)

Because it is usually not feasible to store all parental generations of an evolving population, it is a common practise to save snapshots of a population during an evolutionary process for further analysis. Operator `SavePopulation` is designed for this purpose. When it is applied to a population, it will save the population to a file specified by parameter `output`.

The tricky part is that populations at different generations need to be saved to different filenames so the expression version of parameter `output` needs to be used (see operator `BaseOperator` for details). For example, expression `'snapshot_%d_%d.pop' % (rep, gen)` is used in Example 4.94 to save population to files such as `snapshot_5_20.pop` during the evolution.

Listing 4.94: Save snapshots of an evolving population

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(100, loci=2),
...     rep=5)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2, 0.8])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=sim.SavePopulation(output="!'snapshot_%d_%d.pop' % (rep, gen)",
...             step = 10),
...     gen = 50
... )
(50L, 50L, 50L, 50L, 50L)
```

4.13.4 Pause and resume an evolutionary process (operator `Pause`) *

If you are presenting an evolutinary process in public, you might want to temporarily stop the evolution so that your audience can have a better look at intermediate results or figures. If you have an exceptionally long evolutionary process, you might want to examine the status of the evolution process from time to time. These can be done using a `Pause` operator.

The `Pause` operator can stop the evolution at specified generations, or when you press a key. In the first case, you usually specify the generations to Pause (e.g. `Pause(step=1000)`) so that you can examine the status of a simulation from time to time. In the second case, you can apply the operator at each generation and Pause the simulation when you press a key (e.g. `Pause(stopOnKeyStroke=True)`). A specific key can be specified so that you can use different keys to stop different populations, as shown in Example 4.95.

Listing 4.95: Pause the evolution of a simulation

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(100), rep=10)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[sim.Pause(stopOnKeyStroke=str(x), reps=x) for x in range(10)],
...     gen = 100
```

```
... )
(100L, 100L, 100L, 100L, 100L, 100L, 100L, 100L, 100L, 100L)
```

When a simulation is Paused, you are given the options to resume evolution, stop the evolution of the Paused population or all populations, or enter an interactive Python shell to examine the status of a population, which will be available in the Python shell as `pop_X_Y` where `X` and `Y` are generation and replicate number of the population, respectively. The evolution will resume after you exit the Python shell.

4.13.5 Measuring execution time of operators (operator `TicToc`) *

The `TicToc` operator can be used to measure the time between two events during an evolutionary process. It outputs the elapsed time since the last time it is called, and the overall time since the operator is created. It is very flexible in that you can measure the time spent for mating in an evolutionary cycle if you apply it before and after mating, and you can measure time spent for several evolutionary cycles using generation applicability parameters such as `step` and `at`. The latter usage is demonstrated in Example 4.96.

Listing 4.96: Monitor the performance of operators

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(10000, loci=[100]*5), rep=2)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.1, 0.9])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.TicToc(step=50, reps=-1),
...     ],
...     gen = 101
... )
Start stopwatch.
Elapsed time: 2.00s      Overall time: 2.00s
Elapsed time: 1.00s      Overall time: 3.00s
(101L, 101L)
```

4.14 Hybrid and Python operators

4.14.1 Hybrid operators

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

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

simuPOP operators use parameter names to determine which information should be passed to a user-defined function. For example, a hybrid quantitative trait operator recognizes parameters `ind`, `geno`, `gen` and names of information fields such as `smoking`. If your model depends on genotype, you could provide a function with parameter geno (e.g. `func(geno)`); if your model depends on smoking and genotype, you could provide a function with parameters geno and smoking (e.g. `func(geno, smoking)`); if you model depends on individual sex, you can use a function that passes the whole individual (e.g. `func(ind)`) so that you could check individual sex. When a hybrid operator is applied to a population, it will check the parameter names of provided Python function and send requested information automatically.

For example, Example 4.97 defines a three-locus heterogeneity penetrance model [Risch, 1990] that yields positive penetrance only when at least two disease susceptibility alleles are available. The underlying mechanism of this operator is that for each individual, simuPOP will collect genotype at specified loci (parameter `loci`) and send them to function `myPenetrance` and evaluate. The return values are used as the penetrance value of the individual, which is then interpreted as the probability that this individual will become affected.

Listing 4.97: Use a hybrid operator

```
>>> import simuPOP as sim
>>> def myPenetrance(geno):
...     'A three-locus heterogeneity penetrance model'
...     if sum(geno) < 2:
...         return 0
...     else:
...         return sum(geno)*0.1
...
>>> pop = sim.Population(1000, loci=[20]*3)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.8, 0.2])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.PyPenetrance(func=myPenetrance, loci=[10, 30, 50]),
...         sim.Stat(numOfAffected=True),
...         sim.PyEval(r"'%d: %d\n' % (gen, numOfAffected)")
...     ],
...     gen = 5
... )
0: 97
1: 96
2: 78
3: 95
4: 80
5L
```

4.14.2 Python operator `PyOperator` *

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

A `PyOperator` that is applied pre- or post- mating expects a function with one or both parameters `pop` and `param`, where `pop` is the population being applied, and `param` is optional, depending on whether or not a parameter is passed to the `PyOperator()` constructor. Function `func` can perform arbitrary action to `pop` and must return `True` or `False`. **The**

evolution of pop will be stopped if this function returns False. This is essentially how operator `TerminateIf` works. Alternatively, this callback function can accept `ind` as one of the parameters. In this case, the function will be called for all individuals or individuals in specified (virtual) subpopulations. **Individuals will be removed from the populaton if this function returns False**.

Example 4.98 defines such a function. It accepts a cutoff value and two mutation rates as parameters. It then calculate the frequency of allele 1 at each locus and apply a two-allele model at high mutation rate if the frequency is lower than the cutoff and a low mutation rate otherwise. The `kAlleleMutate` function is the function form of a mutator `KAlleleMutator` (see Section 4.1.6 for details).

Listing 4.98: A frequency dependent mutation operator

```
import simuPOP as sim
def dynaMutator(pop, param):
    '''This mutator mutates commom loci with low mutation rate and rare
    loci with high mutation rate, as an attempt to raise allele frequency
    of rare loci to an higher level.'''
    # unpack parameter
    (cutoff, mu1, mu2) = param;
    sim.stat(pop, alleleFreq=range(pop.totNumLoci()))
    for i in range(pop.totNumLoci()):
        # Get the frequency of allele 1 (disease allele)
        if pop.dvars().alleleFreq[i][1] < cutoff:
            sim.kAlleleMutate(pop, k=2, rates=mu1, loci=[i])
        else:
            sim.kAlleleMutate(pop, k=2, rates=mu2, loci=[i])
    return True
```

Example 4.99 demonstrates how to use this operator. It first initializes the population using two `InitGenotype` operators that initialize loci with different allele frequencies. It applies a `PyOperator` with function `dynaMutator` and a tuple of parameters. Allele frequencies at all loci are printed at generation `0`, `10`, `20`, and `30`. Note that this `PyOperator` is applied at to the parental generation so allele frequencies have to be recalculated to be used by post-mating operator `PyEval`.

Listing 4.99: Use a PyOperator during evolution

```
>>> pop = sim.Population(size=10000, loci=[2, 3])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.99, .01], loci=[0, 2, 4]),
...         sim.InitGenotype(freq=[.8, .2], loci=[1, 3])
...     ],
...     preOps=sim.PyOperator(func=dynaMutator, param=(.2, 1e-2, 1e-5)),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=range(5), step=10),
...         sim.PyEval(r"' '.join(['%.2f' % alleleFreq[x][1] for x in range(5)]) + '\n'",
...             step=10),
...     ],
...     gen = 31
... )
0.02 0.20 0.02 0.20 0.02
0.11 0.22 0.11 0.20 0.11
0.19 0.21 0.20 0.20 0.18
0.21 0.21 0.22 0.21 0.21
31L
```

4.14.3 During-mating Python operator *

A `PyOperator` can also be applied during-mating. They can be used to filter out unwanted offspring (by returning `False` in a user-defined function), modify offspring, calculate statistics, or pass additional information from parents to offspring. Depending the names of parameters of your function, the Python operator will pass offspring (parameter `off`), his or her parents (parameter `dad` and `mom`), the whole population (parameter `pop`) and an optional parameter (parameter `param`) to this function. For example, function `func(off)` will accept references to an offspring, and `func(off, mom, dad)` will accept references to both offspring and his or her parents.

Example 4.100 demonstrates the use of a during-mating Python operator. This operator rejects an offspring if it has allele 1 at the first locus of the first homologous chromosome, and results in an offspring population without such individuals.

Listing 4.100: Use a during-mating PyOperator

```
>>> import simuPOP as sim
>>> def rejectInd(off):
...     'reject an individual if it off.allele(0) == 1'
...     return off.allele(0) == 0
...
>>> pop = sim.Population(size=100, loci=1)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.PyOperator(func=rejectInd)
...         ]),
...     gen = 1
... )
1L
>>> # You should see no individual with allele 1 at locus 0, ploidy 0.
>>> pop.genotype()[:20]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
```

`PyOperator` is the most powerful operator in simuPOP and has been widely used, for example, to calculate statistics and is not supported by the `Stat()` operator, to examine population property during evolution, or prepare populations for a special mating scheme. However, because `PyOperator` works in the Python interpreter, it is expected that it runs slower than operators that are implemented at the C/C++ level. If performance becomes an issue, you can re-implement part or all the operator in C++. Section 5.3.5 describes how to do this.

4.14.4 Define your own operators *

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

Example 5.23 rewrites the `dynaMutator` defined in Example 4.98 into a derived operator. The parameters are now passed to the constructor of `dynaMutator` and are saved as member variables. A member function `mutate` is defined

and is passed to the constructor of `PyOperator`. Other than making `dynaMutator` look like a real simuPOP operator, this example does not show a lot of advantage over defining a function. However, when the operator gets complicated (as in the case for `VarPlotter`), the object oriented implementation will prevail.

Listing 4.101: Define a new Python operator

```
>>> import simuPOP as sim
>>> class dynaMutator(sim.PyOperator):
...     '''This mutator mutates commom loci with low mutation rate and rare
...     loci with high mutation rate, as an attempt to raise allele frequency
...     of rare loci to an higher level.'''
...     def __init__(self, cutoff, mu1, mu2, *args, **kwargs):
...         self.cutoff = cutoff
...         self.mu1 = mu1
...         self.mu2 = mu2
...         sim.PyOperator.__init__(self, func=self.mutate, *args, **kwargs)
...     #
...     def mutate(self, pop):
...         sim.stat(pop, alleleFreq=range(pop.totNumLoci()))
...         for i in range(pop.totNumLoci()):
...             # Get the frequency of allele 1 (disease allele)
...             if pop.dvars().alleleFreq[i][1] < self.cutoff:
...                 sim.kAlleleMutate(pop, k=2, rates=self.mu1, loci=[i])
...             else:
...                 sim.kAlleleMutate(pop, k=2, rates=self.mu2, loci=[i])
...         return True
...
>>> pop = sim.Population(size=10000, loci=[2, 3])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.99, .01], loci=[0, 2, 4]),
...         sim.InitGenotype(freq=[.8, .2], loci=[1, 3])
...     ],
...     preOps=dynaMutator(cutoff=.2, mu1=1e-2, mu2=1e-5),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=range(5), step=10),
...         sim.PyEval(r"' '.join(['%.2f' % alleleFreq[x][1] for x in range(5)]) + '\n'",
...             step=10),
...     ],
...     gen = 31
... )
0.02 0.20 0.02 0.20 0.02
0.11 0.22 0.11 0.20 0.11
0.19 0.21 0.20 0.20 0.18
0.21 0.21 0.22 0.21 0.21
31L
```

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

Chapter 5

Evolving populations

5.1 Mating Schemes

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

- A **homogeneous mating scheme** is the most flexible and most frequently used mating scheme and is the center topic of this section. A homogeneous mating is composed of a *parent chooser* that is responsible for choosing parent(s) from a (virtual) subpopulation and an *offspring generator* that is used to populate all or part of the offspring generation. During-mating operators are used to transmit genotypes from parents to offspring. Figure 5.1 demonstrates this process.
- A **heterogeneous mating scheme** applies several homogeneous mating scheme to different (virtual) subpopulations. Because the division of virtual subpopulations can be arbitrary, this mating scheme can be used to simulate mating in heterogeneous populations such as populations with age structure.
- A **pedigree mating scheme** evolves a population by following the pedigree structure of a pedigree. This mating scheme is used to a replay a recorded or manually created evolutionary process.

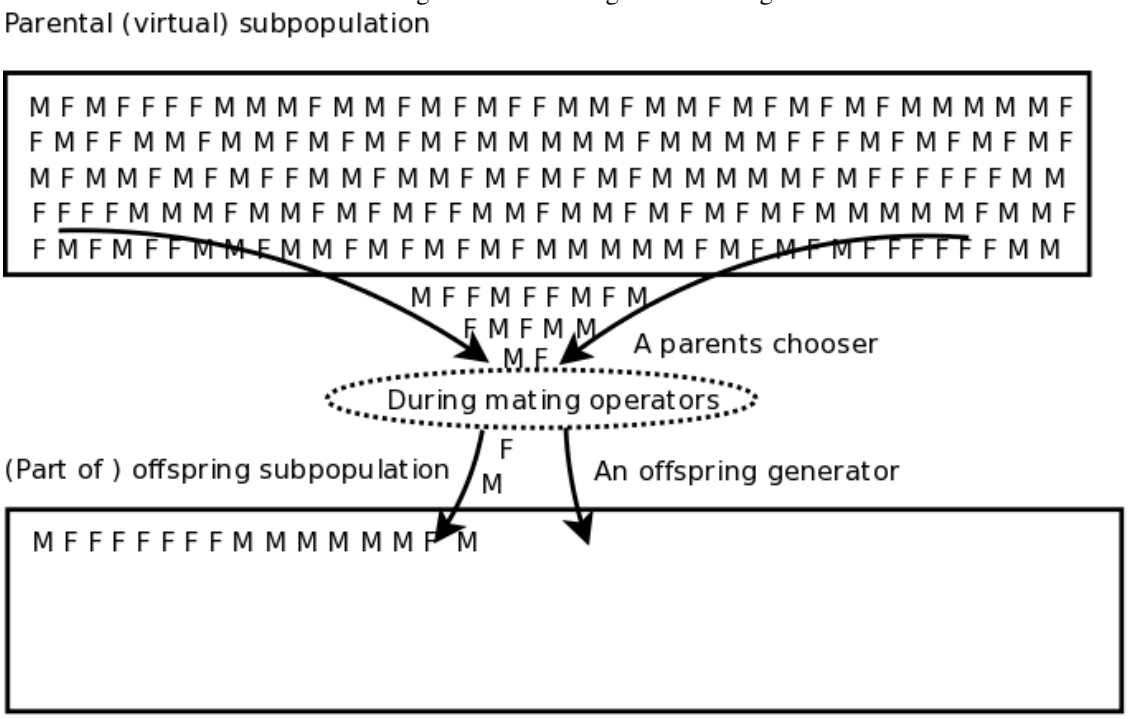
This section describes some standard features of mating schemes and most pre-defined mating schemes. The next section will demonstrate how to build complex nonrandom mating schemes from scratch.

5.1.1 Control the size of the offspring generation

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

- A simuPOP mating scheme, by default, produces an offspring generation that has the same subpopulation sizes as the parental generation. This does not guarantee a constant population size because some operators, such as a `Migrator` and `DiscardIf` can change population or subpopulation sizes.
- If fixed subpopulation sizes are given to parameter `subPopSize`. A mating scheme will generate an offspring generation with specified sizes even if an operator has changed parental population sizes.
- A **demographic function** can be specified to parameter `subPopSize`. This function should take one of the two forms `func(gen)` or `func(gen, pop)` where `gen` is the current generation number and `pop` is the parental population just before mating. This function should return an array of new subpopulation sizes. A single number can be

Figure 5.1: A homogeneous mating scheme



A homogeneous mating scheme is responsible to choose parent(s) from a subpopulation or a virtual subpopulation, and population part or all of the corresponding offspring subpopulation. A parent chooser is used to choose one or two parents from the parental generation, and pass it to an offspring generator, which produces one or more offspring. During mating operators such as taggers and Recombinator can be applied when offspring is generated.

returned if there is only one subpopulation. The `simuPOP.demography` module provides a number of demography-related functions for complex evolutionary secenarios. **Please consider contributing to this module if you have implemented demographic models for particular populations.**

The following examples demonstrate these cases. Example 5.1 uses a default `RandomMating()` scheme that keeps parental subpopulation sizes. Because migration between two subpopulations are asymmetric, the size of the first subpopulation increases at each generation, although the overall population size keeps constant.

Listing 5.1: Free change of subpopulation sizes

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[500, 1000], infoFields='migrate_to')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=sim.Migrator(rate=[[0.8, 0.2], [0.4, 0.6]]),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%s\n" % subPopSize')
...     ],
...     gen = 3
... )
[843, 657]
[948, 552]
[1010, 490]
3L
```

Example 5.2 uses the same Migrator to move individuals between two subpopulations. Because a constant subpopulation size is specified, the offspring generation always has 500 and 1000 individuals in its two subpopulations. Note that operators `Stat` and `PyEval` are applied both before and after mating. It is clear that subpopulation sizes changes before mating as a result of migration, although the pre-mating population sizes vary because of uncertainties of migration.

Listing 5.2: Force constant subpopulation sizes

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[500, 1000], infoFields='migrate_to')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=[
...         sim.Migrator(rate=[[0.8, 0.2], [0.4, 0.6]]),
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%s\n" % subPopSize')
...     ],
...     matingScheme=sim.RandomMating(subPopSize=[500, 1000]),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%s\n" % subPopSize')
...     ],
...     gen = 3
... )
[843, 657]
[500, 1000]
[795, 705]
[500, 1000]
[821, 679]
[500, 1000]
3L
```

Example 5.3 uses a demographic function to control the subpopulation size of the offspring generation. This example implements a linear population expansion model but arbitrarily complex demographic model can be implemented similarly.

Listing 5.3: Use a demographic function to control population size

```
>>> import simuPOP as sim
>>> def demo(gen):
...     return [500 + gen*10, 1000 + gen*10]
...
>>> pop = sim.Population(size=[500, 1000], infoFields='migrate_to')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=sim.Migrator(rate=[[0.8, 0.2], [0.4, 0.6]]),
...     matingScheme=sim.RandomMating(subPopSize=demo),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%s\n" % subPopSize')
...     ],
...     gen = 3
... )
[500, 1000]
[510, 1010]
[520, 1020]
3L
```

If the size of the offspring generation can not be determined directly from generation number, you can pass the parental population as parameter `pop` to the demographic function. For example, Example 5.4 implements a demographic model where a population expand at random numbers at each generation.

Listing 5.4: Use parental population to determine the size of offspring population

```
>>> import simuPOP as sim
>>> import random
>>> def demo(pop):
...     return [x + random.randint(50, 100) for x in pop.subPopSizes()]
...
>>> pop = sim.Population(size=[500, 1000], infoFields='migrate_to')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(subPopSize=demo),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%s\n" % subPopSize')
...     ],
...     gen = 3
... )
[594, 1083]
[656, 1159]
[745, 1246]
3L
```

In all the above examples, migration and demographic changes are introduced manually to influence the evolution of populations. However, the demographic changes might be driven by other factors such as natural selection so that it is difficult to predict the size of offspring generations in advance. In this case, you can manually remove individuals from parental (or offspring) populations using appropriate operators.

For example, a population in Example 5.5 suffers from a sudden reduction of population size (due to perhaps a famine) at generation 3, and a gradual reduction of population size (due to perhaps an outburst of an infectious disease) after generation 5. The first event is implemented using a `ResizeSubPops` operator that directly shrink the population size in half. The second event is implemented using a `MaPenetrance` and a `DiscardIf` operator. The first operator assigns affection status of each individual using a disease model that involves individual genotype. The second operator discard all individuals that are affected with the disease. Despite of these unfortunate events, the population tries to expand exponentially with offspring population sizes set to 105% of their parental populations.

Listing 5.5: Change of population size caused by natural selection

```
>>> import simuPOP as sim
>>> def demo(pop):
...     return int(pop.popSize() * 1.05)
...
>>> pop = sim.Population(size=10000, loci=1)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.7, 0.3])
...     ],
...     preOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%d %s --> " % (gen, subPopSize)'),
...         sim.ResizeSubPops(0, proportions=[0.5], at=2),
...         sim.MaPenetrance(loci=0, penetrance=[0.01, 0.2, 0.6], begin=4),
...         sim.DiscardIf('ind.affected()', exposeInd='ind', begin=4),
```

```
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%s --> " % subPopSize'),
...     ],
...     matingScheme=sim.RandomMating(subPopSize=demo),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%s\n" % subPopSize')
...     ],
...     gen = 6
... )
0 [10000] --> [10000] --> [10500]
1 [10500] --> [10500] --> [11025]
2 [11025] --> [5512] --> [5787]
3 [5787] --> [5787] --> [6076]
4 [6076] --> [5188] --> [5447]
5 [5447] --> [4847] --> [5089]
6L
```

5.1.2 Advanced use of demographic functions *

The parental population passed to a demographic function is usually used to determine offspring population size from parental population size. However, because this function is called immediately before mating happens, it provides a good opportunity for you to prepare the parental generation for mating. Such activities could generally be done by operators, but operations related to demographic changes could be done here. For example, Example 5.6 uses a demographic function to split populations at certain generation. The advantage of this method over the use of a `SplitSubPops` operator (for example as in Example 4.22) is that all demographic information presents in the same function so you do not have to worry about changing an operator when your demographic model changes.

Listing 5.6: Use a demographic function to split parental population

```
>>> import simuPOP as sim
>>> def demo(gen, pop):
...     if gen < 2:
...         return 1000 + 100 * gen
...     if gen == 2:
...         # this happens right before mating at generation 2
...         size = pop.popSize()
...         pop.splitSubPop(0, [size // 2, size - size//2])
...     # for generation two and later
...     return [x + 50 * gen for x in pop.subPopSizes()]
...
>>> pop = sim.Population(1000)
>>> pop.evolve(
...     preOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"Gen %d:\t%s (before mating)\t" % (gen, subPopSize)')
...     ],
...     matingScheme=sim.RandomSelection(subPopSize=demo),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%s (after mating)\n" % subPopSize')
...     ],
...     gen = 5
... )
```

```
Gen 0:  [1000] (before mating)  [1000] (after mating)
Gen 1:  [1000] (before mating)  [1100] (after mating)
Gen 2:  [1100] (before mating)  [650, 650] (after mating)
Gen 3:  [650, 650] (before mating)      [800, 800] (after mating)
Gen 4:  [800, 800] (before mating)      [1000, 1000] (after mating)
5L
```

5.1.3 Determine the number of offspring during mating

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

- If a single number is given, `numOffspring` offspring are produced at each mating event.
- If a Python function is given, this function will be called each time when a mating event happens. Generation number can be passed to this function as parameter `gen` to allow different numbers of offspring at different generations. A python generator function can also be passed to provide an iterator interface to yield number of offspring for all mating events.
- If a tuple (or list) with more than one numbers is given, the first number must be one of `GEOMETRIC_DISTRIBUTION`, `POISSON_DISTRIBUTION`, `BINOMIAL_DISTRIBUTION` and `UNIFORM_DISTRIBUTION`, with one or two additional parameters.

The number of offspring in the last case will then follow a specific statistical distribution. More specifically,

- `numOffspring=(GEOMETRIC_DISTRIBUTION, p)`: The number of offspring for each mating event follows a geometric distribution with mean $1/p$ and variance $(1-p)/p^2$:

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

- `numOffspring=(POISSON_DISTRIBUTION, p)`: The number of offspring for each mating event follows a Poisson distribution with mean p and variance p. The distribution is

$$\Pr(k) = \frac{p^k e^{-p}}{k!} \quad \text{for } k \geq 0$$

  Note that, however, because families with zero offspring are ignored, the distribution of the observed number of offspring (excluding zero) follows a zero-truncated Poission distribution with probability

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

  The mean number of offspring is therefore $\frac{1}{1-e^{-p}}p$, which is 2.31 for $p = 2$.

- `numOffspring=(BINOMIAL_DISTRIBUTION, p, n)`: The number of offspring for each mating event follows a Binomial distribution with mean np and variance $np(1-p)$.

$$\Pr(k) = \frac{n!}{k!(n-k)!}p^k(1-p)^{n-k} \quad \text{for } n \geq k \geq 0$$

  Because families with zero offspring are ignored, the distribution of the observed number of offspring (excluding zero) follows a zero-truncated Bionimial distribution, with mean number of offspring being $\frac{np}{(1-p)^n}$.

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

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

  The lower bound of this distribution can be `0` but is identical to the case with $a = 1$.

Example 5.7 demonstrates how to use parameter `numOffspring`. In this example, a function `checkNumOffspring` is defined. It takes a mating scheme as its input parameter and use it to evolve a population with 30 individuals. After evolving a population for one generation, parental indexes are used to identify siblings, and then the number of offspring per mating event.

Listing 5.7: Control the number of offspring per mating event.

```
>>> import simuPOP as sim
>>> def checkNumOffspring(numOffspring, ops=[]):
...     '''Check the number of offspring for each family using
...        information field father_idx
...     '''
...     pop = sim.Population(size=[30], loci=1, infoFields=['father_idx', 'mother_idx'])
...     pop.evolve(
...         initOps=[
...             sim.InitSex(),
...             sim.InitGenotype(freq=[0.5, 0.5]),
...         ],
...         matingScheme=sim.RandomMating(ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.ParentsTagger(),
...             ] + ops,
...             numOffspring=numOffspring),
...         gen=1)
...     # get the parents of each offspring
...     parents = [(x, y) for x, y in zip(pop.indInfo('mother_idx'),
...         pop.indInfo('father_idx'))]
...     # Individuals with identical parents are considered as siblings.
...     famSize = []
...     lastParent = (-1, -1)
...     for parent in parents:
...         if parent == lastParent:
...             famSize[-1] += 1
...         else:
...             lastParent = parent
...             famSize.append(1)
...     return famSize
...
>>> # Case 1: produce the given number of offspring
>>> checkNumOffspring(numOffspring=2)
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
>>> # Case 2: Use a Python function
>>> import random
>>> def func(gen):
...     return random.randint(5, 8)
...
>>> checkNumOffspring(numOffspring=func)
[6, 7, 6, 7, 4]
```

```
>>> # Case 3: A geometric distribution
>>> checkNumOffspring(numOffspring=(sim.GEOMETRIC_DISTRIBUTION, 0.3))
[3, 1, 2, 1, 1, 1, 1, 7, 4, 6, 1, 2]
>>> # Case 4: A Possition distribution
>>> checkNumOffspring(numOffspring=(sim.POISSON_DISTRIBUTION, 1.6))
[1, 1, 1, 2, 1, 2, 2, 3, 2, 1, 1, 2, 1, 3, 2, 2, 1, 2]
>>> # Case 5: A Binomial distribution
>>> checkNumOffspring(numOffspring=(sim.BINOMIAL_DISTRIBUTION, 0.1, 10))
[2, 1, 1, 2, 1, 1, 3, 1, 1, 1, 2, 2, 2, 2, 1, 1, 2, 1, 2, 1]
>>> # Case 6: A uniform distribution
>>> checkNumOffspring(numOffspring=(sim.UNIFORM_DISTRIBUTION, 2, 6))
[2, 6, 5, 2, 2, 2, 4, 4, 3]
>>> # Case 7: With selection on offspring
>>> checkNumOffspring(numOffspring=8,
...     ops=[sim.MapSelector(loci=0, fitness={(0,0):1, (0,1):0.8, (1,1):0.5})])
[8, 6, 6, 5, 3, 2]
```

However, **the actual number of offspring can be less than specified because offspring can be discarded during mating.** More specifically, if any during-mating generator, such as a during-mating selector, returns `False` during the production of offspring, the offspring will be discarded so the total number of offspring will be reduced. This is the case in the seventh case of Example 5.7 where offspring with certain genotypes have lower probabilities to survive. If you would like to control size of families in the presence of natural selection, you could set a larger `numOffspring` use a `OffspringTagger` to mark the index of offspring, and discard offspring conditionally using operator `DiscardIf` . Please refer to example 4.64 for details.

5.1.4 Dynamic population size determined by number of offspring *

What we have described so far requires you to determine the size of offspring population in advance. Each mating event produces a number of offspring that is determined by parameter `NumOffspring`. The mating process stops when the offspring population is filled. This works for most scenarios but there are cases where the offspring population size is determined dynamically from a fixed number of mating events with random number of offspring. For example, you might design a mating scheme where all males in a population mate only once and produce random number of offspring.

These kind of mating schemes can be simulated using a demographic model that calculates offspring population size from pre-simulated number of offspring for each family. More specifically, we

- Define a demogrphic function (model) that will be called before mating happens.
- This function determines and save the number of offspring for each mating event, and return the total number of offspring as offspring population size.
- Pass a function or generator to parameter numOffspring to pass pre-determined number of offspring. This function will be called each time when number of offspring is needed.

The number of offspring could be saved and retrieved as global variable but a more clever method is to store the numbers of offspring in a demographic model (class). Example 5.8 demonstrates this method by implementing a demographic model that simulate, save, and return the number of offspring. Note that although we determine the number of mating events from number of males in the parental population, a random mating scheme will choose parents with replacement so it is likely that some parents will be chosen multiple times while some others are not chosen at all. Please refer to section "Non-random and customized mating schemes" to learn how to define a mating scheme that picks parents without replacement.

Listing 5.8: Dynamic population size determined by number of offspring

```
>>> import simuPOP as sim
>>>
>>> import random
>>>
>>> class RandomNumOff:
...     # a demographic model
...     def __init__(self):
...         self.numOff = []
...
...     def getNumOff(self):
...         # return the pre-simulated number of offspring as a generator function
...         for item in self.numOff:
...             yield item
...
...     def __call__(self, pop):
...         # define __call__ so that a RandomNumOff object is callable.
...         #
...         # Each male produce from 1 to 3 offspring. For large population, get the
...         # number of males instead of checking the sex of each individual
...         self.numOff = [random.randint(1, 3) for ind in pop.individuals() if ind.sex() == sim.MALE]
...         # return the total population size
...         print('{} mating events with number of offspring {}'.format(len(self.numOff), self.numOff))
...         return sum(self.numOff)
...
>>>
>>> pop = sim.Population(10)
>>>
>>> # create a demogranic model
>>> numOffModel = RandomNumOff()
>>>
>>> pop.evolve(
...     preOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(
...         # the model will be called before mating to deteremine
...         # family and population size
...         subPopSize=numOffModel,
...         # the getNumOff function (generator) returns number of offspring
...         # for each mating event
...         numOffspring=numOffModel.getNumOff
...     ),
...     gen=3
... )
5 mating events with number of offspring [3, 2, 3, 3, 1]
6 mating events with number of offspring [3, 3, 3, 2, 1, 2]
6 mating events with number of offspring [1, 3, 3, 3, 1, 3]
3L
>>>
```

5.1.5 Determine sex of offspring

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

simuPOP assign `Male` or `Female` to offspring with equal probability.

Other sex determination methods are also available:

- `sexMode=RANDOM_SEX`: Sex is determined randomly, with equal probability for `MALE` and `FEMALE`. This is the default mode for sexual mating schemes such as random mating.
- `sexMode=NO_SEX`: Sex is not simulated so everyone is `MALE`. This is the default mode for asexual mating schemes.
- `sexMode=(PROB_OF_MALES, prob)`: Produce males with given probability.
- `sexMode=(NUM_OF_MALES, n)`: The first `n` offspring in each family will be `Male`. If the number of offspring at a mating event is less than or equal to `n`, all offspring will be male.
- `sexMode=(NUM_OF_FEMALES, n)`: The first `n` offspring in each family will be `Female`.
- `sexMode=(SEQUENCE_OF_SEX, s1, s2, ...)`: Use sequence `s1`, `s2`, ... for offspring in each mating event.
- `sexMode=(GLOBAL_SEQUENCE_OF_SEX, s1, s2, ...)`: Use sequence `s1`, `s2`, ... for all offspring in a subpopulation. Because other mode of sex determination works within each mating event, this is the only way to ensure proportion of sex in a subpopulation. For example, `(GLOBAL_SEQUENCE_OF_SEX, MALE, FEMALE)` will gives `MALE` and `FEMALE` iteratively to all offspring, making sure there are equal number of males and females (if there are even number of offspring).
- `sexMode=func` or `sexMode=generator_func`: In this last case, a Python function or a Python generator function can be specified to provide sex to each offspring. The function is called whenever an offspring is created. The generator function is called for each subpopulation, and provides an iterator that provides sex for all offspring in a subpopulation.

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

Example 5.9 demonstrates how to use parameter `sexMode`. In this example, a function `checkSexMode` is defined. It takes a mating scheme as its input parameter and use it to evolve a population with 40 individuals. After evolving a population for one generation, sexes of all offspring are returned as a string.

Listing 5.9: Determine the sex of offspring

```
>>> import simuPOP as sim
>>> def checkSexMode(ms):
...     '''Check the assignment of sex to offspring'''
...     pop = sim.Population(size=[40])
...     pop.evolve(initOps=sim.InitSex(), matingScheme=ms, gen=1)
...     # return individual sex as a string
...     return ''.join(['M' if ind.sex() == sim.MALE else 'F' for ind in pop.individuals()])
...
>>> # Case 1: sim.NO_SEX (all male, sim.RandomMating will not continue)
>>> checkSexMode(sim.RandomMating(sexMode=sim.NO_SEX))
'MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM'
>>> # Case 2: sim.RANDOM_SEX (sim.Male/Female with probability 0.5)
>>> checkSexMode(sim.RandomMating(sexMode=sim.RANDOM_SEX))
'MFFFFFFMFFFMFFFMMFFMFFMMMFFMMFMFFFFFFFMF'
>>> # Case 3: sim.PROB_OF_MALES (Specify probability of male)
>>> checkSexMode(sim.RandomMating(sexMode=(sim.PROB_OF_MALES, 0.8)))
'MMFMMFFMMFFMMMMMMMMMFMMFFMMMMMMMMMMMMMMMM'
>>> # Case 4: sim.NUM_OF_MALES (Specify number of male in each family)
```

```
>>> checkSexMode(sim.RandomMating(numOffspring=3, sexMode=(sim.NUM_OF_MALES, 1)))
'MFFMFFMFFMFFMFFMFFMFFMFFMFFMFFMFFMFFMFFMFFM'
>>> # Case 5: sim.NUM_OF_FEMALES (Specify number of female in each family)
>>> checkSexMode(sim.RandomMating(
...     numOffspring=(sim.UNIFORM_DISTRIBUTION, 4, 6),
...     sexMode=(sim.NUM_OF_FEMALES, 2))
... )
'FFMMFFMMMFFMMFFMMMMFFMMFFMMMMFFMMFFMMFFM'
>>> # Case 6: sim.SEQUENCE_OF_SEX
>>> checkSexMode(sim.RandomMating(
...     numOffspring=4, sexMode=(sim.SEQUENCE_OF_SEX, sim.MALE, sim.FEMALE))
... )
'MFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMF'
>>> # Case 7: sim.GLOBAL_SEQUENCE_OF_SEX
>>> checkSexMode(sim.RandomMating(
...     numOffspring=3, sexMode=(sim.GLOBAL_SEQUENCE_OF_SEX, sim.MALE, sim.FEMALE))
... )
'MFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMF'
>>> # Case 8: A generator function
>>> def sexFunc():
...     i = 0
...     while True:
...         i += 1
...         if i % 2 == 0:
...             yield sim.MALE
...         else:
...             yield sim.FEMALE
...
>>> checkSexMode(sim.RandomMating(numOffspring=3, sexMode=sexFunc))
'FMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFMFM'
```

5.1.6 Monogamous mating

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

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

Example 5.10 demonstrates one scenario of using a monogamous mating scheme where sex of parents and offspring are strictly specified so that parents will not be exhausted. The sex initializer `InitSex` assigns exactly 10 males and 10 females to the initial population. Because of the use of `numOffspring=2, sexMode=(NUM_OF_MALES, 1)`, each mating event will produce exactly one male and one female. Unlike a random mating scheme that only about 80% of parents are involved in the production of an offspring population with the same size, this mating scheme makes use of all parents.

Listing 5.10: Sexual monogamous mating

```
>>> import simuPOP as sim
>>> pop = sim.Population(20, infoFields=['father_idx', 'mother_idx'])
>>> pop.evolve(
...     initOps=sim.InitSex(sex=(sim.MALE, sim.FEMALE)),
```

```
...     matingScheme=sim.MonogamousMating(
...         numOffspring=2,
...         sexMode=(sim.NUM_OF_MALES, 1),
...         ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.ParentsTagger(),
...         ],
...     ),
...     gen = 5
... )
5L
>>> [ind.sex() for ind in pop.individuals()]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
>>> [int(ind.father_idx) for ind in pop.individuals()]
[16, 16, 2, 2, 4, 4, 8, 8, 0, 0, 14, 14, 10, 10, 12, 12, 18, 18, 6, 6]
>>> [int(ind.mother_idx) for ind in pop.individuals()]
[13, 13, 17, 17, 1, 1, 15, 15, 19, 19, 9, 9, 3, 3, 5, 5, 7, 7, 11, 11]
>>> # count the number of distinct parents
>>> len(set(pop.indInfo('father_idx')))
10
>>> len(set(pop.indInfo('mother_idx')))
10
```

5.1.7 Polygamous mating

In comparison to monogamous mating, parents in a polygamous mate with more than one spouse during their life-cycle. Both *polygany* (one man has more than one wife) and *polyandry* (one woman has more than one husband) are supported.

Other than regular parameters such as `numOffspring`, mating scheme `PolygamousMating` accepts parameters `polySex` (default to `Male`) and `polyNum` (default to 1). During mating, an individual with `polySex` is selected and then mate with `polyNum` randomly selected spouse. Example 5.11 demonstrates the use of this mating schemes. Note that this mating scheme support natural selection, but does not yet handle varying `polyNum` and selection of parents without replacement.

Listing 5.11: Sexual polygamous mating

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, infoFields=['father_idx', 'mother_idx'])
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.PolygamousMating(polySex=sim.MALE, polyNum=2,
...         ops=[sim.ParentsTagger(),
...             sim.MendelianGenoTransmitter()],
...     ),
...     gen = 5
... )
5L
>>> [int(ind.father_idx) for ind in pop.individuals()][:20]
[67, 67, 42, 42, 91, 91, 25, 25, 65, 65, 47, 47, 18, 18, 16, 16, 96, 96, 57, 57]
>>> [int(ind.mother_idx) for ind in pop.individuals()][:20]
[58, 58, 58, 0, 68, 32, 37, 89, 6, 85, 12, 58, 36, 12, 66, 44, 51, 85, 60, 29]
```

5.1.8 Asexual random mating

Mating scheme `RandomSelection` implements an asexual random mating scheme. It randomly select parents from a parental population (with replacement) and copy them to an offspring generation. Both genotypes and sex of the parents are copied because genotype and sex are sometimes related. This mating scheme can be used to simulate the evolution of haploid sequences in a standard haploid Wright-Fisher model.

Example 5.12 applies a `RandomSelection` mating scheme to a haploid population with 100 sequences. A `parentTagger` is used to track the parent of each individual. Although sex information is not used in this mating scheme, Individual sexes are initialized and passed to offspring.

Listing 5.12: Asexual random mating

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, ploidy=1, loci=[5, 5], ancGen=1,
...     infoFields='parent_idx')
>>> pop.evolve(
...     initOps=sim.InitGenotype(freq=[0.3, 0.7]),
...     matingScheme=sim.RandomSelection(ops=[
...         sim.ParentsTagger(infoFields='parent_idx'),
...         sim.CloneGenoTransmitter(),
...     ]),
...     gen = 5
... )
5L
>>> ind = pop.individual(0)
>>> par = pop.ancestor(ind.parent_idx, 1)
>>> print(ind.sex(), ind.genotype())
(1, [1, 1, 0, 1, 1, 0, 1, 0, 0, 0])
>>> print(par.sex(), par.genotype())
(1, [1, 1, 0, 0, 1, 1, 1, 1, 0, 1])
```

5.1.9 Mating in haplodiploid populations

Male individuals in a haplodiploid population are derived from unfertilized eggs and thus have only one set of chromosomes. Mating in such a population is handled by a special mating scheme called `haplodiplodMating`. This mating scheme chooses a pair of parents randomly and produces some offspring. It transmit maternal chromosomes and paternal chromosomes (the only copy) to female offspring, and only maternal chromosomes to male offspring. Example 5.13 demonstrates how to use this mating scheme. It uses three initializers because sex has to be initialized before two other intializers can initialize genotype by sex.

Listing 5.13: Random mating in haplodiploid populations

```
>>> import simuPOP as sim
>>> pop = sim.Population(10, ploidy=sim.HAPLODIPLOID, loci=[5, 5],
...     infoFields=['father_idx', 'mother_idx'])
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[0]*10, subPops=[(0, 'Male')]),
...         sim.InitGenotype(genotype=[1]*10+[2]*10, subPops=[(0, 'Female')])
...     ],
...     preOps=sim.Dumper(structure=False),
...     matingScheme=sim.HaplodiploidMating(
```

```
...         ops=[sim.HaplodiploidGenoTransmitter(), sim.ParentsTagger()]),
...     postOps=sim.Dumper(structure=False),
...     gen = 1
... )
SubPopulation 0 (), 10 Individuals:
   0: FU 11111 11111 | 22222 22222 |  0 0
   1: FU 11111 11111 | 22222 22222 |  0 0
   2: MU 00000 00000 | _____ _____ |  0 0
   3: MU 00000 00000 | _____ _____ |  0 0
   4: MU 00000 00000 | _____ _____ |  0 0
   5: MU 00000 00000 | _____ _____ |  0 0
   6: MU 00000 00000 | _____ _____ |  0 0
   7: FU 11111 11111 | 22222 22222 |  0 0
   8: FU 11111 11111 | 22222 22222 |  0 0
   9: FU 11111 11111 | 22222 22222 |  0 0

SubPopulation 0 (), 10 Individuals:
   0: MU 11111 11111 | _____ _____ |  4 9
   1: MU 11111 22222 | _____ _____ |  4 8
   2: MU 22222 11111 | _____ _____ |  6 8
   3: MU 22222 11111 | _____ _____ |  3 8
   4: MU 22222 22222 | _____ _____ |  2 8
   5: MU 22222 22222 | _____ _____ |  6 9
   6: FU 22222 22222 | 00000 00000 |  2 1
   7: FU 22222 22222 | 00000 00000 |  2 1
   8: FU 22222 22222 | 00000 00000 |  3 9
   9: FU 11111 11111 | 00000 00000 |  5 8

1L
```

Note that this mating scheme does not support recombination and the standard Recombinator does not work with haplodiploid populations. Please refer to the next Chapter for how to define a customized genotype transmitter to handle such a situation.

5.1.10 Self-fertilization

Some plant populations evolve through self-fertilization. That is to say, a parent fertilizes with itself during the production of offspring (seeds). In a `SelfMating` mating scheme, parents are chosen randomly (one at a time), and are used twice to produce two homologous sets of offspring chromosomes. The standard Recombinator can be used with this mating scheme. Example 5.14 initializes each chromosome with different alleles to demonstrate how these alleles are transmitted in this population.

Listing 5.14: Selfing mating scheme

```
>>> import simuPOP as sim
>>> pop = sim.Population(20, loci=8)
>>> # every chromosomes are different. :-)
>>> for idx, ind in enumerate(pop.individuals()):
...     ind.setGenotype([idx*2], 0)
...     ind.setGenotype([idx*2+1], 1)
...
>>> pop.evolve(
...     matingScheme=sim.SelfMating(ops=sim.Recombinator(rates=0.01)),
...     gen = 1
```

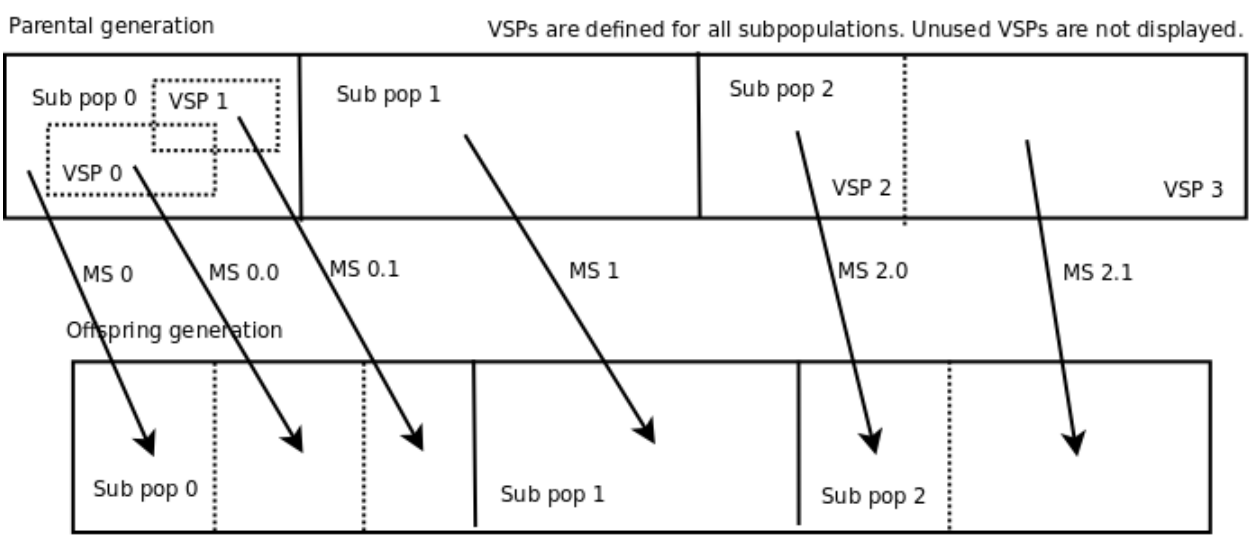
```
... )
1L
>>> sim.dump(pop, width=3, structure=False, max=10)
SubPopulation 0 (), 20 Individuals:
   0: FU  36 36 36 36 36 36 36 36 |  36 36 36 36 36 36 36 36
   1: FU   6  6  6  6  6  6  6  6 |   7  7  7  7  7  7  7  7
   2: MU  33 33 33 33 33 33 33 33 |  33 33 33 33 33 33 33 33
   3: MU  22 22 22 22 22 23 23 23 |  22 22 22 22 22 22 22 22
   4: FU  27 27 27 27 27 27 27 27 |  27 27 27 27 27 27 27 27
   5: MU  26 26 26 26 26 26 26 26 |  26 26 26 26 26 26 26 26
   6: MU  39 39 39 39 39 39 39 39 |  38 38 38 38 38 38 38 38
   7: MU  10 10 10 10 10 10 10 10 |  11 11 11 11 11 11 11 11
   8: MU  11 11 11 11 11 11 11 11 |  11 11 11 11 11 11 11 11
   9: FU  24 24 24 24 24 24 24 24 |  25 25 25 25 25 25 25 25
```

5.1.11 Heterogeneous mating schemes *

Different groups of individuals in a population may have different mating patterns. For example, individuals with different properties can have varying fecundity, represented by different numbers of offspring generated per mating event. This can be extended to aged populations in which only adults (may be defined by age > 20 and age < 40) can produce offspring, where other individuals will either be copied to the offspring generation or die.

A heterogeneous mating scheme (`HeteroMating`) accepts a list of mating schemes that are applied to different subpopulation or virtual subpopulations. If multiple mating schemes are applied to the same subpopulation, each of them only populate part of the offspring subpopulation. This is illustrated in Figure 5.2.

Figure 5.2: Illustration of a heterogeneous mating scheme



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

For example, Example 5.15 applies two random mating schemes to two subpopulations. The first mating scheme produces two offspring per mating event, and the second mating scheme produces four.

Listing 5.15: Applying different mating schemes to different subpopulations

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000, 1000], loci=2,
...     infoFields=['father_idx', 'mother_idx'])
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.HeteroMating([
...         sim.RandomMating(numOffspring=2, subPops=0,
...             ops=[sim.MendelianGenoTransmitter(), sim.ParentsTagger()]
...         ),
...         sim.RandomMating(numOffspring=4, subPops=1,
...             ops=[sim.MendelianGenoTransmitter(), sim.ParentsTagger()]
...         )
...     ]),
...     gen=10
... )
10L
>>> [int(ind.father_idx) for ind in pop.individuals(0)][:10]
[134, 134, 451, 451, 780, 780, 443, 443, 457, 457]
>>> [int(ind.father_idx) for ind in pop.individuals(1)][:10]
[1978, 1978, 1978, 1978, 1582, 1582, 1582, 1582, 1322, 1322]
```

The real power of heterogeneous mating schemes lies on their ability to apply different mating schemes to different virtual subpopulations. For example, due to different micro-environmental factors, plants in the same population may exercise both self and cross-fertilization. Because of the randomness of such environmental factors, it is difficult to divide a population into self and cross-mating subpopulations. Applying different mating schemes to groups of individuals in the same subpopulation is more appropriate.

Example 5.16 applies two mating schemes to two VSPs defined by proportions of individuals. In this mating scheme, 20% of individuals go through self-mating and 80% of individuals go through random mating. This can be seen from the parental indexes of individuals in the offspring generation: individuals whose `mother_idx` are `-1` are genetically only derived from their fathers.

It might be surprising that offspring resulted from two mating schemes mix with each other so the same VSPs in the next generation include both selfed and cross-fertilized offspring. If this not desired, you can set parameter `shuffleOffspring=False` in `HeteroMating`(). Because the number of offspring that are produced by each mating scheme is proportional to the size of parental (virtual) subpopulation, the first 20% of individuals that are produced by self-fertilization will continue to self-fertilize.

Listing 5.16: Applying different mating schemes to different virtual subpopulations

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000], loci=2,
...     infoFields=['father_idx', 'mother_idx'])
>>> pop.setVirtualSplitter(sim.ProportionSplitter([0.2, 0.8]))
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.HeteroMating(matingSchemes=[
...         sim.SelfMating(subPops=[(0, 0)],
...             ops=[sim.SelfingGenoTransmitter(), sim.ParentsTagger()]
...         ),
...         sim.RandomMating(subPops=[(0, 1)],
...             ops=[sim.SelfingGenoTransmitter(), sim.ParentsTagger()]
...         )
...     ]),
...     gen = 10
```

```
... )
10L
>>> [int(ind.father_idx) for ind in pop.individuals(0)][:15]
[789, 666, 145, 125, 681, 183, 727, 308, 392, 11, 183, 223, 208, 29, 309]
>>> [int(ind.mother_idx) for ind in pop.individuals(0)][:15]
[370, 272, -1, 520, 121, 91, 220, 519, 101, 271, -1, 263, 663, -1, 286]
```

Because there is no restriction on the choice of VSPs, mating schemes can be applied to overlapped (virtual) subpopulations. For example,

```
HeteroMating(
    matingSchemes = [
        SelfMating(subPops=[(0, 0)]),
        RandomMating(subPops=0)
        ]
)
```

will apply SelfMating to the first 20% individuals, and RandomMating will be applied to all individuals. Similarly,

```
HeteroMating(
    matingSchemes = [
        SelfMating(subPops=0),
        RandomMating(subPops=0)
        ]
)
```

will allow all individuals to be involved in both `SelfMating` and `RandomMating`.

This raises the question of how many offspring each mating scheme will produce. By default, the number of offspring produced will be proportional to the size of parental (virtual) subpopulations. In the last example, because both mating schemes are applied to the same subpopulation, half of all offspring will be produced by selfing and the other half will be produced by random mating.

This behavior can be changed by a weighting scheme controlled by parameter `weight` of each homogeneous mating scheme. Briefly speaking, a positive weight will be compared against other mating schemes. a negative weight is considered proportional to the existing (virtual) subpopulation size. Negative weights are considered before positive or zero weights.

This weighting scheme is best explained by an example. Assuming that there are three mating schemes working on the same parental subpopulation

- Mating scheme A works on the whole subpopulation of size 1000
- Mating scheme B works on a virtual subpopulation of size 500
- Mating scheme C works on another virtual subpopulation of size 800

Assuming the corresponding offspring subpopulation has N individuals,

- If all weights are 0, the offspring subpopulation is divided in proportion to parental (virtual) subpopulation sizes. In this example, the mating schemes will produce $\frac{10}{23}N$, $\frac{5}{23}N$, $\frac{8}{23}N$ individuals respectively.
- If all weights are negative, they are multiplied to their parental (virtual) subpopulation sizes. For example, weight (-1, -2, -0.5) will lead to sizes (1000, 1000, 400) in the offspring subpopulation. If $N \neq 2400$ in this case, an error will be raised.
- If all weights are positive, the number of offspring produced from each mating scheme is proportional to these weights. For example, weights (1, 2, 3) will lead to $\frac{1}{6}N$, $\frac{2}{6}N$, $\frac{1}{3}N$ individuals respectively. In this case, 0 weights will produce no offspring.

- If there are mixed positive and negative weights, the negative weights are processed first, and the rest of the individuals are divided using non-negative weights. For example, three mating schemes with weights (-0.5, 2, 3) will produce 500, $\frac{2}{5}(N-500)$, $\frac{3}{5}(N-500)$ individuals respectively.

The last case is demonstrated in Example 5.17 where three random mating schemes are applied to subpopulation `0`, virtual subpopulation `(0, 0)` and virtual subpopulation `(0, 1)`, with weights `-0.5`, `2`, and `3` respectively. This example uses an advanced features that will be described in the next section. Namely, three during-mating Python operators are passed to each mating scheme to mark their offspring with different numbers.

Listing 5.17: A weighting scheme used by heterogeneous mating schemes.

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000], loci=2,
...     infoFields='mark')
>>> pop.setVirtualSplitter(sim.RangeSplitter([[0, 500], [200, 1000]]))
>>>
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.HeteroMating([
...         sim.RandomMating(subPops=0, weight=-0.5,
...             ops=[sim.InfoExec('mark=0'), sim.MendelianGenoTransmitter()]),
...         sim.RandomMating(subPops=[(0, 0)], weight=2,
...             ops=[sim.InfoExec('mark=1'), sim.MendelianGenoTransmitter()]),
...         sim.RandomMating(subPops=[(0, 1)], weight=3,
...             ops=[sim.InfoExec('mark=2'), sim.MendelianGenoTransmitter()])
...     ]),
...     gen = 10
... )
10L
>>> marks = list(pop.indInfo('mark'))
>>> marks.count(0.)
500
>>> marks.count(1.)
200
>>> marks.count(2.)
300
```

5.1.12 Conditional mating schemes

A `ConditionalMating` mating scheme allows you to apply different mating schemes to populations with different properties. The condition can be a constant (True or False), an expression that will be evaluated in the local namspace of the parental population, or a function that can take parental population as its input paramter (with parameter name `pop`).

Using variable `rep` and `gen` in the local namespace of the parental population, we can use this mating scheme to apply different mating schemes to different replicates and/or at different generations. For example, 5.18 simulates the evolution of three replicates. The first replicate uses regular mating scheme, the third replicate uses a mating scheme that produces 70% of males, and the second replicate do this only for the first 5 generations. Because there are three cases, a nested `ConditionalMating` is used.

Listing 5.18: Apply different mating schemes for different replicates at different generations

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(1000, loci=[10]), rep=3)
>>> simu.evolve(
```

```
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.ConditionalMating('rep == 0',
...         # the first replicate use standard random mating
...         sim.RandomMating(),
...         sim.ConditionalMating('rep == 1 and gen >= 5',
...             # the second replicate produces more males for the first 5 generations
...             sim.RandomMating(),
...             # the last replicate produces more males all the time
...             sim.RandomMating(sexMode=(sim.PROB_OF_MALES, 0.7))
...             )
...         ),
...     postOps=[
...         sim.Stat(numOfMales=True),
...         sim.PyEval("'gen=%d' % gen", reps=0),
...         sim.PyEval(r"'\t%d' % numOfMales"),
...         sim.PyOutput('\n', reps=-1)
...     ],
...     gen=10
... )
gen=0   477     686     718
gen=1   477     689     698
gen=2   519     692     713
gen=3   479     709     704
gen=4   539     710     688
gen=5   496     482     698
gen=6   489     488     701
gen=7   495     508     715
gen=8   497     488     688
gen=9   528     498     698
(10L, 10L, 10L)
```

A function can be passed as the condition of a `ConditionalMating` mating scheme. This allows you to apply operators such as `Stat` to examine the condition of populations more closely and determine which mating scheme to use.

5.2 Simulator

A simuPOP simulator evolves one or more copies of a population forward in time, subject to various operators. Although a population could evolve by itself using function `Population.evolve`, a simulator with one replicate is actually used.

5.2.1 Add, access and remove populations from a simulator

A simulator could be created by one or more replicates of a list of populations. For example, you could create a simulator from five replicates of a population using

```
Simulator(pop, rep=5)
```

or from a list of populations using

```
Simulator([pop, pop1, pop2])
```

. `pop`, `pop1` and `pop2` do not have to have the same genotypic structure. In order to avoid duplication of potentially large populations, a population is by default *stolen* after it is used to create a simulator. If you would like to keep the populations, you could set parameter `stealPops` to `False` so that the populations will be copied to the simulator. Populations in a simulator can be added or removed using functions `Simulator.add()` and `Simulator.extract(idx)`.

When a simulator is created, you can access populations in this simulator using function `Simulator.population(idx)` or iterate through all populations using function `Simulator.populations()`. These functions return references to the populations so that you can access populations. Modifying these references will change the corresponding populations within the simulator. The references will become invalid once the simulator object is destoryed.

Example 5.19 demonstrates different ways to create a simulator and how to access populations within it.

Listing 5.19: Create a simulator and access populations

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=10)
>>> # five copies of the same population
>>> simu = sim.Simulator(pop, rep=5)
>>> simu.numRep()
5L
>>> # evolve for ten generations and save the populations
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.3, 0.7])
...     ],
...     matingScheme=sim.RandomMating(),
...     finalOps=sim.SavePopulation('!"pop%d.pop"%rep'),
...     gen=10
... )
(10L, 10L, 10L, 10L, 10L)
>>> # load the population and create another Simulator
>>> simu = sim.Simulator([sim.loadPopulation('pop%d.pop' % x) for x in range(5)])
>>> # continue to evolve
>>> simu.evolve(
...     matingScheme=sim.RandomMating(),
...     gen=10
... )
(10L, 10L, 10L, 10L, 10L)
>>> # print out allele frequency
>>> for pop in simu.populations():
...     sim.stat(pop, alleleFreq=0)
...     print('%.2f' % pop.dvars().alleleFreq[0][0])
...
0.36
0.30
0.28
0.01
0.11
>>> # get a population
>>> pop = simu.extract(0)
>>> simu.numRep()
4L
```

5.2.2 Number of generations to evolve

A simulator usually evolves a specific number of generations according to parameter `gen` of the `evolve` function. A generation number is used to track the number of generations a simulator has evolved. Because a new population has generation number 0, a population would be at the beginning of generation n after it evolves n generations. The generation number would increase if the simulator continues to evolve. During evoluting, variables `rep` (replicate number) and `gen` (current generation number) are set to each population's local namespace.

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 the `gen` parameter) 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 use a `TerminateIf` operator to terminate the evolution according to the value of an expression. Example 5.20 demonstrates the use of such a terminator, which terminates the evolution of a population if allele 0 at locus 5 is fixed or lost. It also shows the application of an interesting operator `IfElse`, which applies an operator, in this case `PyEval`, only when an expression returns `True`. Note that this example calls the `simulator.evolve` function twice. The first call does not specify a mating scheme so a default empty mating scheme (`MatingScheme`) that does not transmit genotype is used. Populations start from the beginning of the fifth generation when the second `simulator.evole` function is called.

The generation number is stored in each Population using population variable `gen`.You can access these numbers from a simulator using function `Simulator.dvars(idx)` or from a population using function `Population.dvars()`. If needed, **you can reset generation numbers by changing these variables.**

Listing 5.20: Generation number of a simulator

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(50, loci=[10], ploidy=1),
...     rep=3)
>>> simu.evolve(gen = 5)
(5L, 5L, 5L)
>>> simu.dvars(0).gen
5
>>> simu.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 in rep %d at gen %d\n' % (rep, gen)")),
...         sim.IfElse('alleleNum[5][0] == 50',
...             sim.PyEval(r"'Allele 0 is fixed in rep %d at gen %d\n' % (rep, gen)")),
...         sim.TerminateIf('len(alleleNum[5]) == 1'),
...     ],
... )
Allele 0 is fixed in rep 2 at gen 29
Allele 0 is fixed in rep 1 at gen 74
Allele 0 is lost in rep 0 at gen 120
(116L, 70L, 25L)
>>> [simu.dvars(x).gen for x in range(3)]
[121, 75, 30]
```

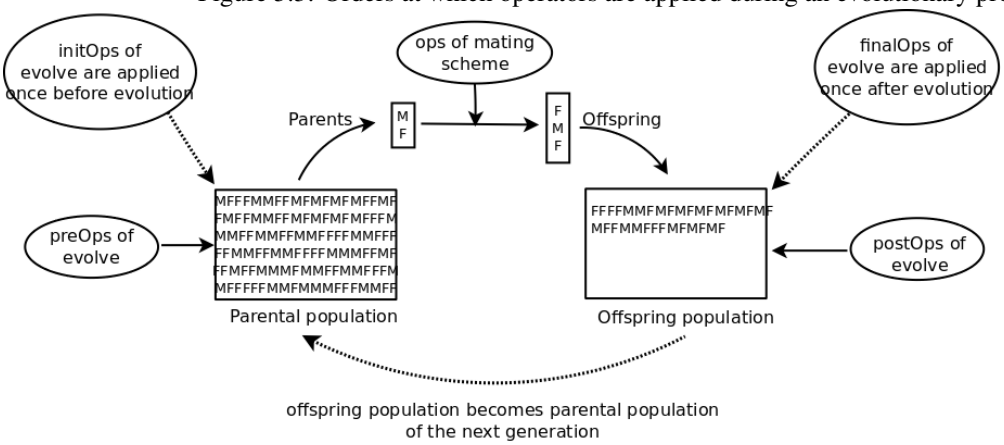
5.2.3 Evolve populations in a simulator

There are a number of rules about when and how operators are applied during the evolution of a population. In summary, in the order at which operators are processed and applied,

- Operators specified in parameter `initOps` of function `Simulator.evolve` will be applied to the initial population before evolution, subject to replicate applicability restraint specified by parameter `reps`.
- Operators specified in parameter `preOps` of function `Simulator.evolve` will be applied to the parental population at each generation, subject to replicate and generation applicability restraint specified by parameters `begin`, `end`, `step`, `at`, and `reps`.
- During-mating operators specified in the `ops` parameter of a mating scheme will be called during mating to transmit genotype (and possibly information fields etc) from parental to offspring, subject to replicate and generation applicability restraint specified by parameters `begin`, `end`, `step`, `at`, and `reps`.
- Operators specified in parameter postOps of function `Simulator.evolve` will be applied to the offspring population at each generation, subject to replicate and generation applicability restraint specified by parameters `begin`, `end`, `step`, `at`, and `reps`.
- Operators specified in parameter `finalOps` of function `Simulator.evolve` will be applied to the final population after evolution, subject to replicate applicability restraint specified by parameter `reps`.

Figure 5.3 illustrated how operators are applied to an evolutionary process. It worth noting that a default during-mating operator is defined for each mating scheme. User-specfied operators will **replace** the default operator so you need to explicitly specify the default operator if you intent to add another one.

Figure 5.3: Orders at which operators are applied during an evolutionary process



If you suspect that your simulation is not running as expected, you can have a close look at your evolutionary process by setting the `dryrun` parameter of an `evolve` function to `True`, or by calling function `describeEvolProcess()`. This function takes the same set of parameters as `Simulator.evolve()` and returns a description of the evolution process, which might help you identify misuse of operators.

Listing 5.21: describe an evolutionary process

```
>>> import simuPOP as sim
>>>
>>> def outputstat(pop):
...     'Calculate and output statistics, ignored'
...     return True
...
>>> # describe this evolutionary process
>>> print(sim.describeEvolProcess(
...     initOps=[
...         sim.InitSex(),
...         sim.InitInfo(lambda: random.randint(0, 75), infoFields='age'),
```

```
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         sim.IdTagger(),
...         sim.PyOutput('Prevalence of disease in each age group:\n'),
...     ],
...     preOps=sim.InfoExec('age += 1'),
...     matingScheme=sim.HeteroMating([
...         sim.CloneMating(subPops=[(0,0), (0,1), (0,2)], weight=-1),
...         sim.RandomMating(ops=[
...             sim.IdTagger(),
...             sim.Recombinator(intensity=1e-4)
...         ], subPops=[(0,1)]),
...     ]),
...     postOps=[
...         sim.MaPenetrance(loci=0, penetrance=[0.01, 0.1, 0.3]),
...         sim.PyOperator(func=outputstat)
...     ],
...     gen = 100,
...     numRep = 3
... ))
Replicate 0 1 2:
Apply pre-evolution operators to the initial population (initOps).
   * <simuPOP.InitSex> initialize sex randomly
   * <simuPOP.InitInfo> initialize information field age using a Python
     function <lambda>
   * <simuPOP.InitGenotype> initialize individual genotype acccording to
     allele frequencies.
   * <simuPOP.IdTagger> assign an unique ID to individuals
   * <simuPOP.PyOutput> write 'Prevalence of disease in each age group:... '
     to output

Evolve a population for 100 generations
   * Apply pre-mating operators to the parental generation (preOps)
      # <simuPOP.InfoExec> execute statement age += 1 using information fields
        as variables.

   * Populate an offspring populaton from the parental population using mating
     scheme <simuPOP.HeteroMating> a heterogeneous mating scheme with 2
     homogeneous mating schemes:
      # <simuPOP.HomoMating> a homogeneous mating scheme that uses
         - <simuPOP.SequentialParentChooser> chooses a parent sequentially
         - <simuPOP.OffspringGenerator> produces offspring using operators
            . <simuPOP.CloneGenoTransmitter> clone genotype, sex and
              information fields of parent to offspring
        in subpopulations (0, 0), (0, 1), (0, 2).
      # <simuPOP.HomoMating> a homogeneous mating scheme that uses
         - <simuPOP.RandomParentsChooser> chooses two parents randomly
         - <simuPOP.OffspringGenerator> produces offspring using operators
            . <simuPOP.IdTagger> assign an unique ID to individuals
            . <simuPOP.Recombinator> genetic recombination.
        in subpopulations (0, 1).


   * Apply post-mating operators to the offspring population (postOps).
      # <simuPOP.MaPenetrance> multiple-alleles penetrance
```

```
      # <simuPOP.PyOperator> calling a Python function outputstat

No operator is applied to the final population (finalOps).
```

5.3 Non-random and customized mating schemes *

5.3.1 The structure of a homogeneous mating scheme *

A *homogeneous mating scheme* (`HomoMating`) populates an offspring generation as follows:

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

To define a homogeneous mating scheme, you will need to provide a `chooser` (a *parent chooser* that is responsible for choosing one or two parents from the parental generation) and a `generator` (an *offspring generator* that is responsible for generating a number of offspring from the chosen parents). For example, a `selfingMating` mating scheme uses a `RandomParentChooser` to choose a parent randomly from a population, possibly according to individual fitness, it uses a standard `OffspringGenerator` that uses a `selfingOffspringGenerator` to transmit genotype. The constructor of `HomoMating` also accepts parameters `subPopSize` (parameter to control offspring subpopulation sizes), `subPops` (applicable subpopulatiosn or virtual subpopulations), and `weight` (weighting parameter when used in a heterogeneous mating scheme). When this mating scheme is applied to the whole population, `subPopSize` is used to determine the subpopulation sizes of the offspring generation (see Section 5.1.1 for details), parameters `subPops` and `weight` are ignored. Otherwise, the number of offspring this mating scheme will produce is determined by the heterogeneous mating scheme.

Example 5.22 demonstrates how the most commonly used mating scheme, the diploid sexual `RandomMating` mating scheme is defined in `simuPOP.py`. This mating scheme uses a `RandomParentsChooser` with replacement, and a standard `OffspringGenerator` using a default `MendelianGenoTransmitter`.

Listing 5.22: Define a random mating scheme

```
def RandomMating(numOffspring=1., sexMode=RANDOM_SEX,
        ops=MendelianGenoTransmitter(), subPopSize=[],
        subPops=ALL_AVAIL, weight=0, selectionField='fitness'):
    'A basic diploid sexual random mating scheme.'
    return HomoMating(
        chooser=RandomParentsChooser(True, selectionField),
        generator=OffspringGenerator(ops, numOffspring, sexMode),
        subPopSize=subPopSize,
        subPops=subPops,
        weight=weight)
```

Different parent choosers and offspring generators can be combined to define a large number of homogeneous mating schemes. Some of the parent choosers return one parent so they work with offspring generators that need one parent (e.g. selfing or clone offspring generator); some of the parent choosers return two parents so they work with offspring

generators that need two parents (e.g. Mendelian offspring generator). For example, the standard `SelfMating` mating scheme uses a `RandomParentChooser` but you can easily use a `SequentialParentChooser` to choose parents sequentially and self-fertilize parents one by one. This is demonstrated in Example 5.23.

Listing 5.23: Define a sequential selfing mating scheme

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=5*3, infoFields='parent_idx')
>>> pop.evolve(
...     initOps=[sim.InitGenotype(freq=[0.2]*5)],
...     preOps=sim.Dumper(structure=False, max=5),
...     matingScheme=sim.HomoMating(
...         sim.SequentialParentChooser(),
...         sim.OffspringGenerator(ops=[
...             sim.SelfingGenoTransmitter(),
...             sim.ParentsTagger(infoFields='parent_idx'),
...         ])
...     ),
...     postOps=sim.Dumper(structure=False, max=5),
...     gen = 1
... )
SubPopulation 0 (), 100 Individuals:
   0: MU 441000142224423 | 431303440010114 |  0
   1: MU 334442443034342 | 113203441333201 |  0
   2: MU 034344042424240 | 344304121430212 |  0
   3: MU 132322330420043 | 141300223114240 |  0
   4: MU 111123040033342 | 344344221133120 |  0

SubPopulation 0 (), 100 Individuals:
   0: MU 441000142224423 | 431303440010114 |  0
   1: FU 334442443034342 | 113203441333201 |  1
   2: MU 344304121430212 | 034344042424240 |  2
   3: FU 141300223114240 | 132322330420043 |  3
   4: FU 344344221133120 | 111123040033342 |  4

1L
```

5.3.2 Offspring generators *

An `OffspringGenerator` accepts a parameters `ops` (a list of during-mating operators), `numOffspring` (control number of offspring per mating event) and `sexMode` (control offspring sex). We have examined the last two parameters in detail in sections 5.1.3 and 5.1.5.

The most tricky parameter is the `ops` parameter. It accepts a list of during mating operators that are used to transmit genotypes from parent(s) to offspring and/or set individual information fields. The standard `OffspringGenerator` does not have any default operator so no genotype will be transmitted by default. All stock mating schemes use a default genotype transmitter. (e,g, a `MendelianGenoTransmitter` in Example 5.22 is passed to the offspring generator used in `RandomMating`). Note that you need to specify all needed operators if you use parameter `ops` to change the operators used in a mating scheme (see Example 5.17). That is to say, you can use `ops=Recombinator()` to replace a default `MendelianGenoTransmitter()`, but you have to use `ops=[IdTagger(), MendelianGenoTransmitter()]` if you would like to add a during-mating operator to the default one.

Another offspring generator is provided in simuPOP. This `ControlledOffspringGenerator` is used to control an evolutionary process so that the allele frequencies at certain loci follows some pre-simulated *frequency trajectories*. Please

refer to Peng et al. [2007] for rationals behind such an offspring generator and its applications in the simulation of complex human diseases.

Example 5.24 demonstrates the use of such a controlled offspring generator. Instead of using a realistic frequency trajectory function, it forces allele frequency at locus 5 to increase linearly. In contrast, the allele frequency at locus 15 on the second chromosome oscillates as a result of genetic drift. Note that the random mating version of this mating scheme is defined in simuPOP as `ControlledRandomMating`.

Listing 5.24: A controlled random mating scheme

```
>>> import simuPOP as sim
>>> def traj(gen):
...     return [0.5 + gen * 0.01]
...
>>> pop = sim.Population(1000, loci=[10]*2)
>>> # evolve the sim.Population while keeping allele frequency 0.5
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.HomoMating(sim.RandomParentChooser(),
...         sim.ControlledOffspringGenerator(loci=5,
...             alleles=[0], freqFunc=traj,
...             ops = sim.SelfingGenoTransmitter())),
...     postOps=[
...         sim.Stat(alleleFreq=[5, 15]),
...         sim.PyEval(r'"%.2f\t%.2f\n" % (alleleFreq[5][0], alleleFreq[15][0])')
...     ],
...     gen = 5
... )
0.50    0.51
0.51    0.51
0.52    0.51
0.53    0.52
0.54    0.54
5L
```

5.3.3 Genotype transmitters *

Although any during mating operators can be used in parameter `ops` of an offspring generator, those that transmit genotype from parents to offspring are customarily called **genotype transmitters**. simuPOP provides a number of genotype transmitters including clone, Mendelian, selfing, haplodiploid, genotype transmitter, and a Recombinator. They are usually used implicitly in a mating scheme, but they can also be used explicitly.

Although simuPOP provides a number of genotype transmitters, they may still be cases where customized genotype transmitter is needed. For example, a Recombinator can be used to recombine parental chromosomes but it is well known that male and female individuals differ in recombination rates. How can you apply two different Recombinators to male and female Individuals separately?

An immediate thought can be the use of virtual subpopulations. If you apply two random mating schemes to two virtual subpopulations defined by sex, `RandomParentsChooser` will not work because no opposite sex can be found in each virtual subpopulation. In this case, a customized genotype transmitter can be used.

A customized genotype transmitter is only a Python during-mating operator. Although it is possible to define a function and use a PyOperator directly (Example 4.98), it is much better to derive an operator from PyOperator, as the case in

Example 4.101.

Example 5.25 defines a `sexSpecificRecombinator` that uses, internally, two different Recombinators to recombine male and female parents. The key statement is the `PyOperator.__init__` line which initializes a Python operator with given function `self.transmitGenotype`. Example 5.25 outputs the population in two generations. You should notice that paternal chromosome are not recombined when they are transmitted to offspring.

Listing 5.25: A customized genotype transmitter for sex-specific recombination

```
>>> from simuPOP import *
>>> class sexSpecificRecombinator(PyOperator):
...     def __init__(self, intensity=0, rates=0, loci=[], convMode=NO_CONVERSION,
...             maleIntensity=0, maleRates=0, maleLoci=[], maleConvMode=NO_CONVERSION,
...             *args, **kwargs):
...         # This operator is used to recombine maternal chromosomes
...         self.Recombinator = Recombinator(rates, intensity, loci, convMode)
...         # This operator is used to recombine paternal chromosomes
...         self.maleRecombinator = Recombinator(maleRates, maleIntensity,
...             maleLoci, maleConvMode)
...         #
...         PyOperator.__init__(self, func=self.transmitGenotype, *args, **kwargs)
...     #
...     def transmitGenotype(self, pop, off, dad, mom):
...         # Form the first homologous copy of offspring.
...         self.Recombinator.transmitGenotype(mom, off, 0)
...         # Form the second homologous copy of offspring.
...         self.maleRecombinator.transmitGenotype(dad, off, 1)
...         return True
...
>>> pop = Population(10, loci=[15]*2, infoFields=['father_idx', 'mother_idx'])
>>> pop.evolve(
...     initOps=[
...         InitSex(),
...         InitGenotype(freq=[0.4] + [0.2]*3)
...     ],
...     matingScheme=RandomMating(ops=[
...         sexSpecificRecombinator(rates=0.1, maleRates=0),
...         ParentsTagger()
...     ]),
...     postOps=Dumper(structure=False),
...     gen = 2
... )
SubPopulation 0 (), 10 Individuals:
   0: FU 230000130212000 130110020112120 | 310300000030330 000113003202000 |  6 7
   1: FU 110100000002000 223313300111002 | 331311301000220 002330110020020 |  6 7
   2: MU 230301121003012 032010332330303 | 303303022100031 310232031321031 |  5 0
   3: MU 103001320130222 031300110100023 | 303303022100031 003000012020002 |  5 9
   4: FU 210230113000000 231111000121000 | 303303022100031 003000012020002 |  5 8
   5: MU 322030133101023 110323303020211 | 322111021000001 301200303300133 |  2 8
   6: MU 210230113000000 231111000121000 | 331303300011323 310232031321031 |  5 8
   7: FU 200331312001001 200011203020203 | 031032120003212 101032020302120 |  3 1
   8: FU 230000130212000 223313300111002 | 303303022100031 003000012020002 |  5 7
   9: FU 200331312001001 130301011230300 | 322111021000001 320103032303101 |  2 1

SubPopulation 0 (), 10 Individuals:
   0: MU 230000130212000 223313300111002 | 322030133101023 301200303300133 |  5 8
```

```
  1: MU 230000130212000 130110020112120 | 303303022100031 310232031321031 |  2 0
  2: FU 303303022100031 003000012020002 | 322111021000001 301200303300133 |  5 4
  3: FU 331311301000220 223313300111002 | 322111021000001 110323303020211 |  5 1
  4: MU 200331312001001 101032020302120 | 230301121003012 032010332330303 |  2 7
  5: FU 031032120003212 200011203020203 | 103001320130222 031300110100023 |  3 7
  6: FU 200331312001001 320103032303101 | 303303022100031 032010332330303 |  2 9
  7: FU 200331312001001 320103032303101 | 303303022100031 310232031321031 |  2 9
  8: FU 200331312001001 130301011230300 | 303303022100031 031300110100023 |  3 9
  9: MU 303303022100031 003000012020002 | 210230113000000 231111000121000 |  6 4

2L
```

5.3.4 A Python parent chooser *

Parent choosers are responsible for choosing one or two parents from a parental (virtual) subpopulation. simuPOP defines a few parent choosers that choose parent(s) sequentially, randomly (with or without replacement), or with additional conditions. Some of these parent choosers support natual selection. We have seen sequential and random parent choosers in Examples 5.23 and 5.24. Please refer to the simuPOP reference manual for details about these objects.

A parent choosing scheme can be quite complicated in reality. For example, salamanders along a river may mate with their neighbors and form several subspecies. This behavior cannot be readily simulated using any pre-define parent choosers so a hybrid parent chooser `PyParentsChooser()` should be used.

A `PyParentsChooser` accepts a user-defined Python generator function, instead of a normal python function, that returns a parent, or a pair of parents repeatedly. Briefly speaking, when a generator function is called, it returns a *generator* object that provides an iterator interface. Each time when this iterator iterates, this function resumes where it was stopped last time, executes and returns what the next *yield* statement returns. For example, example 5.26 defines a function that calculate $f(k) = \sum_{i=1}^{k} \frac{1}{i}$ for $k = 1, ..., 5$. It does not calculate each $f(k)$ repeatedly but returns $f(1)$, $f(2)$, ... sequentially.

Listing 5.26: A sample generator function

```
>>> import simuPOP as sim
>>> def func():
...     i = 1
...     all = 0
...     while i <= 5:
...         all += 1./i
...         i += 1
...         yield all
...
>>> for i in func():
...     print('%.3f' % i)
...
1.000
1.500
1.833
2.083
2.283
```

A `PyParentsChooser` accepts a parent generator function, which takes a population and a subpopulation index as parameters. When this parent chooser is applied to a subpopulation, it will call this generator function and ask the generated generator object repeated for either a parent, or a pair of parents (*references to individual objects or indexes relative to*

a subpopulation). Note that `PyParentsChooser` does not support virtual subpopulation but you can mimic the effect by returning only parents from certain virtual subpopulations.

Example 5.27 implements a hybrid parent chooser that chooses parents with equal social status (`rank`). In this parent chooser, all males and females are categorized by their sex and social status. A parent is chosen randomly, and then his/her spouse is chosen from females/males with the same social status. The rank of their offspring can increase or decrease randomly. It becomes obvious now that whereas a python function can return random male/female pair, the generator interface is much more efficient because the identification of sex/status groups is done only once.

Listing 5.27: A hybrid parent chooser that chooses parents by their social status

```
>>> import simuPOP as sim
>>> from random import randint
>>> def randomChooser(pop, subPop):
...     males = []
...     females = []
...     # identify males and females in each social rank
...     for rank in range(3):
...         males.append([x for x in pop.individuals(subPop) \
...             if x.sex() == sim.MALE and x.rank == rank])
...         females.append([x for x in pop.individuals(subPop) \
...             if x.sex() == sim.FEMALE and x.rank == rank])
...     #
...     while True:
...         # choose a rank randomly
...         rank = int(pop.individual(randint(0, pop.subPopSize(subPop) - 1), subPop).rank)
...         yield males[rank][randint(0, len(males[rank]) - 1)], \
...             females[rank][randint(0, len(females[rank]) - 1)]
...
>>> def setRank(rank):
...     'The rank of offspring can increase or drop to zero randomly'
...     # only use rank of the father
...     return (rank[0] + randint(-1, 1)) % 3
...
>>> pop = sim.Population(size=[1000, 2000], loci=1, infoFields='rank')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitInfo(lambda : randint(0, 2), infoFields='rank')
...     ],
...     matingScheme=sim.HomoMating(
...         sim.PyParentsChooser(randomChooser),
...         sim.OffspringGenerator(ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.PyTagger(setRank),
...             ])
...     ),
...     gen = 5
... )
5L
```

Built-in parent choosers could be used in a `PyParentsChooser` to choose parents. The parent chooser needs to be initialized with the parental population and subpopulation index. Calling the `chooseParents` function repeatedly will return pairs of individuals from the population (`None` will be returned for one of the parents if the parent chooser only returns one parent). The use of built-in parent choosers can improve the performance of your `PyParentsChooser`, especially for complex selection patterns (e.g. with natural selection). For example, 5.28 implements a similar mating

scheme as Example 5.27 but uses a `RandomParentChooser` to choose males randomly.

Listing 5.28: Use built-in parent choosers in a Python parent chooser

```
>>> import simuPOP as sim
>>> from random import randint
>>>
>>> def randomChooser(pop, subPop):
...     maleChooser = sim.RandomParentChooser(sexChoice=sim.MALE_ONLY)
...     maleChooser.initialize(pop, subPop)
...     females = []
...     # identify females in each social rank
...     for rank in range(3):
...         females.append([x for x in pop.individuals(subPop) \
...             if x.sex() == sim.FEMALE and x.rank == rank])
...     #
...     while True:
...         # choose a random male
...         m = maleChooser.chooseParents()[0]
...         rank = int(m.rank)
...         # find a female in the same rank
...         yield m, females[rank][randint(0, len(females[rank]) - 1)]
...
>>> def setRank(rank):
...     'The rank of offspring can increase or drop to zero randomly'
...     # only use rank of the father
...     return (rank[0] + randint(-1, 1)) % 3
...
>>> pop = sim.Population(size=[1000, 2000], loci=1, infoFields='rank')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitInfo(lambda : randint(0, 2), infoFields='rank')
...     ],
...     matingScheme=sim.HomoMating(
...         sim.PyParentsChooser(randomChooser),
...         sim.OffspringGenerator(ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.PyTagger(setRank),
...             ])
...     ),
...     gen = 5
... )
5L
```

5.3.5 Using C++ to implement a parent chooser **

A user defined parent chooser can be fairly complex and computationally intensive. For example, if a parent tends to find a spouse in his/her vincinity, geometric distances between all qualified individuals and a chosen parent need to be calculated for each mating event. If the optimization of the parent chooser can speed up the simulation significantly, it may be worthwhile to write the parent chooser in C++.

Although it is feasible, and sometimes easier to derive a class from class `ParentChooser` in mating.h (.cpp), modifying simuPOP source code is not recommended because you would have to modify a new version of simuPOP whenever

you upgrade your simuPOP distribution. Implementing your parent choosing algorithm in another Python module is preferred.

The first step is to write your own parent chooser in C/C++. Basically, you will need to pass all necessary information to the C++ level and implement an algorithm to choose parents randomly. Although simple function based solutions are possible, a C++ level class such as the `myParentsChooser` class defined in Example 5.29 is recommended. This class is initialized with indexes of male and female individuals and use a function `chooseParents` to return a pair of parents randomly. This parent chooser is very simple but more complicated parent selection scenarios can be implemented similarly.

Listing 5.29: Implement a parent chooser in C++

```
#include <stdlib.h>
#include <vector>
#include <utility>
using std::pair;
using std::vector;
class myParentsChooser
{
public:
        // A constructor takes all locations of male and female.
        myParentsChooser(const std::vector<int> & m, const std::vector<int> & f)
                : male_idx(m), female_idx(f)
        {
                srand(time(0));
        }

        pair<unsigned long, unsigned long> chooseParents()
        {
                unsigned long male = rand() % male_idx.size();
                unsigned long female = rand() % male_idx.size();
                return std::make_pair(male, female);
        }
private:
        vector<int> male_idx;
        vector<int> female_idx;
};
```

The second step is to wrap your C++ functions and classes to a Python module. There are many tools available but SWIG (`www.swig.org`) is arguably the most convenient and powerful one. To use SWIG, you will need to prepare an interface file, which basically tells SWIG which functions and classes you would like to expose and how to pass parameters between Python and C++. Example 5.30 lists an interface file for the C++ class defined in Example 5.29. Please refer to the SWIG reference manual for details.

Listing 5.30: An interface file for the myParentsChooser class

```
%module myParentsChooser
%{
#include "myParentsChooser.h"
%}
// std_vector.i for std::vector
%include "std_vector.i"
%template() std::vector<int>;
// stl.i for std::pair
%include "stl.i"
%template() std::pair<unsigned long, unsigned long>;
%include "myParentsChooser.h"
```

The exact procedure to generate and compile a wrapper file varies from system to system, and from compiler to compiler. Fortunately, the standard Python module setup process supports SWIG. All you need to do is to write a Python `setup.py` file and let the `distutil` module of Python handle all the details for you. A typical `setup.py` file is demonstrated in Example 5.31.

Listing 5.31: Building and installing the myParentsChooser module

```
from distutils.core import setup, Extension
import sys
# Under linux/gcc, lib stdc++ is needed for C++ based extension.
if sys.platform == 'linux2':
    libs = ['stdc++']
else:
    libs = []
setup(name = "myParentsChooser",
    description = "A sample parent chooser",
    py_modules = ['myParentsChooser'],  # will be generated by SWIG
    ext_modules = [
        Extension('_myParentsChooser',
            sources = ['myParentsChooser.i'],
            swig_opts = ['-O', '-shadow', '-c++', '-keyword',],
            include_dirs = ["."],
    )
  ]
)
```

You parent chooser can now be compiled and installed using the standard Python `setup.py` commands such as

```
python setup.py install
```

Please refer to the Python reference manual for other building and installation options. Note that Python 2.4 and earlier do not support option swig_opts well so you might have to pass these options using command

```
python setup.py build_ext --swig-opts=-O -templatereduce \
    -shadow -c++ -keyword -nodefaultctor install
```

Example 5.29 demonstrates how to use such a C++ parents chooser in your simuPOP script. It uses the same Python parent chooser interface as in 5.27, but leaves all the (potentially) computationally intensive parts to the C++ level `myParentsChooser` object.

Listing 5.32: Implement a parent chooser in C++

```

import simuPOP as sim

# The class myParentsChooser is defined in module myParentsChooser
try:
    from myParentsChooser import myParentsChooser
except ImportError:
    # if failed to import the C++ version, use a Python version
    import random
    class myParentsChooser:
        def __init__(self, maleIndexes, femaleIndexes):
            self.maleIndexes = maleIndexes
            self.femaleIndexes = femaleIndexes
        def chooseParents(self):
            return self.maleIndexes[random.randint(0, len(self.maleIndexes)-1)],\
```

```
                self.femaleIndexes[random.randint(0, len(self.femaleIndexes)-1)]

def parentsChooser(pop, sp):
    'How to call a C++ level parents chooser.'
    # create an object with needed information (such as x, y) ...
    pc = myParentsChooser(
        [x for x in range(pop.popSize()) if pop.individual(x).sex() == sim.MALE],
        [x for x in range(pop.popSize()) if pop.individual(x).sex() == sim.FEMALE])
    while True:
        # return indexes of parents repeatedly
        yield pc.chooseParents()

pop = sim.Population(100, loci=1)
simu.evolve(
    initOps=[
        sim.InitSex(),
        sim.InitGenotype(freq=[0.5, 0.5])
    ],
    matingScheme=sim.HomoMating(sim.PyParentsChooser(parentsChooser),
        sim.OffspringGenerator(ops=sim.MendelianGenoTransmitter())),
    gen = 100
)
```

5.4 Age structured populations with overlapping generations **

Age is an important factor in many applications because it is related to many genetic (most obviously mating) and environmental factors that influence the evolution of a population. The evolution of age structured populations will lead to overlapping generations because parents can co-exist with their offspring in such a population. Although simuPOP is based on a discrete generation model, it can be used to simulate age structured populations.

To evolve an age structured population, you will need to

- Define an information field `age` and use it to store age of all individuals. Age is usally assigned randomly at the beginning of a simulation.
- Define a virtual splitter that splits the parental population into several virtual subpopulation. The most important VSP consists of mating individuals (e.g. individuals with age between 20 and 40). Advanced features of virtual splitters can be used to define complex VSPs such as males between age 20 - 40 and females between age 15-30 (use a `ProductSplitter` to split subpopulations by sex and age, and then a `CombinedSplitter` to join several smaller VSPs together).
- Use a heterogeneous mating scheme that clones most individuals to the next generation (year) and produce offspring from the mating VSP.

Example 5.33 gives an example of the evolution of age-structured population.

- Information fields `ind_id`, `father_id` and `mother_id` and operators `IdTagger` and `PedigreeTagger` are used to track pedigree information during evolution.
- A `CloneMating` mating scheme is used to copy surviving individuals and a `RandomMating` mating scheme is used to produce offspring.
- `IdTagger` and `PedigreeTagger` are used in the `ops` parameter of `RandomMating` because only new offspring should have a new ID and record parental IDs. If you use these operators in the `duringOps` parameter of the `evolve` function, individuals copied by `CloneMating` will have a new ID, and a missing parental ID.

- The resulting population is age-structured so Pedigrees could be extracted from such a population.
- The penetrance function is age dependent. Because this penetrance function is applied to all individuals at each year and an individual will have the disease once he or she is affected, this penetrance function is more or less a hazard function.

Listing 5.33: Example of the evolution of age-structured population.

```
>>> import simuPOP as sim
>>> import random
>>> N = 10000
>>> pop = sim.Population(N, loci=1, infoFields=['age', 'ind_id', 'father_id', 'mother_id'])
>>> pop.setVirtualSplitter(sim.InfoSplitter(field='age', cutoff=[20, 50, 75]))
>>> def demoModel(gen, pop):
...     '''A demographic model that keep a constant supply of new individuals'''
...     # number of individuals that will die
...     sim.stat(pop, popSize=True, subPops=[(0,3)])
...     # individuals that will be kept, plus some new guys.
...     return pop.popSize() - pop.dvars().popSize + N / 75
...
>>> def pene(geno, age, ind):
...     'Define an age-dependent penetrance function'
...     # this disease does not occur in children
...     if age < 16:
...         return 0
...     # if an individual is already affected, keep so
...     if ind.affected():
...         return 1
...     # the probability of getting disease increases with age
...     return (0., 0.001*age, 0.001*age)[sum(geno)]
...
>>> def outputstat(pop):
...     'Calculate and output statistics'
...     sim.stat(pop, popSize=True, numOfAffected=True,
...         subPops=[(0, sim.ALL_AVAIL)],
...         vars=['popSize_sp', 'propOfAffected_sp'])
...     for sp in range(3):
...         print('%s: %.3f%% (size %d)' % (pop.subPopName((0,sp)),
...             pop.dvars((0,sp)).propOfAffected * 100.,
...             pop.dvars((0,sp)).popSize))
...     #
...     return True
...
>>>
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         # random assign age
...         sim.InitInfo(lambda: random.randint(0, 75), infoFields='age'),
...         # random genotype
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         # assign an unique ID to everyone.
...         sim.IdTagger(),
...         sim.PyOutput('Prevalence of disease in each age group:\n'),
...     ],
```

```
...     # increase the age of everyone by 1 before mating.
...     preOps=sim.InfoExec('age += 1'),
...     matingScheme=sim.HeteroMating([
...         # all individuals with age < 75 will be kept. Note that
...         # CloneMating will keep individual sex, affection status and all
...         # information fields (by default).
...         sim.CloneMating(subPops=[(0,0), (0,1), (0,2)], weight=-1),
...         # only individuals with age between 20 and 50 will mate and produce
...         # offspring. The age of offspring will be zero.
...         sim.RandomMating(ops=[
...             sim.IdTagger(),                   # give new born an ID
...             sim.PedigreeTagger(),             # track parents of each individual
...             sim.MendelianGenoTransmitter(),   # transmit genotype
...         ],
...         numOffspring=(sim.UNIFORM_DISTRIBUTION, 1, 3),
...         subPops=[(0,1)]),],
...         subPopSize=demoModel),
...     # number of individuals?
...     postOps=[
...         sim.PyPenetrance(func=pene, loci=0),
...         sim.PyOperator(func=outputstat, step=20)
...     ],
...     gen = 200
... )
Prevalence of disease in each age group:
age < 20: 0.381% (size 2628)
20 <= age < 50: 2.504% (size 3953)
50 <= age < 75: 4.814% (size 3282)
age < 20: 0.639% (size 2660)
20 <= age < 50: 26.901% (size 3933)
50 <= age < 75: 50.407% (size 3313)
age < 20: 0.526% (size 2660)
20 <= age < 50: 27.720% (size 3961)
50 <= age < 75: 60.744% (size 3332)
age < 20: 0.489% (size 2660)
20 <= age < 50: 29.624% (size 3990)
50 <= age < 75: 62.121% (size 3300)
age < 20: 0.639% (size 2660)
20 <= age < 50: 28.045% (size 3990)
50 <= age < 75: 63.188% (size 3325)
age < 20: 0.564% (size 2660)
20 <= age < 50: 28.922% (size 3990)
50 <= age < 75: 60.722% (size 3325)
age < 20: 0.714% (size 2660)
20 <= age < 50: 28.371% (size 3990)
50 <= age < 75: 61.774% (size 3325)
age < 20: 0.526% (size 2660)
20 <= age < 50: 29.298% (size 3990)
50 <= age < 75: 60.451% (size 3325)
age < 20: 0.714% (size 2660)
20 <= age < 50: 29.649% (size 3990)
50 <= age < 75: 61.083% (size 3325)
age < 20: 0.414% (size 2660)
20 <= age < 50: 28.897% (size 3990)
```

```
50 <= age < 75: 63.218% (size 3325)
200L
>>>
>>> # draw two Pedigrees from the last age-structured population
>>> from simuPOP import sampling
>>> sample = sampling.drawNuclearFamilySample(pop, families=2, numOffspring=(2,3),
...     affectedParents=(1,2), affectedOffspring=(1,3))
>>> sim.dump(sample)
Ploidy: 2 (diploid)
Chromosomes:
1:  (AUTOSOME, 1 loci)
   (1)
Information fields:
age ind_id father_id mother_id
population size: 8 (1 subpopulations with 8 Individuals)
Number of ancestral populations: 0

SubPopulation 0 (), 8 Individuals:
   0: MA 1 | 0 |  41 31100 28012 27744
   1: FA 1 | 1 |  34 31950 27515 26655
   2: FA 1 | 1 |  66 27744 22633 22484
   3: FA 1 | 0 |  74 26655 20957 20911
   4: FU 1 | 0 |  41 31099 28012 27744
   5: FU 0 | 1 |  34 31949 27515 26655
   6: MA 1 | 0 |  64 28012 23909 23470
   7: MU 1 | 1 |  68 27515 24745 21596

>>>
```

5.5 Tracing allelic lineage *

Lineage of alleles consists of information such as the distribution of alleles (how many people carry this allele, and the relationship between carriers) and age of alleles (when the alleles were introduced to the population). These information are important for the study of evolutionary history of mutants. They are not readily available for normal simulations, and even if you can track the generations when mutants are introduced, alleles in the present generation that are of the same type (Identity by Stat, IBS) do not necessarily have the same ancestral origin (Identity by Decent, IBD).

The lineage modules of simuPOP provides facilities to track allelic lineage. More specifically,

- Each allele is associated with an integer number (an allelic lineage) that identifies the origin, or the source of the allele.
- The lineage of each allele is transmitted along with the allele during evolution. New alleles will be introduced with their own lineage, even if they share the same states with existing alleles.
- Origin of alleles can be accessed using member functions of the `Individual` and `Population` classes.

Example 5.34 demonstrates how to determine the contribution of genetic information from each ancestor. For this simulation, the alleles of each ancestor are associated with individual-specific numbers. During evolution, some alleles might get lost, some are copied, and pieces of chromosomes are mixed due to genetic recombination. At the end of simulation, the average number of 'contributors' of genetic information to each individual is calculated, as well as the percent of genetic information from each ancestor. Although this particular simulation can be mimicked using

pure-genotype simulations by using special alleles for each ancestor, the combined information regarding the state and origin of each allele will be very useful for genetic studies that involve IBD and IBS.

Listing 5.34: Contribution of genetic information from ancestors

```
>>> import simuOpt
>>> simuOpt.setOptions(alleleType='lineage', quiet=True)
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=[10]*4)
>>>
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.25]*4),
...         sim.InitLineage(range(1000), mode=sim.PER_INDIVIDUAL),
...     ],
...     matingScheme=sim.RandomMating(ops=sim.Recombinator(rates=0.001)),
...     gen = 100
... )
100L
>>> # average number of 'contributors'
>>> num_contributors = [len(set(ind.lineage())) for ind in pop.individuals()]
>>> print('Average number of contributors is %.2f' % (sum(num_contributors) / float(pop.popSize())))
Average number of contributors is 13.98
>>> # percent of genetic information from each ancestor (baseline is 1/1000)
>>> lineage = pop.lineage()
>>> lin_perc = [lineage.count(x)/float(len(lineage)) for x in range(1000)]
>>> # how many of ancestors do not have any allele left?
>>> print('Number of ancestors with no allele left: %d' % lin_perc.count(0.))
Number of ancestors with no allele left: 817
>>> # top five contributors
>>> lin_perc.sort()
>>> lin_perc.reverse()
>>> print('Top contributors (started with 0.001): %.5f %.5f %.5f' % (lin_perc[0], lin_perc[1], lin_perc[2]))
Top contributors (started with 0.001): 0.03474 0.03058 0.02475
```

Example 5.34 uses operator `InitLineage` to explictly assign lineage to alleles of each individual. You can also track the fate of finer genetic pieces by assigning different lineage values to chromosomes, or each loci using different `mode`. This operator can also assign lineage of alleles to an ID stored in an information field, which is usually `ind_id`, a field used by operators such as `IdTagger` and `PedigreeTagger` to assign and trace the pedigree (parentship) information during evolution. More interesting, when such a field is present, mutation operators will assign the IDs of recipients of mutants as the lineage of these mutants. This makes it possible to track the origin of mutants. Moreover, when a mode `FROM_INFO_SIGNED` is used, additional ploidy information will be tagged to lineage values (negative values for mutants on the second homologous copy of chromosomes) so that you can track the inheritance of haplotypes.

To make use of these features, it is important to assign IDs to individuals before these operators are applied. Example 5.35 demonstrates how to use the lineage information to determine the age of mutants. This example evolves a constant population of size 10,000. An `IdTagger` is used before `InitGenotype` so individual IDs will be assigned as allelic lineages. Because all offspring get their own IDs during evolution, the IDs of individuals are assigned to mutants as their lineages, and can be used to determine the age of these mutants. This is pretty easy to do in this example because of constant population size. For more complex demographic models, you might have to record the minimal and maximum IDs of each generation in order to determine the age of mutants.

Listing 5.35: Distribution of age of mutants

```
>>> import simuOpt
>>> simuOpt.setOptions(alleleType='lineage', quiet=True)
```

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=10000, loci=[10]*10, infoFields='ind_id')
>>> # just to make sure IDs starts from 1
>>> sim.IdTagger().reset(1)
>>> pop.evolve(
...     initOps = [
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2, 0.3, 0.4, 0.1]),
...         sim.IdTagger(),
...         sim.InitLineage(mode=sim.FROM_INFO),
...     ],
...     # an extremely high mutation rate, just for demonstration
...     preOps = sim.AcgtMutator(rate=0.01, model='JC69'),
...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.IdTagger(),
...             sim.MendelianGenoTransmitter(),
...         ]
...     ),
...     gen = 10
... )
10L
>>> lin = pop.lineage()
>>> # Number of alleles from each generation
>>> for gen in range(10):
...     id_start = gen*10000 + 1
...     id_end = (gen+1)*10000
...     num_mut = len([x for x in lin if x >= id_start and x <= id_end])
...     print('Gen %d: %5.2f %%' % (gen, num_mut / (2*10000*100.) * 100))
...
Gen 0: 93.40 %
Gen 1:  0.72 %
Gen 2:  0.71 %
Gen 3:  0.70 %
Gen 4:  0.74 %
Gen 5:  0.76 %
Gen 6:  0.73 %
Gen 7:  0.74 %
Gen 8:  0.75 %
Gen 9:  0.75 %
```

5.6 Pedigrees

5.6.1 Create a pedigree object

A `Pedigree` object is basically a static population object that is used to track relationship between individuals. An unique ID is required for all individuals so that individuals could be identified easily using their IDs. Individuals in a pedigree usually have one or two information fields to record the IDs of their parents. Operators `IdTagger` and `PedigreeTagger` are usually used to maintain these information fields which are, although customizable, almost always `ind_id`, `father_id` and `mother_id`. After pedigrees are identified, population operations could be applied, for example, to extracted identified pedigrees from an existing population. This is basically how module `simuPOP.sampling` works.

A new pedigree can be created from a population object with an ID field (default to `ind_id`), and two optional parental ID fields (default to `father_id` and `mother_id`). For example,

```
ped = Pedigree(pop, infoFields=ALL_AVAIL)
```

will create a pedigree object from population `pop` with information fields `ind_id`, `father_id` and `mother_id`, copying all available information fields. The ID field should have an unique ID for each individual and the parental ID fields should record the ID of his or her parents. Genotype information and additional information fields can be copied to a pedigree object if needed. The population object is unchanged.

Another method is to directly convert a population object to a pedigree object, using member function `asPedigree` of a population class. For example,

```
pop.asPedigree()
```

will convert the existing population to a pedigree object. Object pop can then be able to call all pedigree member functions. Once your task is done, you can convert the object back to a population using the `Pedigree.asPopulation()` member function of the object.

A pedigree object can also be created from a file saved by function `Pedigree.save()` or operator `PedigreeTagger` using function `loadPedigree`. Please refer to section *save and load pedigrees* in details.

5.6.2 Locate close and remote relatives of each individual

A pedigree object provides several functions for you to identify spouse, sibling and more distant relatives of each individual. The results are stored to additional information fields of each individual. For example, if you would like to know the offspring of all individuals, you can call function `Pedigree.locateRelatives` as follows:

```
offFields = ['off1', 'off2', 'off3']
ped.addInfoFields(offFields)
ped.locateRelatives(OFFSPRING, resultFields=offFields)
```

This function will locate up to 3 (determined by the length of `resultFields`) offspring of each individual and put their IDs in specified informaton fields. This function allows you to identify spouses (it is common to have multiple spouses when random mating is used), outbred spouse (exclude spouses who share at least one of the parents), offspring (all offspring) and common offspring with a specified spouse, siblings (share at least one parent) and full siblings (share two parents). It also allows you to limit the result by sex and affection status (e.g. find only affected female offspring).

More distant relationship can be derived from these relationship using function `Pedigree.traceRelatives`. This function accepts a path of information fields and follows the path to identify relatives. For example

```
sibFields = ['sib1', 'sib2']
offFields = ['off1', 'off2', 'off3']
cousinFields = ['cousin1', 'cousin2', 'cousin3']
ped.addInfoFields(sibFields + offFields + cousinFields)
ped.locateRelatives(FULLSIBLING, resultFields=sibFields)
ped.locateRelatives(OFFSPRING, resultFields=offFields)
ped.traceRelatives([['father_id', 'mother_id'], sibFields, offFields],
    sex=[ANY_SEX, MALE_ONLY, FEMALE_ONLY],
    resultField=cousinFields)
```

would first identify full siblings and offspring of all individuals and then locate father or mother's male sibling's daughters. As you can imagine, this function can be used to track very complicated relationships.

This function also provides a function for you to identify individuals with specified relatives. Example 5.36 gives an example how to locate a grandfather with at least five grandchildren. With such information, functions such as `Population.extractIndividuals()` could be used to extract Pedigrees from a population. This is basically how `simuPOP.sampling` module works.

Listing 5.36: Locate close and distant relatives of individuals

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, ancGen=2, infoFields=['ind_id', 'father_id', 'mother_id'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.IdTagger(),
...     ],
...     matingScheme=sim.RandomMating(
...         numOffspring=(sim.UNIFORM_DISTRIBUTION, 2, 4),
...         ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.IdTagger(),
...             sim.PedigreeTagger()
...         ],
...     ),
...     gen = 5
... )
5L
>>> ped = sim.Pedigree(pop)
>>> offFields = ['off%d' % x for x in range(4)]
>>> grandOffFields = ['grandOff%d' % x for x in range(5)]
>>> ped.addInfoFields(['spouse'] + offFields + grandOffFields)
>>> # only look spouse for fathers...
>>> ped.locateRelatives(sim.OUTBRED_SPOUSE, ['spouse'], sex=sim.FEMALE_ONLY)
>>> ped.locateRelatives(sim.COMMON_OFFSPRING, ['spouse'] + offFields)
>>> # trace offspring of offspring
>>> ped.traceRelatives([offFields, offFields], resultFields=grandOffFields)
True
>>> #
>>> IDs = ped.individualsWithRelatives(grandOffFields)
>>> # check on ID.
>>> grandFather = IDs[0]
>>> grandMother = ped.indByID(grandFather).spouse
>>> # some ID might be invalid.
>>> children = [ped.indByID(grandFather).info(x) for x in offFields]
>>> childrenSpouse = [ped.indByID(x).spouse for x in children if x >= 1]
>>> childrenParents = [ped.indByID(x).father_id for x in children if x >= 1] \
...     + [ped.indByID(x).mother_id for x in children if x >= 1]
>>> grandChildren = [ped.indByID(grandFather).info(x) for x in grandOffFields]
>>> grandChildrenParents = [ped.indByID(x).father_id for x in grandChildren if x >= 1] \
...     + [ped.indByID(x).mother_id for x in grandChildren if x >= 1]
>>>
>>> def idString(IDs):
...     uniqueIDs = list(set(IDs))
...     uniqueIDs.sort()
...     return ', '.join(['%d' % x for x in uniqueIDs if x >= 1])
...
>>> print('''GrandParents: %d, %d
... Children: %s
... Spouses of children: %s
... Parents of children: %s
... GrandChildren: %s
... Parents of grandChildren: %s ''' % \
```

```
... (grandFather, grandMother, idString(children), idString(childrenSpouse),
...     idString(childrenParents), idString(grandChildren), idString(grandChildrenParents)))
GrandParents: 3040, 3847
Children: 4078, 4079, 4080
Spouses of children: 4446, 4797
Parents of children: 3040, 3847
GrandChildren: 5188, 5189, 5879, 5880, 5881
Parents of grandChildren: 4078, 4079, 4446, 4797
>>>
>>> # let us look at the structure of this complete pedigree using another method
>>> famSz = ped.identifyFamilies()
>>> # it is amazing that there is a huge family that connects almost everyone
>>> len(famSz), max(famSz)
(533, 2383L)
>>> # if we only look at the last two generations, things are much better
>>> ped.addInfoFields('ped_id')
>>> famSz = ped.identifyFamilies(pedField='ped_id', ancGens=[0,1])
>>> len(famSz), max(famSz)
(664, 114L)
```

5.6.3 Identify pedigrees (related individuals)

The `Pedigree` class provides some other functions that allows you to identify related individuals. For example,

- Function `Pedigree.identifyAncestors` identifies all ancestors of specified individuals or all individuals at the present generation. In a diaploid population when there is only one parent, you can see that only a small portion of ancestors have offspring in the last generation.
- Function `Pedigree.identifyOffspring` identifies all offspring of specified individuals across multiple generations.
- Function `Pedigree.identifyFamilies` groups all related individuals into families and assign a family ID to all family members. You might be surprised by how large this kind of family can be when parents are allowed to have multiple spouses.

All these functions support parameters `subPops` and `ancGens` so that you can limit your search in specific subpopulations and ancestral generations. For example, you can limit your search to all male individuals to find out someone's male offspring. Example 5.37 demonstrates how to use these functions to analyze the structure of a complete pedigree.

Listing 5.37: Identify all ancestors, offspring or all related individuals

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, ancGen=-1, infoFields=['ind_id', 'father_id', 'mother_id'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.IdTagger(),
...     ],
...     matingScheme=sim.RandomMating(
...         numOffspring=(sim.UNIFORM_DISTRIBUTION, 2, 4),
...         ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.IdTagger(),
...             sim.PedigreeTagger()
...         ],
```

```
...     ),
...     gen = 19
... )
19L
>>> # we now have the complete pedigree of 20 generations
>>> pop.asPedigree()
>>> # total number of individuals should be 20 * 1000
>>> # how many families do we have?
>>> fam = pop.identifyFamilies()
>>> len(fam)
525
>>> # but how many families with more than 1 individual?
>>> # The rest of them must be in the initial generation
>>> len([x for x in fam if x > 1])
18
>>> # let us look backward. allAnc are the ancestors who have offspring in the
>>> # last generation. You can see this is a small number compared the number of
>>> # ancestors.
>>> allAnc = pop.identifyAncestors()
>>> len(allAnc)
8614
```

5.6.4 Save and load pedigrees

A complete pedigree, including ID, sex and affection status of each individual, IDs of their parents, and optionally values of some information fields and genotypes at some loci could be saved to a file, and be loaded using function `loadPedigree`. The loaded pedigree could be analyzed using pedigree functions, or be used to direct the evolution of another evolutionary process using a pedigree mating scheme.

A pedigree could be saved in two ways. In the first method, a pedigree could be created using the methods described above and be saved using function `Pedigree.save()`. However, if the population is large, recording all ancestral generations may not be feasible. If this is the case, you can use a `PedigreeTagger` operator to save individual information during the evolution. If you do not care about details of the top-most ancestral generation, a PedigreeTagger used in a mating scheme should be enough to record pedigree information of all offspring. Individual in the top-most generation who have offspring in the next generation will be constructed in `loadPedigree`. If you would like to include detailed information about all individuals in the top-most ancestral generation, you can use a `PedigreeTagger` in the `initOps` parameter of the `Simulator.evolve()` or `Population.evolve()` function.

Example 5.38 demonstrates how to use these functions to analyze the structure of a complete pedigree.

Listing 5.38: Save and load a complete pedigree

```
>>> import simuPOP as sim
>>> pop = sim.Population(4, loci=1, infoFields=['ind_id', 'father_id', 'mother_id'],
...     ancGen=-1)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.IdTagger(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         sim.PedigreeTagger(output='>>pedigree.ped', outputLoci=0)
...     ],
...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.MendelianGenoTransmitter(),
```

```
...             sim.IdTagger(),
...             sim.PedigreeTagger(output='>>pedigree.ped', outputLoci=0)
...         ],
...     ),
...     gen = 2
... )
2L
>>> #
>>> print(open('pedigree.ped').read())
1 0 0 F U 0 0
2 0 0 F U 0 1
3 0 0 M U 1 1
4 0 0 M U 1 1
5 4 1 M U 0 1
6 4 2 F U 1 1
7 3 2 F U 0 1
8 3 2 M U 1 1
9 8 7 F U 1 1
10 5 6 M U 1 1
11 5 6 M U 1 1
12 5 7 F U 0 1

>>> pop.asPedigree()
>>> pop.save('pedigree1.ped', loci=0)
>>> print(open('pedigree1.ped').read())
1 0 0 F U 0 0
2 0 0 F U 0 1
3 0 0 M U 1 1
4 0 0 M U 1 1
5 4 1 M U 0 1
6 4 2 F U 1 1
7 3 2 F U 0 1
8 3 2 M U 1 1
9 8 7 F U 1 1
10 5 6 M U 1 1
11 5 6 M U 1 1
12 5 7 F U 0 1

>>> #
>>> ped = sim.loadPedigree('pedigree1.ped')
>>> sim.dump(ped, ancGens=range(3))
Ploidy: 2 (diploid)
Chromosomes:
1:  (AUTOSOME, 1 loci)
   (1)
Information fields:
ind_id father_id mother_id
population size: 4 (1 subpopulations with 4 Individuals)
Number of ancestral populations: 2

SubPopulation 0 (), 4 Individuals:
   0: FU 1 | 1 |  9 8 7
   1: MU 1 | 1 |  10 5 6
   2: MU 1 | 1 |  11 5 6
```

```
   3: FU 0 | 1 |  12 5 7

Ancestral population 1
SubPopulation 0 (), 4 Individuals:
   0: MU 0 | 1 |  5 4 1
   1: FU 1 | 1 |  6 4 2
   2: FU 0 | 1 |  7 3 2
   3: MU 1 | 1 |  8 3 2

Ancestral population 2
SubPopulation 0 (), 4 Individuals:
   0: FU 0 | 0 |  1 0 0
   1: FU 0 | 1 |  2 0 0
   2: MU 1 | 1 |  3 0 0
   3: MU 1 | 1 |  4 0 0
```

5.7 Evolve a population following a specified pedigree structure **

There are some applications where you would like to repeat the same evolutionary process repeatedly using the same pedigree structure. For example, a gene-dropping simulation method basically initialize leaves of a pedigree with random genotypes and pass the genotypes along the pedigree according to Mendelian laws. This can be done in simuPOP using a pedigree mating scheme.

A pedigree mating scheme **PedigreeMating** evolves a population following an existing pedigree structure. If the `Pedigree` object has `N` ancestral generations and a present generation, it can be used to evolve a population for `N` generations, starting from the topmost ancestral generation. At the k-th generation, this mating scheme produces an offspring generation according to subpopulation structure of the `N-k-1` ancestral generation in the pedigree object (e.g. producing the offspring population of generation 0 according to the `N-1` ancestral generation of the pedigree object). For each offspring, this mating scheme copies individual ID and sex from the corresponing individual in the pedigree object. It then locates the parents of each offspring using their IDs in the pedigree object. A list of during mating operators are then used to transmit parental genotype to the offspring.

To use this mating scheme, you should

- Prepare a pedigree object with `N` ancestral generations (and a present generation). Parental information should be available at the present, parental, ..., and `N-1` ancestral generations. This object could be created by evolving a population with `ancGen` set to -1 with parental information tracked by operators `idTagger()` and `pedigreeTagger()`.
- Prepare the population so that it contains individuals with IDs matching this generation, or at least individuals who have offspring in the next topmost ancestral generation. Because individuals in such a population will parent offsprings at the `N-1` ancestral generation of the pedigree object, it is a good idea to assign `ind_id` using `ped.indInfo('father_id')` and `ped.infInfo('mother_id')` of the `N-1` ancestral generation of `ped`.
- Evolve the population using a `PedigreeMating` mating scheme for `N` or less generations. Because parents are chosen by their IDs, subpopulation structure is ignored and migration will have no effect on the evolutionary process. No `IdTagger` should be used to assign IDs to offspring because re-labeling IDs will confuse this mating scheme. This mating scheme copies individual sex from pedigree individual to each offspring because individual sex may affect the way genotypes are transmitted (e.g. a `MendelianGenoTransmitter()` with sex chromosomes).

Example 5.39 demonstrates how to create a complete pedigree by evolving a population without genotype, and then replay the evolutionary process using another population.

Listing 5.39: Use a pedigree mating scheme to replay an evolutionary process.

```
>>> import simuPOP as sim
```

```
>>> # create a population without any genotype
>>> from simuPOP.utils import migrSteppingStoneRates
>>> ped = sim.Population(size=[1000]*5, ancGen=-1,
...     infoFields=['ind_id', 'father_id', 'mother_id', 'migrate_to'])
>>> ped.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.IdTagger(),
...     ],
...     preOps=sim.Migrator(rate=migrSteppingStoneRates(0.1, 5)),
...     matingScheme=sim.RandomMating(
...         numOffspring=(sim.UNIFORM_DISTRIBUTION, 2, 4),
...         ops=[
...             # we do not even need a genotype transmitter...
...             sim.IdTagger(),
...             sim.PedigreeTagger(),
...         ]),
...     gen=100
... )
100L
>>> # convert itself to a pedigree object
>>> ped.asPedigree()
>>> # we should have 100 ancestral generations
>>> N = ped.ancestralGens()
>>> # We should have 101 * 1000 * 5 individuals, but how many actually
>>> # contribute genotype to the last generation?
>>> anc = ped.identifyAncestors()
>>> len(anc)
205647
>>> # remove individuals who do not contribute genotype to the last generation
>>> allIDs = [x.ind_id for x in ped.allIndividuals()]
>>> removedIDs = list(set(allIDs) - set(anc))
>>> ped.removeIndividuals(IDs=removedIDs)
>>> # now create a top most population, but we do not need all of them
>>> # so we record only used individuals
>>> IDs = [x.ind_id for x in ped.allIndividuals(ancGens=N)]
>>> sex = [x.sex() for x in ped.allIndividuals(ancGens=N)]
>>> # create a population, this time with genotype. Note that we do not need
>>> # populaton structure because PedigreeMating disregard population structure.
>>> pop = sim.Population(size=len(IDs), loci=1000, infoFields='ind_id')
>>> # manually initialize ID and sex
>>> sim.initInfo(pop, IDs, infoFields='ind_id')
>>> sim.initSex(pop, sex=sex)
>>> pop.evolve(
...     initOps=sim.InitGenotype(freq=[0.4, 0.6]),
...     # we do not need migration, or set number of offspring,
...     # or demographic model, but we do need a genotype transmitter
...     matingScheme=sim.PedigreeMating(ped,
...         ops=sim.MendelianGenoTransmitter()),
...     gen=100
... )
100L
>>> # let us compare the pedigree and the population object
>>> print(ped.indInfo('ind_id')[:5])
```

```
(500001.0, 500002.0, 500003.0, 500004.0, 500005.0)
>>> print(pop.indInfo('ind_id')[:5])
(500001.0, 500002.0, 500003.0, 500004.0, 500005.0)
>>> print([ped.individual(x).sex() for x in range(5)])
[1, 2, 1, 1, 2]
>>> print([pop.individual(x).sex() for x in range(5)])
[1, 2, 1, 1, 2]
>>> print(ped.subPopSizes())
(663L, 1254L, 1213L, 1230L, 640L)
>>> print(pop.subPopSizes())
(663L, 1254L, 1213L, 1230L, 640L)
```

As long as unique IDs are used for individuals in different generations, the same technique could be used for overlapping generations as well. Even if some individuals are copied from generation to generation, separate IDs should be assigned to these individuals so that a pedigree could be correctly constructed. Because these individuals are copied from a single parent, the pedigree object will have mixed number of parents (some individuals have one parent, some have two). If `PedigreeTagger` operators are used to record parental information, such a pedigree could be loaded by function `loadPedigree`. Example 5.40 evolves an age-structured population. Instead of saving all ancestral generations to a population object and convert it to a pedigree, this example saves the complete pedigree to file `structure.ped` and load the pedigree using function `loadPedigree`.

Listing 5.40: Replay an evolutionary process of an age-structured population

```
>>> import simuPOP as sim
>>>
>>> import random
>>> N = 10000
>>> pop = sim.Population(N, infoFields=['age', 'ind_id', 'father_id', 'mother_id'])
>>> # we simulate age 0, 1, 2, 3
>>> pop.setVirtualSplitter(sim.InfoSplitter(field='age', values=[0, 1, 2, 3]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         # random assign age
...         sim.InitInfo(lambda: random.randint(0, 3), infoFields='age'),
...         # random genotype
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         # assign an unique ID to everyone.
...         sim.IdTagger(),
...     ],
...     # increase the age of everyone by 1 before mating.
...     preOps=sim.InfoExec('age += 1'),
...     matingScheme=sim.HeteroMating([
...         # age 1, 2 will be copied
...         sim.CloneMating(
...             ops=[
...                 # This will set offspring ID
...                 sim.CloneGenoTransmitter(),
...                 # new ID for offspring in order to track pedigree
...                 sim.IdTagger(),
...                 # both offspring and parental IDs will be the same
...                 sim.PedigreeTagger(output='>>structured.ped'),
...             ],
...             subPops=[(0,1), (0,2)],
...             weight=-1
```

```
...         ),
...         # age 2 produce offspring
...         sim.RandomMating(
...             ops=[
...                 # new ID for offspring
...                 sim.IdTagger(),
...                 # record complete pedigree
...                 sim.PedigreeTagger(output='>>structured.ped'),
...                 sim.MendelianGenoTransmitter(),   # transmit genotype
...             ],
...             subPops=[(0,2)]
...         )]
...     ),
...     gen=20
... )
20L
>>>
>>> # use a pedigree object recovered from a file saved by operator PedigreeTagger
>>> ped = sim.loadPedigree('structured.ped')
>>> # create a top most population, but we do not need all of them
>>> # so we record only used individuals
>>> IDs = [x.ind_id for x in ped.allIndividuals(ancGens=ped.ancestralGens())]
>>> sex = [x.sex() for x in ped.allIndividuals(ancGens=ped.ancestralGens())]
>>> # create a population, this time with genotype. Note that we do not need
>>> # populaton structure because PedigreeMating disregard population structure.
>>> pop = sim.Population(size=len(IDs), loci=1000, infoFields='ind_id')
>>> # manually initialize ID and sex
>>> sim.initInfo(pop, IDs, infoFields='ind_id')
>>> sim.initSex(pop, sex=sex)
>>> pop.evolve(
...     initOps=sim.InitGenotype(freq=[0.4, 0.6]),
...     # we do not need migration, or set number of offspring,
...     # or demographic model, but we do need a genotype transmitter
...     matingScheme=sim.PedigreeMating(ped,
...         ops=sim.IfElse(lambda mom: mom is None,
...                 sim.CloneGenoTransmitter(),
...                 sim.MendelianGenoTransmitter())
...     ),
...     gen=100
... )
20L
>>> #
>>> print(pop.indInfo('ind_id')[:5])
(200001.0, 200002.0, 200003.0, 200004.0, 200005.0)
>>> print([pop.individual(x).sex() for x in range(5)])
[1, 2, 2, 1, 1]
>>> # The pedigree object does not have population structure
>>> print(pop.subPopSizes())
(10000L,)
```

The pedigree is then used to repeat the evolutionary process. However, because some individuals were produced sexually using `MendelianGenoTransmitter` and some were copied using `CloneGenoTransitter`, an `IfElse` operator has to be used to transmit genotypes correctly. This example uses the function condition of the `IfElse` operator and makes use of the fact that parent `mom` will be `None` if an individual is copied from his or her father.

Bibliography

B Devlin and N Risch. A comparison of linkage disequilibrium measures for fine-scale mapping. *Genomics*, 29: 311–322, 1995. 124

M Nei. Analysis of gene diversity in subdivided populations. *PNAS*, 70:3321–3323, 1973. 127

T Ohta and M Kimura. The model of mutation appropriate to estimate the number of electrophoretically detectable alleles in a genetic population. *Genet Res*, 22:201–204, 1973. 80

Bo Peng and Marek Kimmel. Simulations provide support for the common disease common variant hypothesis. *Genetics*, 175:1–14, 2007. 243

Bo Peng, Chris I Amos, and Marek Kimmel. Forward-time simulations of human populations with complex diseases. *PLoS Genetics*, 3:e47, 2007. 174, 207

David E Reich and Eric S Lander. On the allelic spectrum of human disease. *Trends Genet*, 17(9):502–510, 2001. 243, 245

Neil Risch. Linkage strategies for genetically complex traits. i. multilocus models. *Am J Hum Genet*, 46:222–228, 1990. 144

B S Weir and C C Cockerham. Estimating f-statsitics for the analysis of population structure. *Evolution*, 38:1358–1370, 1984. 127

5.8 Simulation of mitochondrial DNAs (mtDNAs) *

Mitochondrial DNAs resides in human mitochondrion. A zygote inherits its organelles from the cytoplasm of the egg, and thus organelle inheritance is generally maternal. Whereas there is only one copy of a nuclear chromosome per gamete, there are man copies of an organellar chromosome, forming a population of identical organelle chromosomes that is transmitted to the offspring through the egg. Because these organellar chromosomes are identical, they are modelled in simuPOP as a single chromosome with type `MITOCHONDRIAL`. In order to simulate mitochondrial DNAs, it is important to remember:

- `MendelianGenoTransmitter` and `Recombinator` do not handle mitochondrial DNAs so you will have to explicitly use `MitochondrialGenoTransmitter` to transmit the mitochondrial DNAs from mother to offspring. Note that `CloneGenoTransmitter` is a special transmitter that will copy everything including sex, information fields to offspring.
- The `Stat` operator recognizes this chromosome type and will report allele, haplotype, and genotype counts, and other statistics correctly, although some diploid-specific statistics are not applicable.
- Natural selections on mtDNAs is usually performed using operator `MapSelector` where single alleles are assigned a fitness value. Operator `MaSelector` assumes two alleles and is not applicable.

Example 5.41 demonstrates the use of a `Recombinator` to recombine an autosome and two sex chromosomes, and a `MitochondrialGenoTransmitter` to transmit mitochondrial chromosomes. Natural selection is applied to allele 3 at the 3rd locus on the mitochondrial DNA, whose frequency in the population decreases as a result.

Listing 5.41: Transmission of mitochondrial chromosomes

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=[5]*4,
...     # one autosome, two sex chromosomes, and one mitochondrial chromosomes
...     chromTypes=[sim.AUTOSOME, sim.CHROMOSOME_X, sim.CHROMOSOME_Y, sim.MITOCHONDRIAL],
...     infoFields=['fitness'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.25]*4)
...     ],
...     preOps=[
...         sim.MapSelector(loci=17, fitness={(0,): 1, (1,): 1, (2,): 1, (3,): 0.4})
...     ],
...     matingScheme=sim.RandomMating(ops= [
...         sim.Recombinator(rates=0.1),
...         sim.MitochondrialGenoTransmitter(),
...     ]),
...     postOps=[
...         sim.Stat(alleleFreq=17, step=10),
...         sim.PyEval(r'"%.2f %.2f %.2f %.2f\n" % (alleleNum[17][0],'
...             'alleleNum[17][1], alleleNum[17][2], alleleNum[17][3])', step=10),
...     ],
...     gen = 100
... )
1288.00 273.00 325.00 114.00
1384.00 245.00 371.00 0.00
1492.00 138.00 370.00 0.00
1461.00 69.00 470.00 0.00
1449.00 65.00 486.00 0.00
1536.00 17.00 447.00 0.00
1624.00 7.00 369.00 0.00
1538.00 0.00 462.00 0.00
1619.00 0.00 381.00 0.00
1623.00 0.00 377.00 0.00
100L
```

You might wonder how a mutation can change the allele of all organelles in the mitochondrion. This is generally believed to be done through natural drift during cytoplasmic segreagation, which is not a mitotic process because it takes place in dividing asexual cells. Because only one mitochondrial chromosome is allowed in simuPOP, you will have to use customized chromosome types if you would like to simulate this process. Fortunately, operator `MitochondrialGenoTransmitter` can select random organelles from multiple customized chromosomes, if no chromosome of type `MITOCHONDRIAL` is present.

Example 5.42 demonstrates the fixation of mutant in cells with multiple organelles. Althogh mutations are introduced to only one of the organelles, after a number of cell divisions, the majority of the cells now have only one type of allele. This example uses a `RandomSelection` mating scheme to select cells randomly from the parental population. Because no sexual reproduction is involved, `MitochondrialGenoTransmitter` passes the parental genotype to offspring regardless of sex of parent. This example also demonstrates a disadvantage of using customized chromosomes in that you will have to calculate statistics by yourself because only you know the meaning of these chromosomes. In this example, a function is written to count the number of mutants in each cell (individual), and summarize the number of cells with

0, 1, 2, 3, 4, and 5 copies of the mutant.

Listing 5.42: Evolution of multiple organelles in mitochondrion

```
>>> import simuPOP as sim
>>>
>>> def alleleCount(pop):
...     summary = [0]* 6
...     for ind in pop.individuals():
...         geno = ind.genotype(ploidy=0)
...         summary[geno[0] + geno[2] + geno[4] + geno[6] + geno[8]] += 1
...     print('%d %s' % (pop.dvars().gen, summary))
...     return True
...
>>> pop = sim.Population(1000, loci=[2]*5, chromTypes=[sim.CUSTOMIZED]*5)
>>> pop.evolve(
...     # every one has miDNAs 10, 00, 00, 00, 00
...     initOps=[
...         sim.InitGenotype(haplotypes=[[1]+[0]*9]),
...     ],
...     # random select cells for cytoplasmic segregation
...     matingScheme=sim.RandomSelection(ops= [
...         sim.MitochondrialGenoTransmitter(),
...     ]),
...     postOps=sim.PyOperator(func=alleleCount, step=10),
...     gen = 51
... )
0 [333, 408, 219, 38, 2, 0]
10 [806, 16, 14, 16, 11, 137]
20 [816, 1, 1, 3, 0, 179]
30 [833, 0, 0, 0, 0, 167]
40 [805, 0, 0, 0, 0, 195]
50 [849, 0, 0, 0, 0, 151]
51L
```

Chapter 6

Utility Modules

6.1 Module `simuOpt` (function `simuOpt.setOptions`)

Module `simuOpt` handles options to specify which simuPOP module to load and how this module should be loaded, using function `simuOpt.setOptions` with parameters *alleleType* (`short`, `long`, or `binary`), *optimized* (`standard` or `optimized`), *gui* (whether or not use a graphical user interface and which graphical toolkit to use), *revision* (minimal required version/revision), *quiet* (with or without banner message, and *debug* (which debug code to turn on). These options have been discussed in Example 2.6 and 2.7 and other related sections. Note that **most options can be set by environmental variables and command line options** which are sometimes more versatile to use.

6.1.1 Class `simuOpt.Params` (deprecated)

The `simuOpt` module also provides a class `Params` to help users handle and manage script parameters. There are many other standard or third-party parameter handling modules in Python but this class is designed to help users run a simuPOP script in both batch and GUI modes, using a combination of parameter determination methods. More specifically, if a script uses the `simuOpt.Params` class to handle parameters,

- By default, a parameter input dialog is used to accept user input if the script is executed directly. Default values are given to each parameter and users are allowed to edit them using standard parameter input widgets (on/off button, edit box, dropdown list etc). Detailed explanations to parameters are available as tooltips of corresponding input widgets. A help button is provided that will display the usage of the script when clicked.
- If a configuration file is saved for a previous simulation, command line option `--config configFile` can be used to load all parameters from that configuration file. The parameter input dialog is still used to review and modify parameters.
- Each parameter can also be set using command line options. Command line inputs will override values read from a configuration file.
- If command line option `--gui=interactive` is given, the script will work in batch mode. If the value of a parameter cannot be determined through command line or a configuration file, and is set not to use its default value, users will be asked to enter its value interactively. For example, `myscript.py --gui=interactive --config configFile` will execute a previous simulation directly.
- Using `--gui=batch`, the script will use all default variables unless some of them are specified from command line arguments.

NOTE: `simuOpt.Params` was designed when `getopt` was the only parameter handling module of python. Although it still has some unique features (gui mode, validation etc) compared to the newly introduced standard module `argparse`, it is

less extensible and powerful than `argparse`. **This class is therefore deprecated although it will be kept in simuPOP for backward-compatibility reasons.**

The following sections describes how to use the `simuOpt` class in a simuPOP script.

Define a parameter specification list.

A `Params` object is created from a list of parameter specification dictionaries, and optional short and long descriptions of a script. Each parameter specification dictionary consits of the following fields:

- `name` (required): This field specifies the name of the argument. The argument can be specified from a commandline using `--name=value`, or `--name` if this argument is of a boolean type.
- `default` (required): default value for this parameter.
- `type` (optional): Type of acceptable input, which can be `'boolean'`, `'integer'`, `'integers'`, `'number'`, `'numbers'`, `'string'`, `'strings'`, `'filename'`, `'dirname'`, `('chooseOneOf', values)`, `('chooseFrom', values)`, and a list of acceptable types (e.g. `types.ListType`). This type determines the GUI widget to accept a parameter (e.g. a checkbox for boolean type, and a listbox for choosing one or more values from list, a file browser to browser for filename), how to convert user input to appropriate format (e.g. convert a string to float and a single value to a list), and how to validate a parameter (e.g. a valid filename).
- `label` (optional): A label to display in the parameter input dialog (when `--gui=True`) and as prompt for user input (when `--gui=False`). If this field is missing, a parameter will not be displayed in the parameter input dialog.
- `description` (optional): 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` (optional): A function or an expression to validate if a user input is valid.

Field `validator` is very useful in that it helps simuPOP determine whether or not a user input should be accepted. It accepts a function or an expression. Module `simuOpt` defines a number of functions that you can use. For example, if a parameter defines a probability, you might want to use

```
simuOpt.valueBetween(0, 1)
```

to validate if the input is between 0 and 1. If a list of probabilities is needed, you can use

```
simuOpt.valueListOf(simuOpt.valueBetween(0, 1))
```

More complex logics can be defined using `simuOpt.valueOr` or `simuOpt.valueAnd`. If validation of one parameter involves the values of other parameters, a Python expression can be used. For example, if two parameters need to have the same length, you can use

```
'len(opt1) == len(opt2)'
```

to validate `opt1` or `opt2`.

Example 6.3 shows a parameter specification list that defines parameter `help`, `rate`, `rep` and `pops`. What is special about each parameter is that `help` will not be listed in the parameter input dialog (no `label`) and setting `help` to `True` during interactive parameter input will ignore all other options (`jump`); `rate` has to be between 0 and 1 (using a validation function `valueBetween`), `rep` has to be a positive integer, and `pops` can be one of the three HapMap populations. Please refer to the simuPOP reference manual for details about each dictionary key. The `description` of parameter `pop` demonstrates a special rule in the formatting of such description texts, namely **lines with symbol '|' as the first non-space/tab character are outputed as a separate line without the leading '|' character.**

Listing 6.1: A sample parameter specification list

```
import simuPOP as sim
import types, simuOpt
options = [
    {'name': 'rate',
     'default': [0.01],
     'label': 'Recombination rate',
     'type': 'numbers',
     'description': '''Recombination rate for each replicate. If a single value
            is given, it will be used for all replicates.''',
     'validator': simuOpt.valueListOf(simuOpt.valueBetween(0, 1))
    },
    {'name': 'rep',
     'default': 5,
     'label': 'Number of replicates',
     'type': 'number',
     'description': 'Number of replicates to simulate.',
     'validator': simuOpt.valueGT(0)
    },
    {'name': 'pop',
     'default': 'CEU',
     'label': 'Initial population',
     'type': ('chooseOneOf', ['CEU', 'YRI', 'CHB+JPT']),
     'description': '''Use one of the HapMap sim.populations as the initial
            sim.Population for this simulation. You can choose from:
            |YRI: 33 trios from the Yoruba people in Nigeria (Africa)
            |CEU: 30 trios from Utah with European ancestry (European)
            |CHB+JPT: 90 unrelated individuals from China and Japan (Asia)
            ''',
    }
]
pars = simuOpt.Params(options, 'A demo simulation')
print(pars.usage())
# You can manually feed parameters...
pars.processArgs(['--rep=10'])
```

If you dislike an explicit list of dictionaries, you can use function `simuOpt.addOption` to add options one by one. Example 6.2 shows an equivalent way of specifying three options using this function. This style is used by Python modules such as `optparse` and `argparse`, and is preferred by some simuPOP users.

Listing 6.2: Using simuOpt.param to specify parameters

```
import types, simuOpt
pars = simuOpt.Params(doc='A demo simulation')
pars.addOption('rate', [0.01], label = 'Recombination rate',
        type = 'numbers', description = '''Recombination rate for each replicate.
            If a single value is given, it will be used for all replicates.''')
pars.addOption('rep', 5, label = 'Number of replicates', type = 'integer',
        description = 'Number of replicates to simulate.',
        validator = simuOpt.valueGT(0))
pars.addOption('pop', 'CEU', label = 'Initial population',
        type = ('chooseOneOf', ['CEU', 'YRI', 'CHB+JPT']),
        description = '''Use one of the HapMap sim.populations as the initial
            sim.Population for this simulation. You can choose from:
            |YRI: 33 trios from the Yoruba people in Nigeria (Africa)
```

```
            |CEU: 30 trios from Utah with European ancestry (European)
            |CHB+JPT: 90 unrelated individuals from China and Japan (Asia)
            ''')
print(pars.usage())
```

Get parameters (function `Params.getParam`)

A `Params` object can be created from a parameter specification list. A few member functions are immediately usable. For example, `Params.usage()` returns a detailed usage message about the script and all its parameters (although the usage message will be displayed automatically if command line option `-h` or `--help` is detected). The parameters become attributes of this object using `longarg` names so that you can access them easily (e.g. `par.rate`). Not surprosingly, all parameters now have the default value you assigned to them.

Function `Params.saveConfig(filename)` saves current values of parameters to a configuraiton file `filename`. Parameters that do not have a label are ignored. This configuration file can be loaded later using command line option `--config filename`, perhaps with option `--gui=False` to run the script in batch mode. A less noticed feature of this function is that it also writes a complete command that specifies the same parameters using command line options. This can be handy if you would like to use real parameter definitions instead of `--config filename` in a batch file.

The `params.Params` class provides a number of member functions that allow you to acquire user input in a number of ways. For example `Params.loadConfig` reads a configuration file, `Params.processArgs` checks commandline options, `Params.termGetParam` asks user input interactively, and `Params.guiGetParam` generates and uses a parameter input dialog. These functions can be used several times, on different sets of parameters. In addition, new options could be added programmatically using function `Params.addOption` and allows further flexibility on how parameters are generated. Please refer to *the simuPOP reference manual* on how to use these functions.

Listing 6.3: Get prameters using function getParam

```
usage: runSampleCode.py [--opt[=arg]] ...

options:
  -h, --help
        Display this help message and exit.

  --config=ARG (default: None)
        Load parameters from a configuration file ARG.

  --optimized
        Run the script using an optimized simuPOP module.

  --gui=[batch|interactive|True|Tkinter|wxPython] (default: None)
        Run the script in batch, interactive or GUI mode.

  --rate=ARG  (default: [0.01])
        Recombination rate for each replicate. If a single value is given, it
        will be used for all replicates.

  --rep=ARG  (default: 5)
        Number of replicates to simulate.

  --pop=ARG  (default: 'CEU')
        Use one of the HapMap sim.populations as the initial sim.Population for
        this simulation. You can choose from:
```

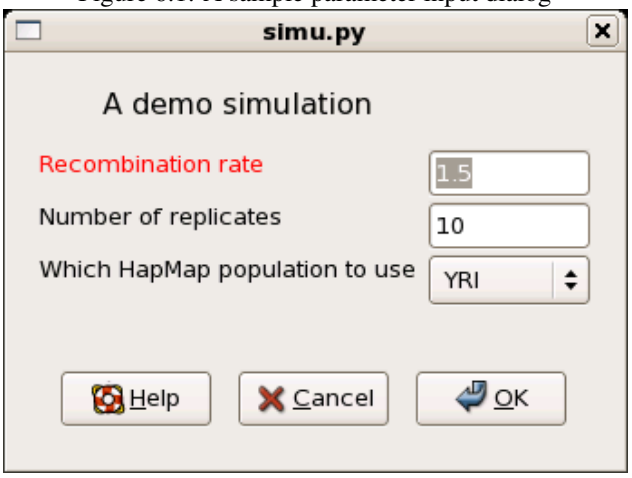
```
        YRI: 33 trios from the Yoruba people in Nigeria (Africa)
        CEU: 30 trios from Utah with European ancestry (European)
        CHB+JPT: 90 unrelated individuals from China and Japan (Asia)
>>> # You can manually feed parameters...
>>> pars.processArgs(['--rep=10'])
True
>>> pars.rep
10
```

Example 6.3 lists some methods to determine parameter values but the last function, `Params.getParam()`, will be used most of the time. This function processes each parameter in the following order:

- If a short or a long command line argument exists, use the command line argument.
- If a configuration file is specified from command line (`--config configFile`), look in this configuration file for a value.
- If `useDefault` is specified, assign a default value to this parameter.
- If `--gui=False` is specified, and the value of the parameter has not be determined, ask users interactively for a value. Otherwise, a parameter input dialog is displayed. A *Tkinter* dialog is usually used but a *wxPython* dialog will be used if *wxPython* is available (unless parameter `--gui=Tkinter` is set).

`Params.getParam` returns `False` if this process fails (e.g. users click cancel in the parameter input dialog). The parameter input dialog for Example 6.3 is shown in Figure 6.1.

Figure 6.1: A sample parameter input dialog



A parameter input dialog for a script that uses the same parameter specification list as Example 6.3. The command line is `simu.py --pop=YRI`. The first parameter is in red because its input is invalid.

Access, manipulate and extract parameters

If `Params.getParam` runs successfully, the `Params` object should have valid value for each parameter. They can be retrieved as attributes (such as `par.rate`) and manipulated easily. Example 6.4 demonstrates how to extend parameter `par.rate` to have the same length as `par.rep`.

When there are a large number of parameters, passing this `Params` object, instead of tens of parameters, is a good way to provide clean interfaces. Alternatively, you can get a list or a dictionary of parameters using member functions `Params.asList()` and `Params.asDict()`.

Listing 6.4: Use the simuOpt object

```
>>> if not pars.getParam():
...     sys.exit(1)
...
>>> pars.saveConfig('sample.cfg')
>>> # post-process parameters
>>> pars.rate
[0.25]
>>> pars.rep
5
>>> pars.rate = pars.rate * pars.rep
>>> # extract parameters as a dictionary or a list
>>> pars.asDict()
{'rate': [0.25, 0.25, 0.25, 0.25, 0.25], 'rep': 5, 'pop': 'CEU'}
>>> pars.asList()
[[0.25, 0.25, 0.25, 0.25, 0.25], 5, 'CEU']
>>> # Default value of parameter rep is changed
>>> # additional attribute is added.
>>> par1 = simuOpt.Params(options, # all parameters with default values
...     rep=50,                     # default value of rep is changed
...     additional=10               # derived parameters are added
... )
>>> # print all parameters except for derived ones.
>>> print(par1.asDict())
{'rate': [0.01], 'rep': 50, 'pop': 'CEU'}
>>> # All parameters are derived ...
>>> par2 = simuOpt.Params(rep=50, pop='CEU', rate=[0.5])
>>> print(par2.asDict())
{}
>>> print(par2.rep, par2.pop)
(50, 'CEU')
```

It is easy to set **additional attributes** to a `Params` object, using either `par.name = value` statement or additional `name=value` pairs in the constructor of a `simuOpt` object. These attributes are not considered as parameters of an `simuOpt` object (e.g. they are not returned by function `Params.asDict()`) but could be used just like regular parameters.

Additional attributes can be used to create a `Params` object without user interaction. For example, objects `par1` and `par2` in Example 6.4 are created with needed attributes. They can be passed to functions where a `Params` object is needed, although some of the attributes are not real parameters (in the sense that they are not created by a parameter specification dictionary and will not be used to handle user input).

6.2 Module `simuPOP.utils`

The `simuPOP.utils` module provides a few utility functions and classes. They do not belong to the simuPOP core but are distributed with simuPOP because they are frequently used and play an important role in some specialized simulation techniques. Please refer to the simuPOP online cookbook (`http://simupop.sourceforge.net/cookbook`) for more utility modules and functions.

6.2.1 Trajectory simulation (classes `Trajectory` and `TrajectorySimulator`)

A forward-time simulation, by its nature, is directly influenced by random genetic drift. Starting from the same parental generation, allele frequencies in the offspring generation would vary from simulation to simulation, with perhaps

a predictable mean frequency which is determined by factors such as parental allele frequency, natural selection, mutation and migration.

Genetic drift is unavoidable and is in many cases the target of theoretical and simulation studies. However, in certain types of studies, there is often a need to control the frequencies of certain alleles in the present generation. For example, if we are studying a particular penetrance model with pre-specified frequencies of disease predisposing alleles, the simulated populations would better have consistent allele frequencies at the disease predisposing loci, and consequently consistent disease prevalence.

simuPOP provides a special offspring generator `ControlledOffspringGenerator` and an associated mating scheme called `ControlledRandomMating` that can be used to generate offspring generations conditioning on frequencies of one or more alleles. This offspring generator essentially uses a reject-sampling algorithm to select (or reject) offspring according to their genotypes at specified loci. A detailed description of this algorithm is given in Peng et al. [2007].

The controlled random mating scheme accepts a user-defined trajectory function that tells the mating scheme the desired allele frequencies at each generation. Example 5.24 uses a manually defined function that raises the frequency of an allele steadily. However, given known demographic and genetic factors, **a trajectory should be simulated randomly so that it represents a random sample from all possible trajectories that match the allele frequency requirement**. If such a condition is met, the controlled evolutionary process can be considered as a random process conditioning on allele frequencies at the present generation. Please refer to Peng et al. [2007] for a detailed discussion about the theoretical requirements of a valid trajectory simulator.

The `simuUtil` module provides functions and classes that implement two trajectory simulation methods that can be used in different situations. The first class is `TrajectorySimulator` which takes a demographic model and a selection model as its input and simulates allele frequency trajectories using a forward or backward algorithm. The demographic model is given by parameter `N`, which can be a constant (e.g. `N=1000`) for constant population size, a list of subpopulation sizes (e.g. `N=[1000, 2000]`) for a structured population with constant size, or a demographic function that returns population or subpopulation sizes at each generation. In the last case, subpopulations can be split or merged with the constrait that subpopulations can be merged into one, from split from one population.

A fitness model specifies the fitness of genotypes at one or more loci using parameter `fitness`. It can be a list of three numbers (e.g. `fitness=[1, 1.001, 1.003]`), repsenting the fitness of genotype `AA`, `Aa` and `aa` at one or more loci; or different fitness for genotypes at each locus (e.g. `fitness=[1, 1.001, 1.003, 1, 1, 1.002]`), or for each combination or genotype (interaction). In the last case, 3^n values are needed for each genotype if there are n loci. This trajectory simulator also accepts generation-specific fitness values by accepting a function that returns fitness values at each generation.

The simulator then simulates trajectories of allele frequencies and return them as objects of class `Trajectory`. This object can be used provide a trajectory function that can be used directly in a `ControlledRandomMating` mating scheme (function `Trajectory.func()`) or provide a list of `PointMutator` to introduce mutants at appropriate generations (function `Trajectory.mutators()`). If a simulation failed after specified number of attempts, a `None` object will be returned.

Forward-time trajectory simulations (function `simulateForwardTrajectory`)

A forward simulation starts from a specified generation with specified allele frequencies at one or more loci. The simulator simulates allele frequencies forward-in-time, until it reaches a specified ending generation. A trajectory object will be returned if the simulated allele frequencies fall into specified ranges. Example 6.5 demonstrates how to use this simulation method to obtain and use a simulated trajectory, for two unlinked loci under different selection pressure.

Listing 6.5: Simulation and use of forward-time simulated trajectories.

```
>>> import simuOpt
>>> simuOpt.setOptions(quiet=True)
>>> import simuPOP as sim
>>> from simuPOP.utils import Trajectory, simulateForwardTrajectory
>>>
>>> traj = simulateForwardTrajectory(N=[2000, 4000], fitness=[1, 0.99, 0.98],
```

```
...     beginGen=0, endGen=100, beginFreq=[0.2, 0.3],
...     endFreq=[[0.1, 0.11], [0.2, 0.21]])
>>> #
>>> traj.plot('log/forwardTrajectory.png', set_ylim_top=0.5,
...     plot_c_sp=['r', 'b'], set_title_label='Simulated Trajectory (forward-time)')
{'c': 'r'}
{'c': 'b'}
>>> pop = sim.Population(size=[2000, 4000], loci=10, infoFields='fitness')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.8, 0.2], subPops=0),
...         sim.InitGenotype(freq=[0.7, 0.3], subPops=1),
...         sim.PyOutput('Sp0: loc2\tloc5\tSp1: loc2\tloc5\n'),
...     ],
...     matingScheme=sim.ControlledRandomMating(
...         ops=[sim.Recombinator(rates=0.01)],
...         loci=5, alleles=1, freqFunc=traj.func()),
...     postOps=[
...         sim.Stat(alleleFreq=[2, 5], vars=['alleleFreq_sp'], step=20),
...         sim.PyEval(r"'%.2f\t%.2f\t%.2f\t%.2f\n' % (subPop[0]['alleleFreq'][2][1],"
...             "subPop[0]['alleleFreq'][5][1], subPop[1]['alleleFreq'][2][1],"
...             "subPop[1]['alleleFreq'][5][1])", step=20)
...     ],
...     gen = 101
... )
Sp0: loc2       loc5    Sp1: loc2       loc5
0.19    0.20    0.30    0.29
0.22    0.20    0.29    0.27
0.19    0.14    0.29    0.27
0.16    0.13    0.24    0.26
0.13    0.13    0.23    0.23
0.18    0.10    0.22    0.20
101L
```

Figure 6.2 plots simulated trajectories of one locus in two subpopulations. The plot function uses either rpy or matplotlib as the underlying plotting library.

Backward-time trajectory simulations (function `simulateBackwardTrajectory`).

A backward simulation starts from specified frequencies at the present generation. In the single-allele case, the simulations goes backward-in-time until an allele gets lost. The length of such a trajectory is random, which is usually a desired property because the age of a mutant in the present generation is usually unknown and is assumed to be random.

This trajectory simulation technique is usually used as follows:

1. Determine a demographic and a natural selection model using which a forward-time simulation will be performed.
2. Given current disease allele frequencies, simulate trajectories of allele frequencies at each DSL using a backward approach.
3. Evolve a population forward-in-time, using designed demographic and selection models. A `ControlledRandomMating` scheme instead of the usual `RandomMating` scheme should be used.

Figure 6.2: Simulated trajectories of one locus in two subpopulations

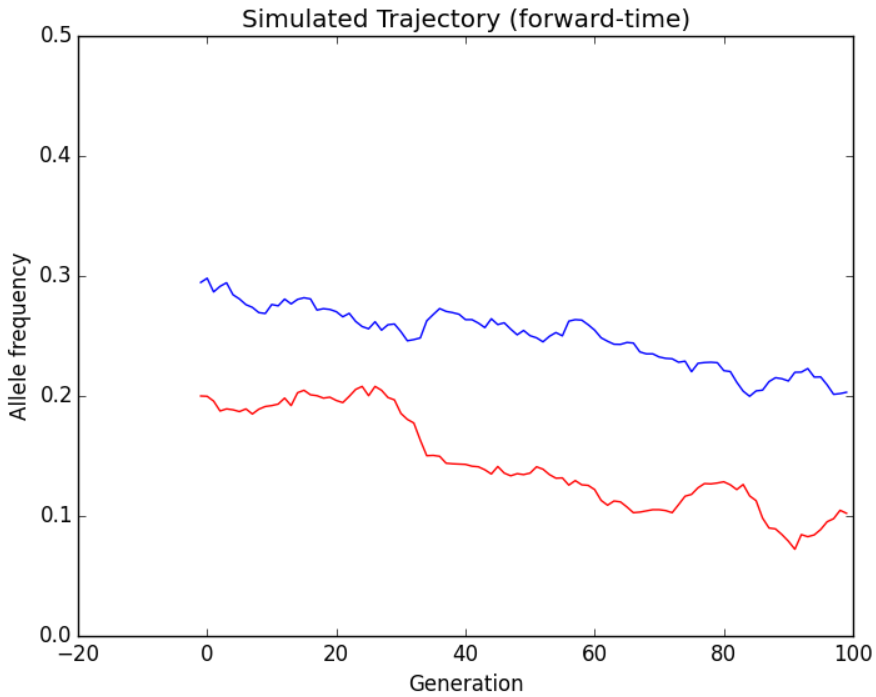


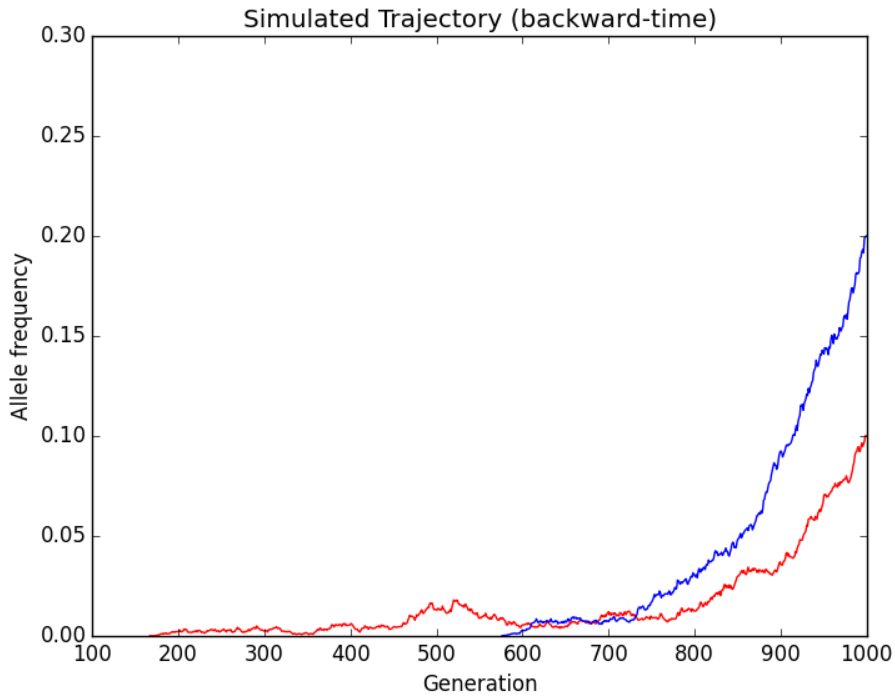
Figure 6.3 plots simulated trajectories of two unlinked loci.

The trajectory is used in a `ControlledRandomMating` scheme in the following evolutionary scenario:

Listing 6.6: Simulation and use of backward-time simulated trajectories.

```
>>> import simuPOP as sim
>>> from simuPOP.utils import Trajectory, simulateBackwardTrajectory
>>> from math import exp
>>> def Nt(gen):
...     'An exponential sim.Population growth demographic model.'
...     return int((5000) * exp(.00115 * gen))
...
>>> def fitness(gen, sp):
...     'Constant positive selection pressure.'
...     return [1, 1.01, 1.02]
...
>>> # simulate a trajectory backward in time, from generation 1000
>>> traj = simulateBackwardTrajectory(N=Nt, fitness=fitness, nLoci=2,
...      endGen=1000, endFreq=[0.1, 0.2])
>>> # matplotlib syntax
>>> traj.plot('log/backTrajectory.png', set_ylim_top=0.3, set_ylim_bottom=0,
...         plot_c_loc=['r', 'b'], set_title_label='Simulated Trajectory (backward-time)')
{'c': 'r'}
{'c': 'b'}
>>>
>>> print('Trajectory simulated with length %s ' % len(traj.traj))
Trajectory simulated with length 834
>>> pop = sim.Population(size=Nt(0), loci=[1]*2)
```

Figure 6.3: Simulated trajectories of two unlinked loci




```
>>> # save Trajectory function in the sim.population's local namespace
>>> # so that the sim.PyEval operator can access it.
>>> pop.dvars().traj = traj.func()
>>> pop.evolve(
...     initOps=[sim.InitSex()],
...     preOps=traj.mutators(loci=[0, 1]),
...     matingScheme=sim.ControlledRandomMating(loci=[0, 1], alleles=[1, 1],
...         subPopSize=Nt, freqFunc=traj.func()),
...     postOps=[
...         sim.Stat(alleleFreq=[0, 1], begin=500, step=100),
...         sim.PyEval(r"'%4d: %.3f (exp: %.3f), %.3f (exp: %.3f)\n' % (gen, alleleFreq[0][1],"
...             "traj(gen)[0], alleleFreq[1][1], traj(gen)[1])",
...             begin=500, step=100)
...     ],
...     gen=1001  # evolve 1001 generations to reach the end of generation 1000
... )
 500: 0.013 (exp: 0.013), 0.000 (exp: 0.000)
 600: 0.005 (exp: 0.005), 0.003 (exp: 0.003)
 700: 0.011 (exp: 0.011), 0.008 (exp: 0.008)
 800: 0.012 (exp: 0.013), 0.031 (exp: 0.031)
 900: 0.037 (exp: 0.037), 0.092 (exp: 0.092)
1000: 0.101 (exp: 0.100), 0.200 (exp: 0.200)
1001L
```

6.2.2 Graphical or text-based progress bar (class `ProgressBar`)

If your simulation takes a while to finish, you could use a progress bar to indicate its progress. The `ProgressBar` class is provided for such a purpose. Basically, you create a `ProgressBar` project with intended total steps, and calls its `update` member function with each progress. Depending on available graphical toolkit and the global or local GUI settings, a `wxPython` based dialog, a `Tkinter` based dialog, or a text-based dialog will be used. Example 6.7 demonstrates how to use a text-based progress bar. If the progress bar is updated at each step (such as in this example), function `update()` can be called without parameter because it updates the progress bar at an increment of 1 in this case.

Listing 6.7: Using a text-based progress bar

```
>>> import simuPOP as sim
>>> from simuPOP.utils import ProgressBar
>>> pop = sim.Population(10000, loci=[10], infoFields='index')
>>> prog = ProgressBar('Setting individual genotype...\n', pop.popSize(), gui=False)
Setting individual genotype...
>>> for idx in range(pop.popSize()):
...     # do something to each individaul
...     pop.individual(idx).index = idx
...     # idx + 1 can be ignored in this case.
...     prog.update(idx + 1)
...
....1....2....3....4....5....6....7....8....9.... Done.
```

6.2.3 Display population variables (function `viewVars`)

If a population has a large number of variables, or if you are not sure which variable to output, you could use function `viewVars` to view the population variables in a tree form. If wxPython is available, a dialog could be used to view the variables interactively. Example 6.8 demonstrates how to use this function. The wxPython-based dialog is displayed in Figure 6.8.

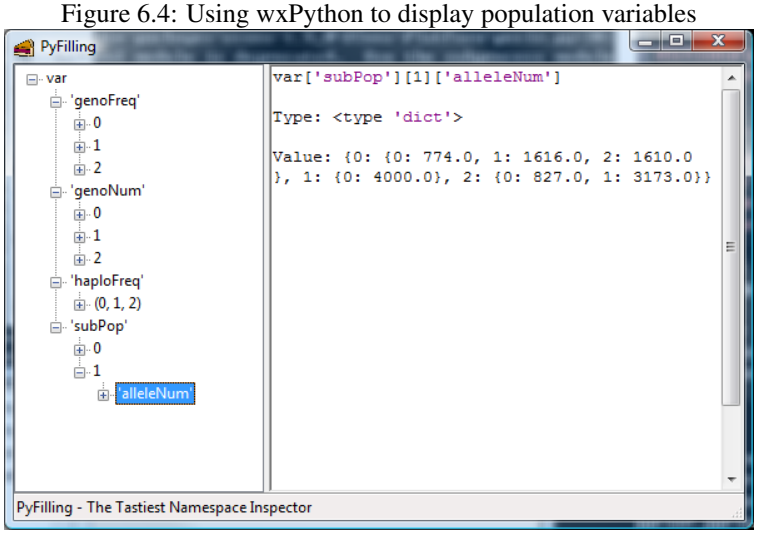
Listing 6.8: Using function viewVars to display population variables

```
import simuPOP as sim
from simuPOP.utils import viewVars
pop = sim.Population([1000, 2000], loci=3)
sim.initGenotype(pop, freq=[0.2, 0.4, 0.4], loci=0)
sim.initGenotype(pop, freq=[0.2, 0.8], loci=2)
sim.stat(pop, genoFreq=[0, 1, 2], haploFreq=[0, 1, 2],
    alleleFreq=range(3),
    vars=['genoFreq', 'genoNum', 'haploFreq', 'alleleNum_sp'])
viewVars(pop.vars())
```

6.2.4 Import simuPOP population from files in `GENEPOP`, `PHYLIP` and `FSTAT` formats (function `importPopulation`)

A function importPopulation is provided in the `simuPOP.utils` module to import populations from files in `GENEPOP`, `PHYLIP` and `FSTAT` formats. Because these formats do not support many of the features of a simuPOP population, this function can only import genotype and basic information of a population. Because formats GENEPOP and FSTAT formats uses allele 0 to indicate missing value, true alleles in these formats start at value 1. If you would like to import alleles with starting value 0, you can use parameter adjust=-1 to adjust imported values, if you data do not have any missing value.

Figure 6.4: Using wxPython to display population variables



6.2.5 Export simuPOP population to files in STRUCTURE, GENEPOP, FSTAT, Phylip, PED, MAP, MS, and CSV formats (function export and operator Exporter)

simuPOP uses a program-specific binary format to save and load populations but you can use the `export` function to export a simuPOP population in other formats if you would like to use other programs to analyze simulated populations. An operator Exporter is also provided so that you could export populations during evolution. Operator arameters such as output, begin, end, step, at, reps, and subPops are supported so that you could export subsets of individuals at multiple generations using different file names (e.g. `output='!"%d.ped" % gen'` to output to different files at different generations).

Commonly used population genetics file formats such as GENEPOP, FSTAT, Phylip, MS, and STRUCTURE are supported. Because these formats cannot store all information in a simuPOP population, export and import operations can lose information. Also, because the processing application have different assumptions, some conversion of genotypes might be needed. For example, because GENEPOP uses allele 0 as missing genotype, function `export(format='genepop')` accepts a parameter `adjust` with default value `1` to export alleles 0, 1 etc to 1, 2, ... The same applies to function `importPopulation` where some file formats accepts a parameter `adjust` (with default value 1) to adjust allele values. Please refer to the simuPOP reference manual for a detailed list of acceptable parameters for each format.

Example 6.9 demonstrates how to import and export a population in formats FSTAT and STRUCTURE. For the FSTAT format, because the population is exported with allele values shifted by 1, the imported population has different alleles than the original population. This can be fixed by adding parameter `adjust=-1` to the `importPopulation` function.

Listing 6.9: Save and load a population

```
>>> import simuPOP as sim
>>> from simuPOP.utils import importPopulation, export
>>> pop = sim.Population([2,4], loci=5, lociNames=['a1', 'a2', 'a3', 'a4', 'a5'],
...     infoFields='BMI')
>>> sim.initGenotype(pop, freq=[0.3, 0.5, 0.2])
>>> sim.initSex(pop)
>>> sim.initInfo(pop, [20, 30, 40, 50, 30, 25], infoFields='BMI')
>>> export(pop, format='fstat', output='fstat.txt')
Exporting....1....2....3....4....5....6....7....8....9.... Done.
>>> print(open('fstat.txt').read())
2 5 3 1
a1
```

```
a2
a3
a4
a5
1 21 21 23 12 12
1 22 23 22 22 21
2 31 21 22 11 13
2 22 22 33 23 21
2 22 32 33 22 21
2 33 33 22 21 32

>>> export(pop, format='structure', phenotype='BMI', output='stru.txt')
Exporting....1....2....3....4....5....6....7....8....9.... Done.
>>> print(open('stru.txt').read())
a1      a2      a3      a4      a5
-1      1.0     1.0     1.0     1.0
1       1       20      1       1       1       0       0
1       1       20      0       0       2       1       1
2       1       30      1       1       1       1       1
2       1       30      1       2       1       1       0
1       2       40      2       1       1       0       0
1       2       40      0       0       1       0       2
2       2       50      1       1       2       1       1
2       2       50      1       1       2       2       0
3       2       30      1       2       2       1       1
3       2       30      1       1       2       1       0
4       2       25      2       2       1       1       2
4       2       25      2       2       1       0       1

>>> pop1 = importPopulation(format='fstat', filename='fstat.txt')
>>> sim.dump(pop1)
Ploidy: 2 (diploid)
Chromosomes:
1:  (AUTOSOME, 5 loci)
  a1 (1), a2 (2), a3 (3), a4 (4), a5 (5)
population size: 6 (2 subpopulations with 2 (1), 4 (2) Individuals)
Number of ancestral populations: 0

SubPopulation 0 (1), 2 Individuals:
   0: MU 22211 | 11322
   1: MU 22222 | 23221
SubPopulation 1 (2), 4 Individuals:
   2: MU 32211 | 11213
   3: MU 22322 | 22331
   4: MU 23322 | 22321
   5: MU 33223 | 33212
```

Because coalescent simulations are increasingly used to generate initial populations in equilibrium stats, importing data in MS format is very useful. Because MS only simulates haploid sequences with genotype only at segregating sites, you might have to simulate an even number of sequences and use option ploidy=2 to import the simulated data as a haploid population. In addition, a parameter mergeBy is provided to import multiple replicates as multiple subpopulations or chromosomes. This corresponds to the splitBy parameter when you export your data in MS format. Example 6.10 demonstrates how to use these parameters.

Listing 6.10: Export and import in MS format

```
>>> import simuPOP as sim
>>> from simuPOP.utils import importPopulation, export
>>> pop = sim.Population([20,20], loci=[10, 10])
>>> # simulate a population but mutate only a subset of loci
>>> pop.evolve(
...     preOps=[
...         sim.InitSex(),
...         sim.SNPMutator(u=0.1, v=0.01, loci=range(5, 17))
...     ],
...     matingScheme=sim.RandomMating(),
...     gen=100
... )
100L
>>> # export first chromosome, all individuals
>>> export(pop, format='ms', output='ms.txt')
Exporting...1...2...3...4...5...6...7...8...9... Done.
>>> # export first chromosome, subpops as replicates
>>> export(pop, format='ms', output='ms_subPop.txt', splitBy='subPop')
Exporting...1...2...3...4...5...6...7...8...9... Done.
>>> # export all chromosomes, but limit to all males in subPop 1
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> export(pop, format='ms', output='ms_chrom.txt', splitBy='chrom', subPops=[(1,0)])
Exporting...1...2...3...4...5...6...7...8...9... Done.
>>> #
>>> print(open('ms_chrom.txt').read())
simuPOP_export 20 2
30164 48394 29292

//
segsites: 5
positions: 6.0000 7.0000 8.0000 9.0000 10.0000
11110
11111
11110
11111
11011
11111
01111
10111
11111
11111
01111
01111
11011
11111
01111
11011
11101
10111
11111
11111

//
```

```
segsites: 7
positions: 1.0000 2.0000 3.0000 4.0000 5.0000 6.0000 7.0000
1101111
1110011
1101110
1111111
0111110
1111111
1110001
1111111
0111110
1111111
1111111
1111111
1111111
1011111
1111111
1111111
1011111
1111111
1111111
1011111

>>> # import as haploid sequence
>>> pop = importPopulation(format='ms', filename='ms.txt')
>>> # import as diploid
>>> pop = importPopulation(format='ms', filename='ms.txt', ploidy=2)
>>> # import as a single chromosome
>>> pop = importPopulation(format='ms', filename='ms_subPop.txt', mergeBy='subPop')
```

If the file format you are interested in is not supported, you can export data in csv format and convert the file by yourself. You can also try to write your own import or export functions as described in the advanced topics section of this guide.

6.2.6 Export simuPOP population in csv format (function `saveCSV`, deprecated)

Function `saveCSV` is provided in the `simuPOP.utils` module to save (the present generation of) a simuPOP population in comma separated formats. It allows you to save individual information fields, sex, affection status and genotype (in that order). Because this function allows you to output these information in different formats using parameters `infoFormatter`, `sexFormatter`, `affectionFormatter`, and `genoFormatter`, this function can already be used to export a simuPOP population to formats that are recognizable by some populat software applications. Example 6.11 creates a small population and demonstrates how to save it in different formats.

Listing 6.11: Using function saveCSV to save a simuPOP population in different formats

```
>>> import simuPOP as sim
>>> from simuPOP.utils import saveCSV
>>> pop = sim.Population(size=[10], loci=[2, 3],
...     lociNames=['r11', 'r12', 'r21', 'r22', 'r23'],
...     alleleNames=['A', 'B'], infoFields='age')
>>> sim.initSex(pop)
>>> sim.initInfo(pop, [2, 3, 4], infoFields='age')
>>> sim.initGenotype(pop, freq=[0.4, 0.6])
>>> sim.maPenetrance(pop, loci=0, penetrance=(0.2, 0.2, 0.4))
```

```
>>> # no filename so output to standard output
>>> saveCSV(pop, infoFields='age')
age, sex, aff, r11_1, r11_2, r12_1, r12_2, r21_1, r21_2, r22_1, r22_2, r23_1, r23_2
2.0, F, A, B, B, B, B, B, A, B, B, B, A
3.0, F, U, B, A, B, A, B, A, A, A, A, B
4.0, M, U, B, B, B, B, B, B, B, B, B, A
2.0, M, U, B, A, B, A, B, B, B, B, B, A
3.0, M, A, B, B, B, B, B, B, A, A, B, A
4.0, M, U, A, B, B, A, B, B, B, B, B, B
2.0, M, U, B, B, B, B, B, B, B, B, A, A
3.0, F, U, B, B, A, A, B, B, A, A, B, B
4.0, F, U, A, B, B, B, B, B, B, A, B, B
2.0, F, A, B, A, A, B, A, A, B, B, B, A
>>> # change affection code and how to output genotype
>>> saveCSV(pop, infoFields='age', affectionFormatter={True: 1, False: 2},
...     genoFormatter={(0,0):'AA', (0,1):'AB', (1,0):'AB', (1,1):'BB'})
age, sex, aff, r11, r12, r21, r22, r23
2.0, F, 1, BB, BB, AB, BB, AB
3.0, F, 2, AB, AB, AB, AA, AB
4.0, M, 2, BB, BB, BB, BB, AB
2.0, M, 2, AB, AB, BB, BB, AB
3.0, M, 1, BB, BB, BB, AA, AB
4.0, M, 2, AB, AB, BB, BB, BB
2.0, M, 2, BB, BB, BB, BB, AA
3.0, F, 2, BB, AA, BB, AA, BB
4.0, F, 2, AB, BB, BB, AB, BB
2.0, F, 1, AB, AB, AA, BB, AB
>>> # save to a file
>>> saveCSV(pop, filename='pop.csv', infoFields='age', affectionFormatter={True: 1, False: 2},
...     genoFormatter=lambda geno: (geno[0] + 1, geno[1] + 1), sep=' ')
>>> print(open('pop.csv').read())
age sex aff r11_1 r11_2 r12_1 r12_2 r21_1 r21_2 r22_1 r22_2 r23_1 r23_2
2.0 F 1 2 2 2 2 2 1 2 2 2 1
3.0 F 2 2 1 2 1 2 1 1 1 1 2
4.0 M 2 2 2 2 2 2 2 2 2 2 1
2.0 M 2 2 1 2 1 2 2 2 2 2 1
3.0 M 1 2 2 2 2 2 2 1 1 2 1
4.0 M 2 1 2 2 1 2 2 2 2 2 2
2.0 M 2 2 2 2 2 2 2 2 2 1 1
3.0 F 2 2 2 1 1 2 2 1 1 2 2
4.0 F 2 1 2 2 2 2 2 2 1 2 2
2.0 F 1 2 1 1 2 1 1 2 2 2 1
```

This function is now deprecated with the introduction of function `export` and operator `Exporter`.

6.3 Module `simuPOP.demography`

6.3.1 Predefined migration models

The following functions are defined to generate migration matrixes for popular migration models.

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

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

  for a traditional **island model** where individuals have equal probability of migrating to any other subpopulations. This model is also called a **migrant-pool island model**.

- `migrHierarchicalIslandRates(r1, r2, n)` models a **hierarchical island model** in which local populations are grouped into neighborhoods within which there is considerable gene flow and between which there is less gene flow. n should be a list of group size. r_1 is the within-group migration rate and r_2 is the cross-group migration rate. That is to say, an individual in an island has probability $1 - r_1 - r_2$ to stay, r_1 to be a migratant to other islands in the group (migration rate depending on the size of group), and r_2 to be a migrant to other islands in another group (migration rate depending on the number of islands in other groups). Both r_1 and r_2 can vary across groups of islands. For example, `migrHierarchicalIslandRates([r11, r12], r2, [3, 2])` returns a 5×5 migration matrix

$$\begin{pmatrix} 1-r_{11}-r_2 & \frac{r_{11}}{2} & \frac{r_{11}}{2} & \frac{r_2}{2} & \frac{r_2}{2} \\ \frac{r_{11}}{2} & 1-r_{11}-r_2 & \frac{r_{11}}{2} & \frac{r_2}{2} & \frac{r_2}{2} \\ \frac{r_{11}}{2} & \frac{r_{11}}{2} & 1-r_{11}-r_2 & \frac{r_2}{2} & \frac{r_2}{2} \\ \frac{r_2}{3} & \frac{r_2}{3} & \frac{r_2}{3} & 1-r_{12}-r_2 & r_{12} \\ \frac{r_2}{3} & \frac{r_2}{3} & \frac{r_2}{3} & r_{12} & 1-r_{12}-r_2 \end{pmatrix}$$

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

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

  and if `circular=True`, returns

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

- `migr2DSteppingStoneRates(r, m, n, diagonal=False, circular=False)` models a 2D stepping stone model in which local populations are arranged into a lattice of $m \times n$ (m rows, n columns) patches. The population thus needs to have $m \times n$ subpopulations with subpopulation indexes counted by row. In this model, an individual in a center patch has a probability of $1 - r$ to stay, and $r/4$ to migrate to its neighbor patches if `diagonal` is set to False, or $r/8$ to migrate to 8 neighbors (including diagnal ones) if `range` is set to 8. If `circular` is set to `False`, the corner patch has a probability of $r/2$ or $r/3$ (if range=8) to migrate, and a side patch has a probability $r/3$ or $r/5$ to migrate. If `circular` is set to `True`, the lattice will be conceptually connected to a ball so that there is no boundary effect. For example, for a 3 by 2 lattice

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{pmatrix}$$

with `diagonal=False` and `circular=False`, the migration matrix will be

$$\begin{pmatrix} 1-r & \frac{r}{2} & \frac{r}{2} & & & \\ \frac{r}{2} & 1-r & & \frac{r}{2} & & \\ \frac{r}{3} & & 1-r & \frac{r}{3} & \frac{r}{3} & \\ & \frac{r}{3} & \frac{r}{3} & 1-r & & \frac{r}{3} \\ & & \frac{r_2}{2} & & 1-r & \frac{r}{2} \\ & & & \frac{r}{2} & \frac{r}{2} & 1-r \end{pmatrix}$$

Many more migration models have been proposed and studied, sometimes under different names with slightly different definitions. If you cannot find your model there, it should not be too difficult to construct a migration rate matrix for it. I will be glad to add such functions to this module if you could provide a reference and your implementation of the model.

6.3.2 Uniform interface of demographic models

A realistic demographic models can be very complex that involves population growth, population bottleneck, subdivided populations, migration, population split and admixture for a typical demographic model for human populations, and carrying capacity, fecunity, sex distribution and many more factors for more complex ones (e.g. models for animal populations under continuous habitat). The goal of this module is to provide a common interface for demographic models, classes for frequently used demographic models, and several pre-defined demographic models for human populations. More complex demographic models will be added if needed.

A demographic model usually consists of the following components:

- An initial population size that is used to initialize a population (the `size` parameter of `sim.Population`)
- One or more operators to split and merge populations (e.g. Operators `SplitSubPops`)
- One or more operators to migrate individuals across subpopulations (e.g. operator `Migrator`)
- Determine sizes of subpopulations before mating (parameter `subPopSize` of a mating scheme)
- Number of generations to evolve (parameter `gen` of the `evolve` function) or operators to terminate the evolution conditionally (e.g. operator `TerminateIf`)

Using an object-oriented approach, a demographic model defined in this module encapsulates all these in a single object. More specifically, a demographic object `model` is a callable Python object that

- has attribute `model.init_size` and `model.info_fields` to determine the initial population size and required information fields to construct an initial population (e.g., `sim.Population(size=model.init_size, infoFields=model.info_fields + ['my_fields'])`)
- handles population split, merge, migration etc internally before mating when it is passed to parameter `subPopSize` of a mating scheme. (e.g. `RandomMating(subPopSize=model)`)
- has attribute `model.num_gens` to determine the number of generations to evolve (e.g. `pop.evolve(..., gen=model.num_gens)`). The model can optionally terminate the evolution by returnning an empty offspring population size before mating.
- provides a function `model.plot(filename='', title='')` to plot the demographic function. It by default prints out population sizes whenever population size changes. If a `filename` is specified and if module `matplotlib` is available, it will plot the demographic model and save it to filename. A `title` can be specified for the figure. This function actually use the demographic model to evolve a haploid population using `RandomSelection` mating scheme, which is a good way to test if your demographic model works properly.

- saves population sizes of evolved generations, which makes it possible to revert an evolutionary process to an previous state using operator `RevertIf`.

A demographic model can be defined in two ways. The first approach is to specify the size of subpopulations at each generation, and the second approach is to specify the events that change population sizes. The `simuPOP.demography` module provides functions and classes to define demographic models using both approaches and you can use the one that is most convenient for your model.

6.3.3 Demographic models defined by outcomes

The `simuPOP.demography` module defines a number of widely used demographic models, including linear and exponential population growth with carrying capacity, shrink, split and merge, and bottleneck.

For example,

- `InstantChangeModel(T=1000, N0=1000, G=500, NG=2000)`

  defines an instant population growth model that expands a population of size from 1000 to 2000 instantly at generation 500

- `InstantChangeModel(T=1000, N0=1000, G=[500, 600], NG=[100, 1000])`

  defines a bottleneck model that introduces a bottleneck of size 100 between generation 500 and 600 to a population of size 1000

- `InstantChangeModel(T=1000, N0=1000, G=500, NG=[[400, 600]])`

  defines a bottleneck model that split a population of size into two subpopulations of sizes 400 and 600 at generation 500

- `ExponentialGrowthModel(T=100, N0=1000, NT=10000)`

  expands a population of size 1000 to 10000 in 100 generations

- ```
  ExponentialGrowthModel(T=100, N0=[200, 800], r=[0.02, 0.01],
      ops=Migrator(rate=[[0, 0.1], [0.1, 0]])
  ```

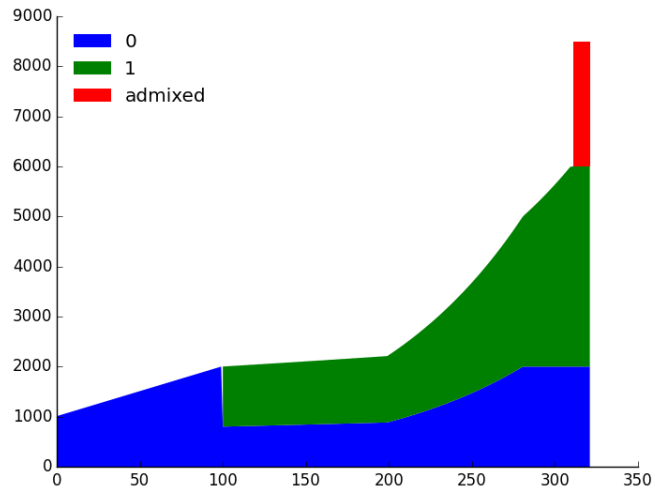
  expands a population of two subpopulation sizes at rate `0.02` and `0.01` for `100` generations, with migration between these two subpopulations. The initial population will be resized (split if necessary) to two populations of sizes 200 and 800.

- `LinearGrowthModel(N0=(200, 'A'), r=0.02, NT=1000)`

  expands a population of size `200` at a rate 0f `0.02` (add 4 individuals at each generation) until it reaches size `1000`. Here the initial size is expressed as a size name tuple, which directs the demographic model to assign the name `A` to the initial population. Such named size is acceptable for all places where population size is needed.

Here we specify only two of the three parameters for linear and exponential growth models and allow simuPOP to figure out the rest. If all three parameters are specified, the ending population size will be interpretted as carraying capacity, namely population growth (or decline of negative rates are specified) will stop after it reaches the specified size.

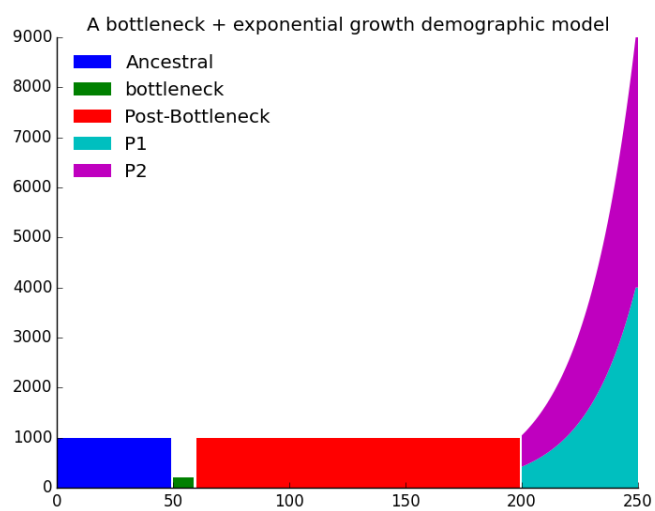
A demographic model does not have to have a fixed initial population size. If an initial population size is not provided, its size will be determined from the population when it is first applied to. For example

- `InstantChangeModel(T=100, G=50, NT=[0.5, 0.5])`
  ```

split a population into two equally sized subpopulations at generation 50. The ending population size is set to `[0.5, 0.5]`, which means 50% of the size at time `G`.

- `InstantChangeModel(T=100, G=50, NT=[None, 100])`

  forks a population of size 100 from the main population at generation 50. `NT=[None, 100]` is equivalent to `NT=[1.0, 100]` in this case.

- `InstantChangeModel(T=0, removEmptySubPops=True)`

  removes all empty subpopulations from the existing subpopulation. Here we do not specify an input population size because the the size of the input population will be kept.

- `InstantChangeMoel(T=0, N0=[None, 0, None], removEmptySubPops=True)`

  removes the second of the three subpopulations while keep other two subpopulations intact. The input population of this demographic model must have three subpopulations.

- `ExponentialGrowthModel(T=100, NT=[10000, 20000])`

  expands a population of two subpopulations to sizes `10000` and `20000` in `100` generations. An error will be raised if the population does not have two subpopulations.

- ```
  ExponentialGrowthModel(T=100, N0=[1., 400], NT=[10000, 20000],
      ops=Migrator(rate=[[0, 0.1], [0.1, 0]])
  ```

  split a population into two subpopulations. The first one keeps all individuals (100%), the second one with 400 individuals, and then expands them, with migration, to sizes `10000` and `20000` in `100` generations.

The demography model also defines two models for population admxture. The HI model (Hybrid Isolation) model creates a separate subpopulation with μ and $1-\mu$ individuals from two specified subpopulations. The CGF (Continuous Gene Flow) model replaces $1-\mu$ individuals from the doner population at each generation, thus keep both the recipient and doner population constant in size. For example,

- `AdmixtureModel(model=('HI', 1, 3, 0.5, 'Admixed'), T=10)`

  Creates a separate population with 50% of individuals from subpopulation 1 and 50% of individuals from subpopulation 3, regardless if population sizes 1 and 3 have the same number of individuals. An optional name Admixed is assigned to the new subpopulation. The admixed population will evolve independently for 10 generations.

- `AdmixtureModel(model=('CGF', 1, 3, 0.9), T=10)`

  Replaces 10% of individuals in subpopulation 1 with individuals from subpopulation 3 for 10 generations.

As you can imagine, these models do not provide a valid `init_size` to initialize a population. As a matter of fact, they are mostly stacked to other demographic models to form more complex demographic models, in model `MultiStageModel`. For example,

- ```
  MultiStageModel([
      InstantChangeModel(T=1000, N0=1000, G=[500, 600], NG=[100, 1000]),
      ExponentialGrowthModel(T=100, NT=10000)
  ])
  ```

  defines a demographic model with a bottleneck followed by exponential population growth. `N0` of the second stage is not specified because it is determined from its previous stage.

- ```
  MultiStageModel([
      LinearGrowthModel(T=100, N0=1000, r=0.01),
      ExponentialGrowthModel(T=100, N0=[0.4, 0.6], r=0.001),
      ExponentialGrowthModel(r=0.01, NT=[2000, 4000]),
      AdmixtureModel(model=('HI', 0, 1, 0.8, 'admixed'), T=10)
  ])
  ```

  defines a demographic model that expands a single population linearly for 100 generations, split into two subpopulations and grow exponentially at a rate of 0.001, and growth at a higher rate of 0.01 until they reaches sizes 2000 and 4000 respectively. This stage is tricky because one of the subpopulations will reach its carrying capacity sooner and keep a contant population size afterwards. As the last step, the two populations admixed and formed a new subpopulation called `admixed`. The model is depicted in figure 6.5

Figure 6.5: A linear and two stage exponential population growth model, followed by population admixture



Example 6.12 defines a demographic model use it to evolve a population. The demographic model is depicted in Figure 6.6.

Listing 6.12: A demographic model for human population

```
>>> import simuPOP as sim
>>> from simuPOP.demography import *
>>> model = MultiStageModel([
...     InstantChangeModel(T=200,
...         # start with an ancestral population of size 1000
...         N0=(1000, 'Ancestral'),
...         # change population size at 50 and 60
...         G=[50, 60],
...         # change to population size 200 and back to 1000
...         NG=[(200, 'bottleneck'), (1000, 'Post-Bottleneck')]),
...     ExponentialGrowthModel(
...         T=50,
...         # split the population into two subpopulations
...         N0=[(400, 'P1'), (600, 'P2')],
...         # expand to size 4000 and 5000 respectively
...         NT=[4000, 5000])]
```
```

```
...     )
>>> #
>>> # model.init_size returns the initial population size
>>> # migrate_to is required for migration
>>> pop = sim.Population(size=model.init_size, loci=1,
...     infoFields=model.info_fields)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.RandomMating(subPopSize=model),
...     finalOps=
...         sim.Stat(alleleFreq=0, vars=['alleleFreq_sp']),
...     gen=model.num_gens
... )
250L
>>> # print out population size and frequency
>>> for idx, name in enumerate(pop.subPopNames()):
...     print('%s (%d): %.4f' % (name, pop.subPopSize(name),
...         pop.dvars(idx).alleleFreq[0][0]))
...
P1 (4000): 0.6185
P2 (5000): 0.7218
>>> # get a visual presentation of the demographic model
>>> model.plot('log/demoModel.png',
...     title='A bottleneck + exponential growth demographic model')
A bottleneck + exponential growth demographic model
0: 1000 (Ancestral)
50: 200 (bottleneck)
60: 1000 (Post-Bottleneck)
200: 419 (P1), 626 (P2)
201: 439 (P1), 653 (P2)
202: 459 (P1), 681 (P2)
203: 481 (P1), 711 (P2)
204: 504 (P1), 742 (P2)
205: 527 (P1), 774 (P2)
206: 552 (P1), 807 (P2)
207: 578 (P1), 842 (P2)
208: 605 (P1), 879 (P2)
209: 634 (P1), 917 (P2)
210: 664 (P1), 957 (P2)
211: 695 (P1), 998 (P2)
212: 728 (P1), 1041 (P2)
213: 762 (P1), 1086 (P2)
214: 798 (P1), 1133 (P2)
215: 836 (P1), 1183 (P2)
216: 875 (P1), 1234 (P2)
217: 916 (P1), 1287 (P2)
218: 960 (P1), 1343 (P2)
219: 1005 (P1), 1401 (P2)
220: 1052 (P1), 1462 (P2)
221: 1102 (P1), 1525 (P2)
222: 1154 (P1), 1591 (P2)
```

```
223: 1208 (P1), 1660 (P2)
224: 1265 (P1), 1732 (P2)
225: 1325 (P1), 1807 (P2)
226: 1387 (P1), 1885 (P2)
227: 1452 (P1), 1967 (P2)
228: 1521 (P1), 2052 (P2)
229: 1592 (P1), 2141 (P2)
230: 1667 (P1), 2234 (P2)
231: 1746 (P1), 2331 (P2)
232: 1828 (P1), 2432 (P2)
233: 1915 (P1), 2537 (P2)
234: 2005 (P1), 2647 (P2)
235: 2099 (P1), 2761 (P2)
236: 2198 (P1), 2881 (P2)
237: 2302 (P1), 3006 (P2)
238: 2410 (P1), 3136 (P2)
239: 2524 (P1), 3272 (P2)
240: 2643 (P1), 3414 (P2)
241: 2767 (P1), 3562 (P2)
242: 2898 (P1), 3716 (P2)
243: 3034 (P1), 3877 (P2)
244: 3177 (P1), 4045 (P2)
245: 3327 (P1), 4220 (P2)
246: 3484 (P1), 4403 (P2)
247: 3648 (P1), 4593 (P2)
248: 3820 (P1), 4792 (P2)
249: 4000 (P1), 5000 (P2)
```

Figure 6.6: A exponential population growth followed by bottleneck demographic model



6.3.4 Demographic models defined by population changes (events)

Another way to define a demographic model is to specify the events that changes population sizes. This approach can be easier to use because it conforms with the way many demographic models are specified, also because the events can

be specified for a subset of subpopulations so you can, for example, split one subpopulation without worrying about its impact on other subpopulations.

A event-based demographic model is defined using

```
EventBasedModel(events=[], T=None, N0=None, ops=[], infoFields=[])
```

where `T` and `N0` are the duration and initial size of the demographic model, respectively, and ops is the operators that will be applied to the population (without checking applicability). Parameter `events` acepts one or more of `DemographicEvent` and its derived classes. For example,

```
ExpansionEvent(rates=0.05, begin=500)
```

expands all subpopulations exponentially at a rate of 0.05, and

```
ExpansionEvent(rates=[0.05, 0.01], capacity=10000, subPops=[0, 2], begin=500)
```

expands two subpopulations at rates 0.05 and 0.01 respectively, until they reach 10000 individuals in each subpopulation.

```
ExpansionEvent(slopes=500, subPops=[0, 2], begin=500)
```

expands the populations linearly by adding 500 individuals to each subpopulation at each generation. These events happen at each generation starting from generation 500.

Simiarly, you can split, merge, and resize subpopulations using events `SplitEvent`, `MergeEvent`, and `ResizeEvent`. For example,

```
SplitEvent(subPops='AF', sizes=[500, 500], names=['AF', 'EU'], at=-4000)
```

splits an ancestral population named AF to two populations AF and EU at 4000 generations before the end of the demographic model. The AF population will be expanded automatically if it does not have 1000 individuals.

Finally, an `AdmixtureEvent` mix two or more subpopulations by certain proportions, and either create a new subpopulation or replace an existing subpopulation. In particular,

```
AdmixtureEvent(subPops=['MX', 'EU'], at=-10, sizes=[0.4, 0.6], name='MXL')
```

creates a new admixed population called MXL with 40% of individuals from the MX population, and the rest from the EU population. The admixture process happens once and follows an Hybrid Isolation model. Alternatively,

```
AdmixtureEvent(subPops=['MX', 'EU'], begin=-10, sizes=[0.8, 0.2], toSubPop='MX')
```

will create an admixed population with 80% MX and 20% EU individuals for 10 generations. Because 20% of the admixed population will be replaced by individuals from the EU population, this models a continuous gene flow model of admixture. If you would like to control the exact size of the admixed population, you can specify the number of individuals as integer numbers instead of proportions:

```
AdmixtureEvent(subPops=['MX', 'EU'], begin=-10, sizes=[int(1400*0.8), int(1400*0.2)], toSubPop='MX')
```

Note that the type of elements in parameter `sizes` is important, `1.` stands for all subpopulation and `1` stands for one individual from it.

Example6.13 defines the same model as 6.12 using an event based demographic model. The result is depicted in Figure 6.7. These two models look similar but the event-based model does not have the same final population sizes as the previous model. This is because the population size of the previous model was calculated by $N(t) = N(0)\exp(rt)$ whereas the event based model was calculated using $N(t) = \text{round}(N(t-1) * (1+r))$ for each generation, and the integer rounding error accumulates over time.

Listing 6.13: A event-based demographic model

```
>>> import simuPOP as sim
```

```
>>> from simuPOP.demography import *
>>> import math
>>> model = EventBasedModel(
...     N0=(1000, 'Ancestral'),
...     T=250,
...     events=[
...         ResizeEvent(at=50, sizes=200),
...         ResizeEvent(at=60, sizes=1000),
...         SplitEvent(sizes=[0.4, 0.6], names=['P1', 'P2'], at=200),
...         ExpansionEvent(rates=[math.log(4000/400)/50, math.log(5000/600)/50], begin=200)
...     ]
... )
>>> #
>>> # model.init_size returns the initial population size
>>> # migrate_to is required for migration
>>> pop = sim.Population(size=model.init_size, loci=1,
...     infoFields=model.info_fields)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.RandomMating(subPopSize=model),
...     finalOps=
...         sim.Stat(alleleFreq=0, vars=['alleleFreq_sp']),
...     gen=model.num_gens
... )
250L
>>> # print out population size and frequency
>>> for idx, name in enumerate(pop.subPopNames()):
...     print('%s (%d): %.4f' % (name, pop.subPopSize(name),
...         pop.dvars(idx).alleleFreq[0][0]))
...
P1 (4000): 0.6506
P2 (4800): 0.6741
>>> # get a visual presentation of the demographic model
>>> model.plot('log/demoEventModel.png',
...     title='A event-based bottleneck + exponential growth demographic model')
A event-based bottleneck + exponential growth demographic model
0: 1000 (Ancestral)
50: 200 (Ancestral)
60: 1000 (Ancestral)
200: 419 (P1), 625 (P2)
201: 439 (P1), 652 (P2)
202: 459 (P1), 680 (P2)
203: 481 (P1), 709 (P2)
204: 504 (P1), 739 (P2)
205: 527 (P1), 770 (P2)
206: 552 (P1), 803 (P2)
207: 578 (P1), 837 (P2)
208: 605 (P1), 872 (P2)
209: 634 (P1), 909 (P2)
210: 664 (P1), 948 (P2)
211: 695 (P1), 988 (P2)
```

```
212: 728 (P1), 1030 (P2)
213: 762 (P1), 1074 (P2)
214: 798 (P1), 1120 (P2)
215: 836 (P1), 1167 (P2)
216: 875 (P1), 1217 (P2)
217: 916 (P1), 1268 (P2)
218: 960 (P1), 1322 (P2)
219: 1005 (P1), 1378 (P2)
220: 1052 (P1), 1437 (P2)
221: 1102 (P1), 1498 (P2)
222: 1154 (P1), 1562 (P2)
223: 1208 (P1), 1628 (P2)
224: 1265 (P1), 1697 (P2)
225: 1325 (P1), 1769 (P2)
226: 1387 (P1), 1844 (P2)
227: 1452 (P1), 1923 (P2)
228: 1521 (P1), 2004 (P2)
229: 1592 (P1), 2089 (P2)
230: 1667 (P1), 2178 (P2)
231: 1746 (P1), 2271 (P2)
232: 1828 (P1), 2367 (P2)
233: 1915 (P1), 2467 (P2)
234: 2005 (P1), 2572 (P2)
235: 2099 (P1), 2681 (P2)
236: 2198 (P1), 2795 (P2)
237: 2302 (P1), 2914 (P2)
238: 2410 (P1), 3038 (P2)
239: 2524 (P1), 3167 (P2)
240: 2643 (P1), 3301 (P2)
241: 2767 (P1), 3441 (P2)
242: 2898 (P1), 3588 (P2)
243: 3034 (P1), 3740 (P2)
244: 3177 (P1), 3899 (P2)
245: 3327 (P1), 4064 (P2)
246: 3484 (P1), 4237 (P2)
247: 3648 (P1), 4417 (P2)
248: 3820 (P1), 4604 (P2)
249: 4000 (P1), 4800 (P2)
>>>
```

6.3.5 Predefined demographic models for human populations

The `simuPOP.demography` module currently defines the following models

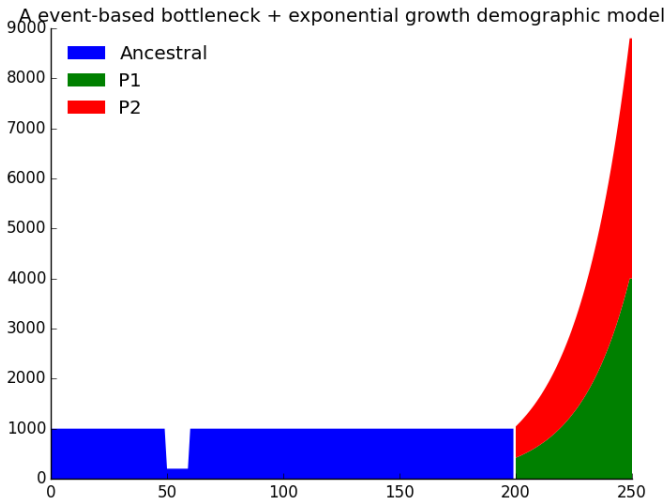
- Out of Africa model for YRI, CEU and CHB populations (6.8),

  ```
  OutOfAfricaModel(10000).plot('OutOfAfrica.png')
  ```

- The settlement of new world model for Mexican American (6.9) (Gutenkunst, 2009, PLoS Genetics). In this model, the simulated CHB and MX populations are mixed to produce an admixed population at the last generation.

  ```
  SettlementOfNewWorldModel(10000).plot('SettlementOfNewWorld.png')
  ```

Figure 6.7: A event-based demographic model



- The demographic model developed by cosi (Schaffner, 2005, genome research).

```
CosiModel(20000).plot('Cosi.png')
```

These functions all accept a parameter scale. If specified, it will scale all population sizes and generation numbers by the specified scaling factor. For example

```
CosiModel(20000, scale=10)
```

will result in a demographic model that evolves 2000 instead of 20000 generations, with all population sizes reduced by a factor of 10. Note that the burn-in period of the examples above are relatively short and you might need to use a longer burn-in period (e.g. T=100,000 generations for a burn-in period of about 80,000 generations).

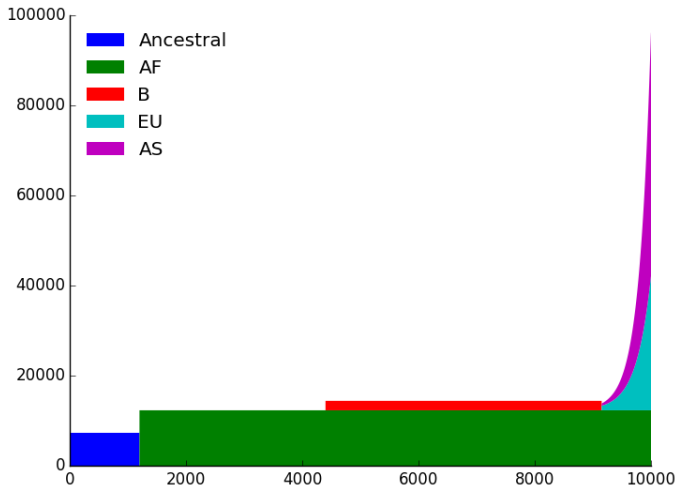
6.3.6 Demographic model without predefined generations to evolve *

All migration models accept one or more operators that will be applied to the population before population population changes are applied. The most frequently application of this operator is to pass a migrator to the model, but we can also pass an operator to terminate a demographic model under certain conditions. For example, Example 6.14 defines a demographic model that starts with a burn-in stage with indefinite size and will stop if the average allele frequency at segregating sites exceeds 0.1. It splits to two equally sized subpopulations and expand rate a rate of 0.01 to size 2000 and 5000 respectively.

Listing 6.14: A demographic model with a terminator

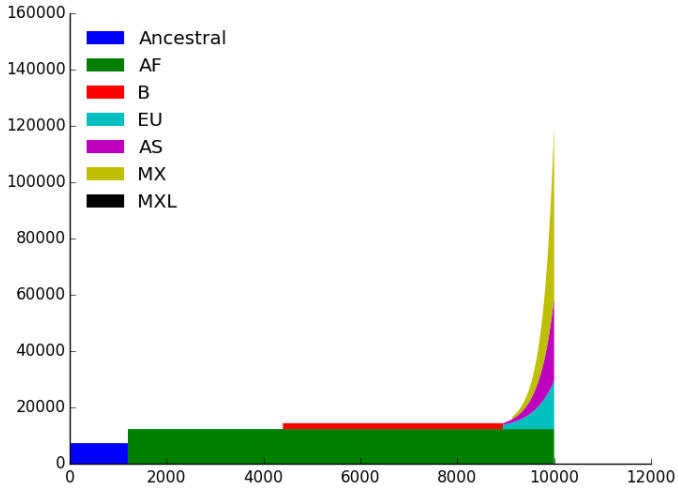
```
>>> import simuPOP as sim
simuPOP Version 1.1.6 : Copyright (c) 2004-2011 Bo Peng
Revision 4972 (Oct 30 2015) for Python 2.7.10 (64bit, 0thread)
Random Number Generator is set to mt19937 with random seed 0x9fffda4c77af39e9.
This is the standard short allele version with 256 maximum allelic states.
For more information, please visit http://simupop.sourceforge.net,
or email simupop-list@lists.sourceforge.net (subscription required).
>>> import simuPOP.demography as demo
>>>
>>> model = demo.MultiStageModel([
```

Figure 6.8: Out of Africa model for YRI, CEU, and CHB populations




```
...     demo.InstantChangeModel(N0=1000,
...         ops=[
...             sim.Stat(alleleFreq=sim.ALL_AVAIL, numOfSegSites=sim.ALL_AVAIL),
...             # terminate if the average allele frequency of segregating sites
...             # are more than 0.1
...             sim.TerminateIf('sum([x[1] for x in alleleFreq.values() if '
...                 'x[1] != 0])/(1 if numOfSegSites==0 else numOfSegSites) > 0.1')
...         ]
...     ),
...     demo.ExponentialGrowthModel(N0=[0.5, 0.5], r=0.01, NT=[2000, 5000])
...     ]
... )
>>>
>>> pop = sim.Population(size=model.init_size, loci=100)
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=sim.SNPMutator(u=0.001, v=0.001),
...     matingScheme=sim.RandomMating(subPopSize=model),
...     postOps=[
...         sim.Stat(alleleFreq=sim.ALL_AVAIL, numOfSegSites=sim.ALL_AVAIL,
...             popSize=True, step=50),
...         sim.PyEval(r'"%d: %s, %.3f\n" % (gen, subPopSize, sum([x[1] for x '
...             'in alleleFreq.values() if x[1] != 0])/(1 if numOfSegSites == 0 '
...             'else numOfSegSites))', step=50)
...     ],
... )
0: [1000], 0.001
50: [1000], 0.047
100: [1000], 0.089
150: [738, 738], 0.128
200: [1218, 1218], 0.166
250: [2000, 2007], 0.199
300: [2000, 3310], 0.230
```

Figure 6.9: Settlement of New World model for Mexican America population




```
343L
>>>
```

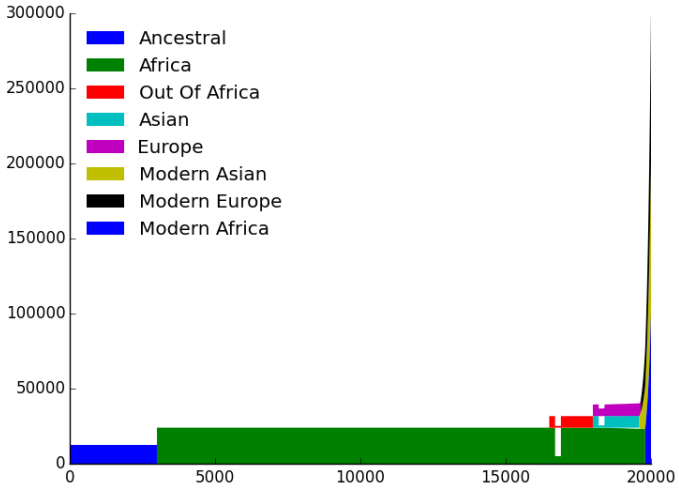
6.4 Module `simuPOP.plotter`

The `simuPOP.plotter` module defines a few utility functions and Python operators that help you plot variables and information fields during evolution. A number of operators are defined that

- Operator `plotter.VarPlotter`: Plot a dynamically evaluated expression with its history. Each expression and its history form a line in the plot. Multiple lines will be plotted for multiple replicates and/or for each element of the expression (if the evaluated value of the expression is a sequence), with options to separate lines to different subplots.
- Operator `plotter.ScatterPlotter`: Plot individuals in specified (virtual) subpopulations using values at two information fields as x and y axes. individuals belonging to different (virtual) subpopulations will be plotted with different colors and shapes.
- Operator `plotter.InfoPlotter`: Using a R function such as `hist` and `qqnorm` to plot one or more information fields of individuala in one or more (virtual) subpopulations. Two specialized operators `plotter.HistPlotter` and `plotter.QQPlotter` are provided to plot the histograms and qq plots. Other functions could also be used, and it is even possible to draw a figure completely by your own (with stratified data provided to you by this operator).
- Operator `plotter.BoxPlotter`: This operator uses R function `boxplot` to plot boxplots of data of one or more information fields of individuals in one ore more (virtual) subpopulations. The whiskers could be grouped by information field or subpopulations.

These operators are derived from class `PyOperator` and call R plot functions when they are applied to a population. For example, operator `plotter.VarPlotter` collects expression values and use functions `plot` and `lines` to plot the data, with help from other functions such as `par` (device property), `dev.print` (save figure to files) and `legend` (add legend). Some functions are called multiple times for different replicate, subpopulation or information fields.

Figure 6.10: Demographic models for African, Asian and European populations (cosi)



6.4.1 Derived keyword arguments *

One of the most interesting feature of this module is its use of derived keyword parameters to send arbitrary parameters to the underlying R functions, which usually accept a large number of parameters to customize every aspect of a figure. A **derived keyword argument** is an argument that is prefixed with a function name and/or suffixed by an iterator name. The former specifies to which underlying R function this parameter will be passed to; the latter allows the users to specify a list of values that will be passed, for example, to lines representing different replicates. For example, parameter `par_mar=[1]*4` will pass `mar=[1]*4` to R function `par`, and `lty_rep=[1, 2, 3]` will pass `lty=1`, `lty=2` and `lty=3` to different replicates. A class usually has one or two default functions (such as `plot`, `lines`) to which keyword aguments without function prefix will be sent.

In addition, the values of these keyword arguments could vary during evolution. More specifically, if the value is a string with a leading exclamation mark (`!`), the remaining string will be considered as an expression. This expression will be evaluated against the current population during evolution and the return value will become the value of the parameter at that generation. For example, keyword parameter `main="!'Allele frequency at generation %d' % gen"` will become `main='Allele frequency at generation 10'` at generation 10.

6.4.2 Plot of expressions and their histories (operator `plotter.VarPlotter`)

Class `plotter.VarPlotter` plots the current and historical values of a Python expression (`expr`), which are evalulated (against each population's local namespace) and saved during evolution. The return value of the expression can be a number or a sequence, but should have the same type and length across all replicates and generations. Histories of each value (or each item in the returned sequence) of each replicate form a line, with generation numbers as its x-axis. Number of lines will be the number of replicates multiplied by dimension of the expression. Although complete histories are usually saved, you can use parameter `win` to save histories only within the last `win` generations.

simuPOP version 1.1.6 and earlier supports both rpy and matplotlib as the underlying plotting library. However, because of bugs in rpy2 and difficulties in supporting rpy, rpy2 and matplotlib, rpy/rpy2 support is removed in simuPOP 1.1.7. Please use simuPOP 1.1.6 if you are interested in using rpy2.

Except for the first generation where no line could be drawn, a figure will be drawn after this operator is applied to the last specified replicate (parameter `reps` could be used to specify a subset of replicates). For example, although linkage disequilibrium values between the first two loci are evaluated and saved at the end of generations 0, 5, 10, ..., (`step=5`) figures are only drawn at generations 40 and 80 (`update=40`) in Exampe 6.15. This example also demonstrates the

use of parameters saveAs and legend. By given a filename rpy.png to parameter saveAs, this operator will save figures (named rpy_40.png and rpy_80.png) after they are drawn.

Listing 6.15: Use rpy or matplotlib to plot an expression

```
>>> import simuPOP as sim
>>> from simuPOP.plotter import VarPlotter
>>> pop = sim.Population(size=1000, loci=2)
>>> simu = sim.Simulator(pop, rep=3)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[1, 2, 2, 1])
...     ],
...     matingScheme=sim.RandomMating(ops=sim.Recombinator(rates=0.01)),
...     postOps=[
...         sim.Stat(LD=[0, 1]),
...         #
...         VarPlotter('LD[0][1]', step=5, update=40, saveAs='log/varplot.png',
...             legend=['Replicate %d' % x for x in range(3)],
...             set_ylabel_ylabel='LD between marker 1 and 2',
...             set_title_label='LD decay',
...             set_ylim_bottom=0, set_ylim_top=0.25,
...             plot_linestyle_rep=['-', ':', '-.'],
...         ),
...     ],
...     gen=100
... )
(100L, 100L, 100L)
```

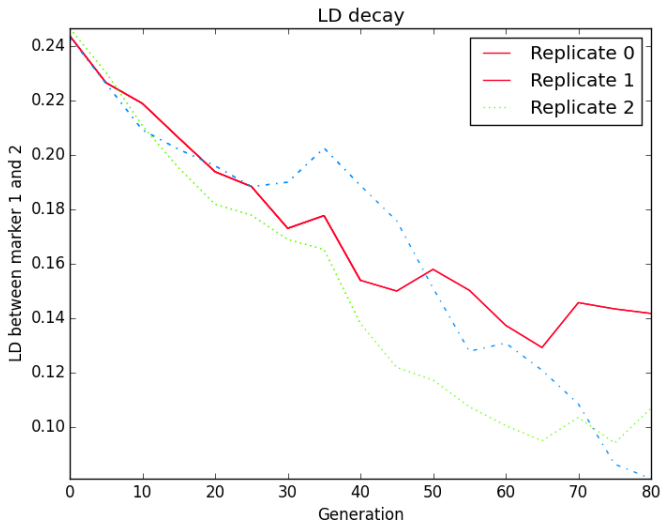
Parameters after legend (xlab, ylab, ylim, main, ...) deserve more attention here. These parameters are derived keyword arguments because they are not defined by VarPlotter. Parameters without prefix are passed directly to the R functions plot and line. They could be used to customize line type (lty), color (col), title (main), limits of x and y axes (xlim and ylim) and many other graphical features (see R manual for details). If multiple lines are drawn, a list of values could be applied to these lines if you add _rep (for each replicate) or _dim (for each item of a sequence) after the name of the parameter. For example, lty_rep=[1, 2, 3] is used in Example 6.15 to pass parameters lty=1, lty=2 and lty=3 to lines for three replicates. Suffix _repdim can also be used to specify values for every replication and dimension. Figure 6.11 displayed rpy_80.png that is saved at generation 80 for this example.

If the expression is multidimensional, the number of lines can be large and it is often desired to separate these lines into subplots. This can be done by parameters byRep or byDim. The former plots lines replicate by replicate and the latter does it dimension by dimension. For example, Example 6.16 and 6.17 both have three replicates and the expression has allele frequency for four loci. The total number of lines is therefore 12. In Example 6.16, these lines are separated to three subplots, replicate by replicate, with different titles (parameter main_rep). In each subplot, allele frequency trajectories (histories) for different loci are plotted in different color (parameter col_dim). The last saved figure (rpy_byRep_90.png) is displayed in Figure 6.12. In Example 6.17, these lines are separated to four subplots, locus by locus, with different titles (parameter main_dim). In each subplot, allele frequency trajectories (histories) for different loci are plotted in different color (parameter col_rep) and line type (parameter lty_rep). The last saved figure (rpy_byDim_90.png) is displayed in Figure 6.13.

Listing 6.16: Separate figures by replicate

```
>>> import simuPOP as sim
>>> import simuPOP as sim
>>> from simuPOP.plotter import VarPlotter
>>> pop = sim.Population(size=1000, loci=1*4)
>>> simu = sim.Simulator(pop, rep=3)
```

Figure 6.11: rpy_80.png saved at generation 80 for Example 6.15




```
>>> simu.evolve(
...     initOps=[sim.InitSex()] +
...         [sim.InitGenotype(freq=[0.1*(x+1), 1-0.1*(x+1)], loci=x) for x in range(4)],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=range(4)),
...         VarPlotter('[alleleFreq[x][0] for x in range(4)]', byRep=True,
...             update=10, saveAs='log/varplot_byRep.png',
...             figure_figsize=(10, 8),
...             legend=['Locus %d' % x for x in range(4)],
...             set_ylabel_ylabel='Allele frequency',
...             set_ylim_bottom=0, set_ylim_top=1,
...             set_title_label_rep=['Genetic drift, replicate %d' % x for x in range(3)],
...         ),
...     ],
...     gen=100
... )
(100L, 100L, 100L)
```

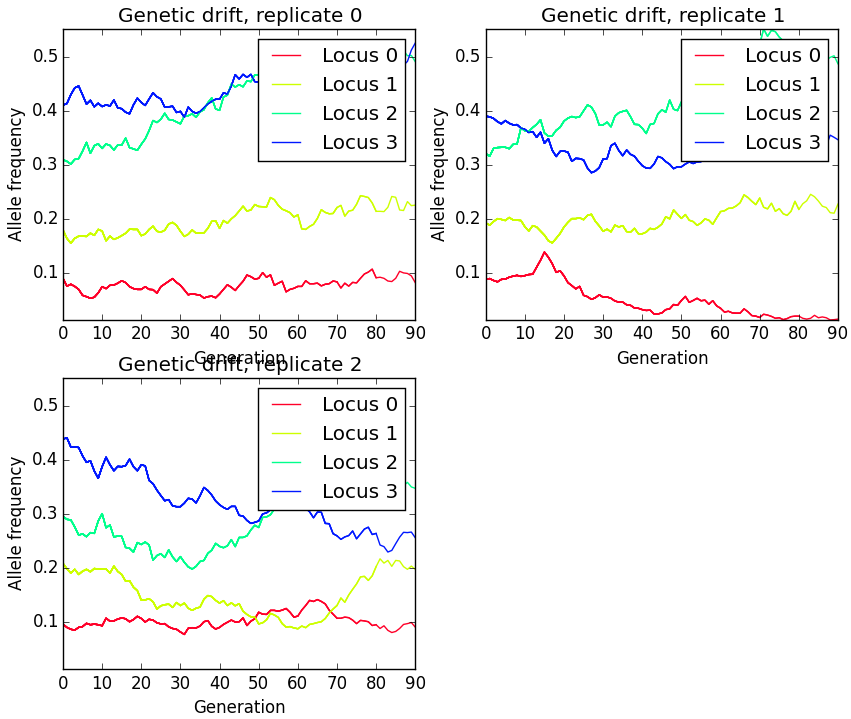
Example 6.17 also demonstrates some advanced features of this plotter that allow further customization of the figures. More specifically,

- Function-specific parameters can be passed to the underlying R function by prefixing function names to parameter names. For example, `plot_axis=False` is used to pass `axis=False` to the `r.plot` function (and not to function `lines` which does not accept this parameter).
- Several hook function can be defined and passed to parameters `preHook`, `postHook` and `plotHook`, which will be called, respectively, before a figure is drawn, after a figure is drawn, and after each `r.plot` call. Example 6.17 uses a `plotHook` function to draw axes of the plots and call `mtext` to add texts to the margins.

Listing 6.17: Separate figures by Dimension

```
>>> import simuPOP as sim
>>> import simuPOP as sim
```

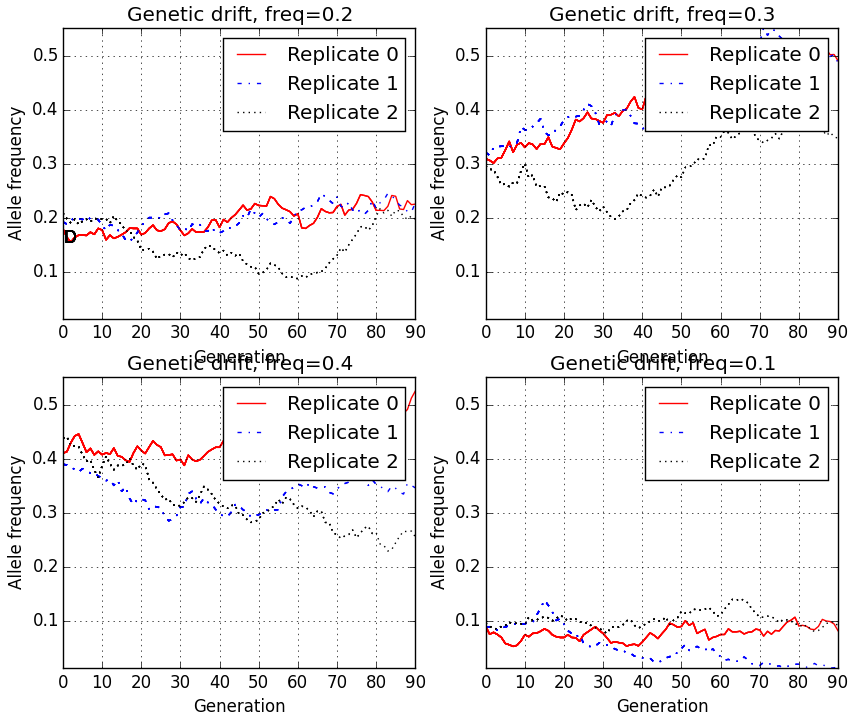
Figure 6.12: Allele frequency trajectories separated by replicates




```
>>> from simuPOP.plotter import VarPlotter
>>> pop = sim.Population(size=1000, loci=1*4)
>>> simu = sim.Simulator(pop, rep=3)
>>> def rpy_drawFrame(r, dim=None, **kwargs):
...     '''Draw a frame around subplot dim. Parameter r is defined in the rpy
...     module and is used for calling R functions. Parameter dim is the dimension
...     index. Other parameters are ignored.
...     '''
...     r.axis(1)
...     r.axis(2)
...     r.grid()
...     r.mtext({0:'A', 1:'B', 2:'C', 3:'D'}[dim], adj=1)
...
>>> def mat_drawFrame(ax, dim=None, **kwargs):
...     '''Draw a frame around subplot dim. Parameter r is defined in the rpy
...     module and is used for calling R functions. Parameter dim is the dimension
...     index. Other parameters are ignored.
...     '''
...     ax.grid()
...     ax.text(0.5, 0.8, {0:'A', 1:'B', 2:'C', 3:'D'}[dim])
...
>>> simu.evolve(
...     initOps=[sim.InitSex()]+
...         [sim.InitGenotype(freq=[0.1*(x+1), 1-0.1*(x+1)], loci=x) for x in range(4)],
...     matingScheme=sim.RandomMating(),
```

```
...     postOps=[
...         sim.Stat(alleleFreq=range(4)),
...         VarPlotter('[alleleFreq[x][0] for x in range(4)]', byDim=True,
...             update=10, saveAs='log/varplot_byDim.png',
...             legend=['Replicate %d' % x for x in range(3)],
...             set_ylabel_ylabel='Allele frequency',
...             set_ylim_bottom=0, set_ylim_top=1,
...             set_title_label_dim=['Genetic drift, freq=%.1f' % ((x+1)*0.10) for x in range(4)],
...             plot_c_rep=['red', 'blue', 'black'],
...             plot_linestyle_rep=['-', '-.', ':'],
...             figure_figsize=(10,8),
...             plotHook = mat_drawFrame,
...         ),
...     ],
...     gen=100
... )
/Users/bpeng1/bin/anaconda/lib/python2.7/site-packages/matplotlib/axes/_subplots.py:69: MatplotlibDeprecationWarning: The use c
  mplDeprecation)
(100L, 100L, 100L)
```

Figure 6.13: Allele frequency trajectories separated by loci



6.4.3 Scatter plots (operator `plotter.ScatterPlotter`)

Operator `plotter.ScatterPlotter` plots individuals in all or selected (virtual) subpopulations in a 2-D plot, using values at two information fields as their x- and y-axis. In the most simplified form,

```
InfoPlotter(infoFields=['x', 'y'])
```

will plot all individuals according their values of information fields `x` and `y`. Additional parameters such as pch, `col`, and `cex` can be used to control the shape, color and size of the points.

What makes this operator useful is its ability to differentiate points (individuals) by (virtual) subpopulations (VSPs). If a list of VSPs are given, points representing individuals from these VSPs will be plotted with different colors and shapes. Because simulations that keep track of multiple information fields are usually complicated, let us simulate something interesting and examine Example 6.18 in details.

At the beginning of this example, all individuals are scattered randomly with `x` and `y` being their physical locations. We use `anc` to record Individual ancestry and assign 0 and 1 each to half of the population. During evolution,

- Offspring ancestry values are the average of their parents.
- Offspring with higher ancestry value tend to move to the right. More specifically, locations of an offspring will be
$$\frac{(x_1+x_2)}{2}+N\left(\frac{a_1+a_2}{2}-0.5,0.1\right),\frac{(y_1+y_2)}{2}+N\left(0,0.1\right)$$
where (x_1,y_1) and (x_2,y_2) are locations of parents, a_1 and a_2 are ancestry values of the parents, and $N\left(a,b\right)$ are a random number with normal distribution.

An `ScatterPlotter` is used to plot the physical location of all individuals. Individual ancestries are divided into five regions (0, 0.2, 0.4, 0.6, 0.8, 1) indicated by small to larger points. MALE and female individuals are plotted by different symbol. This scripts uses the following techniques:

- Set individual information fields randomly using `setIndInfo`.
- Define virtual subpopulations using a `InfoSplitter`.
- Use `PyTagger` to calculate offspring information fields from parental fields.
- Mark individuals in different VSPs using parameters `col_sp` and `cex_sp`.
- Use `plot_axes=False` and `par_mar=[0, 0, 2, 0]` to pass parameters `axes=False` and `mar=[0, 0, 2, 0]` to functions `plot` and `par` respectively.

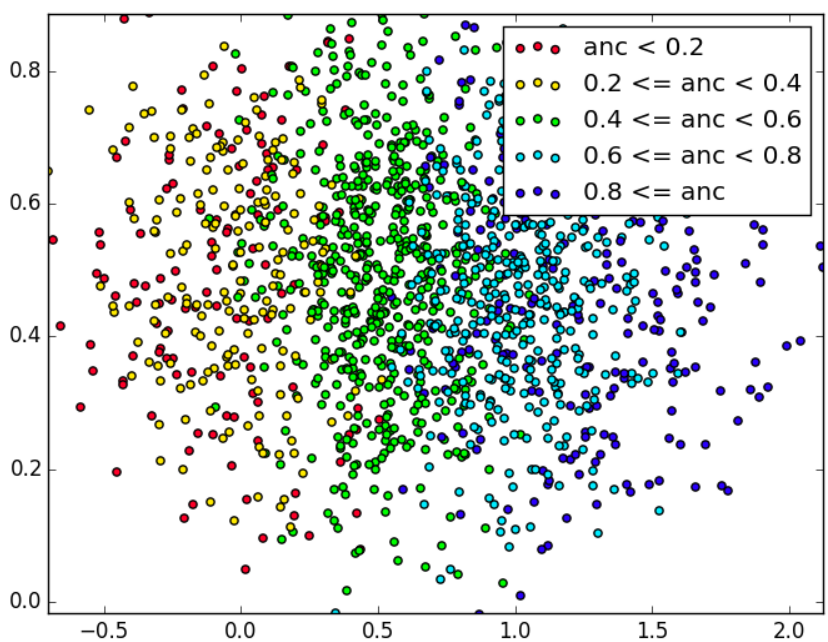
VSPs 0 and 4 appear at the beginning of generation 0, VSP 2 appears at the end of generation 0, and VSP 1 and 3 appear at the end of generation 1. Figure 6.14 displays a figure at the begging of generation 2.

Listing 6.18: Use ScatterPlotter to plot ancestry of individuals with geographic information.

```
>>> import simuPOP as sim
>>> import simuPOP as sim
>>> from simuPOP.plotter import ScatterPlotter
>>> import random
>>> pop = sim.Population([500], infoFields=['x', 'y', 'anc'])
>>> # Defines VSP 0, 1, 2, 3, 4 by anc.
>>> pop.setVirtualSplitter(sim.InfoSplitter('anc', cutoff=[0.2, 0.4, 0.6, 0.8]))
>>> #
>>> def passInfo(x, y, anc):
...     'Parental fields will be passed as tuples'
...     off_anc = (anc[0] + anc[1])/2.
...     off_x = (x[0] + x[1])/2 + random.normalvariate(off_anc - 0.5, 0.1)
...     off_y = (y[0] + y[1])/2 + random.normalvariate(0, 0.1)
...     return off_x, off_y, off_anc
...
```

```
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         # random geographic location
...         sim.InitInfo(random.random, infoFields=['x', 'y']),
...         # anc is 0 or 1
...         sim.InitInfo(lambda : random.randint(0, 1), infoFields='anc')
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.PyTagger(passInfo)]),
...     postOps=[
...         ScatterPlotter(['x', 'y'],
...             saveAs = 'log/ScatterPlotter.png',
...             subPops = [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)],
...             set_ylim_bottom = 0, set_ylim_top=1.2,
...             set_title_label = "!'Ancestry distribution of individuals at generation %d' % gen",
...             legend = ['anc < 0.2', '0.2 <= anc < 0.4', '0.4 <= anc < 0.6',
...                 '0.6 <= anc < 0.8', '0.8 <= anc'],
...         ),
...
...     ],
...     gen = 5,
... )
5L
```

Figure 6.14: Plot of individuals with ancestry marked by different colors



6.5 Module simuPOP.sampling

6.5.1 Introduction

Sampling, in simuPOP term, is the action of extracting individuals from a large, potentially multi-generational, population according to certain criteria. the simuPOP.sampling module provides several classes and functions and allows you to define more complicated sampling schemes by deriving from its these class. For example, you can use drawRandomSample(pop, size=100) to select 100 random individuals from a population, or use drawAffectedSibpairSample(pop, families=100) to select 100 pairs of affected invididuals with their parents from a multi-generational population, or a age-structured population with parents and offspring in the same generation.

The simuPOP.sampling module currently support random, case control, affected sibpair, nuclear family and three-generation family sampling types, and a combined sampling type that allows you to draw different types of samples. For each sampling type X, a sampler class and two functions DrawXSample and DrawXSamples are provided The first function returns a population with all sampled individuals and the second function returns a list of sample populations.

If you would like to define your own sampling type, you can derive your sampler from one of the existing sampler classes. These sampler classes provide member functions prepareSample, drawSample and drawSamples and you typically only need to extend prepareSample of an appropriate base class.

6.5.2 Sampling individuals randomly (class RandomSampler, functions drawRandomSample and drawRandomSamples)

Functions drawRandomSample and drawRandomSamples draw random invidiauls from a given population. If a simple number is given to parameter size, population structure will be ignored so individuals will be drawn from all subpopulations. If a list of numbers are given, this function will draw specified numbers of individuals from each subpopulation. This function does not need parental information. If your population does not have an ID field, you will not be able to locate extracted individuals in the original population.

Example 6.19 demonstrates how to draw a random sample from the whole population, and from each subpopulation. Because sample populations keep the population structure of the source population (this might change when parameter subPops is used, see a later section for details), we can use sample.subPopSizes() to check how many individuals are sampled from each subpopulation.

Listing 6.19: Draw random samples from a structured population

```
>>> import simuPOP as sim
>>> from simuPOP.sampling import drawRandomSample
>>> pop = sim.Population([2000]*5, loci=1)
>>> # sample from the whole population
>>> sample = drawRandomSample(pop, sizes=500)
>>> print(sample.subPopSizes())
(104L, 105L, 110L, 81L, 100L)
>>> # sample from each subpopulation
>>> sample = drawRandomSample(pop, sizes=[100]*5)
>>> print(sample.subPopSizes())
(100L, 100L, 100L, 100L, 100L)
```

6.5.3 Sampling cases and controls (class CaseControlSampler, functions CaseControlSample and CaseControlSamples)

Functions drawCaseControlSample and drawCaseControlSamples draw cases (affected individuals) and controls (unaffected invidiauls) from a given population. If a simple number is given to parameter cases and controls, population

structure will be ignored so individuals will be drawn from all subpopulations. If a list of numbers are given, this function will draw specified number of cases and controls from each subpopulation.

Example 6.20 demonstrates how to draw multiple case-control samples from a population, and perform case-control assocition tests using the `stat` function.

Listing 6.20: Draw case control samples from a population and perform association test

```
>>> import simuPOP as sim
>>> from simuPOP.sampling import drawCaseControlSamples
>>> pop = sim.Population([10000], loci=5)
>>> sim.initGenotype(pop, freq=[0.2, 0.8])
>>> sim.maPenetrance(pop, loci=2, penetrance=[0.11, 0.15, 0.20])
>>> # draw multiple case control sample
>>> samples = drawCaseControlSamples(pop, cases=500, controls=500, numOfSamples=5)
>>> for sample in samples:
...     sim.stat(sample, association=range(5))
...     print(', '.join(['%.6f' % sample.dvars().Allele_ChiSq_p[x] for x in range(5)]))
...
0.694748, 0.333041, 0.001039, 0.078127, 0.774085
0.261750, 0.954592, 0.031830, 0.737788, 0.865679
0.954949, 0.371093, 0.092487, 0.622153, 0.075739
0.654721, 0.433848, 0.002859, 0.696375, 0.956630
0.439721, 1.000000, 0.069651, 0.471087, 0.238199
```

6.5.4 Sampling Pedigrees (functions `indexToID` and `plotPedigree`)

If your sampling scheme involves parental information, you need to prepare your population so that it has

- an ID field (usually `'ind_id'`) that stores a unique ID for each individual.
- two information fields (usually `'father_id'`, and `'mother_id'`) that stores the ID of parents of each individual. Although simuPOP supports one-parent Pedigrees, this feature will not be discussed in this guide.

The preferred method to prepare such a population is to add information fields `ind_id`, `father_id` and `mother_id` to a population and track ID based Pedigrees during evolution. More specifically, you can use operators `IdTagger` and `PedigreeTagger` to assign IDs and record parental IDs of each offspring during mating. This method supports age-structured population when parents and offspring can be stored in the same generation.

You can also use information fields `father_idx` and `mother_idx` and operator `ParentsTagger` to track indexes of parents in the parental generations. Before sampling, you can use function `sampling.indexToID` to add needed information fields and convert index based parental relationship to ID based relationshop. Because parents have to stay in ancestral generations, this method does not support age-structured population.

If you have R and rpy installed on your system, you can install the `kinship` library of R and use it to analyze Pedigree. The `simuPOP.sampling` module provides a function `plotPedigree` to use this library to plot Pedigrees. Example **??** demonstrates how to use function sampling.indexToID to prepare a pedigree and how to use sampling.DrawPedigree to plot it.

Figure **??** plots a small three-generational population with 15 individuals at each generation. It is pretty clear that random mating produces bad pedigree structure because it is common that one parent would have multiple spouses.

6.5.5 Sampling affected sibpairs (class `AffectedSibpairSampler`, functions `drawAffectedSibpairSample(s)`)

An affected sibpair family consists of two parents and their affected offspring. Such families are useful in linkage analysis because of high likelihood of shared disease predisposing alleles between siblings. `simuPOP.sampling` module provides functions `drawAffectedSibpairSample` and `drawAffectedSibpairSamples` to draw such families from a population. Example 6.21 draws two affected sibpair from the pedigree created in Example **??**, with samples plotted in Figure **??**.

Listing 6.21: Draw affected sibpairs from a population

```
>>> import simuPOP as sim
>>> from simuPOP.sampling import indexToID
>>> pop = sim.Population(size=15, loci=5, infoFields=['father_idx', 'mother_idx'], ancGen=2)
>>> pop.evolve(
...     preOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.7, 0.3]),
...     ],
...     matingScheme=sim.RandomMating(numOffspring=(sim.UNIFORM_DISTRIBUTION, 2, 4),
...         ops=[sim.MendelianGenoTransmitter(), sim.ParentsTagger()]),
...     postOps=sim.MaPenetrance(loci=3, penetrance=(0.1, 0.4, 0.7)),
...     gen = 5
... )
5L
>>> indexToID(pop, reset=True)
>>> # three information fields were added
>>> print(pop.infoFields())
('father_idx', 'mother_idx', 'ind_id', 'father_id', 'mother_id')
>>> # save this population for future use
>>> pop.save('log/pedigree.pop')
>>>
>>> from simuPOP.sampling import drawAffectedSibpairSample
>>> pop = sim.loadPopulation('log/pedigree.pop')
>>> sample = drawAffectedSibpairSample(pop, families=2)
Warning: number of requested Pedigrees 2 is greater than what exists (0).
Warning: not enough non-overlapping Pedigrees are found (requested 2, found 0).
```

6.5.6 Sampling nuclear families (class `NuclearFamilySampler`, functions `drawNuclearFamilySample` and `drawNuclearFamilySamples`)

A nuclear family consists of two parents and their offspring. Functions `drawNuclearFamilySample` and `drawNuclearFamilySamples` to draw such families from a population, with restrictions on number of offspring, number of affected parents and number of affected offspring. Although fixed numbers could be given, a range with minimal and maximal acceptable numbers are usually provided. Example 6.22 draws two nuclear families from the pedigree created in Example **??**. The samples are plotted in Figure **??**.

Listing 6.22: Draw nuclear families from a population

```
>>> import simuPOP as sim
>>> from simuPOP.sampling import drawNuclearFamilySample
>>> pop = sim.loadPopulation('log/pedigree.pop')
>>> sample = drawNuclearFamilySample(pop, families=2, numOffspring=(2,4),
...     affectedParents=(1,2), affectedOffspring=(1, 3))
Warning: number of requested Pedigrees 2 is greater than what exists (0).
Warning: not enough non-overlapping Pedigrees are found (requested 2, found 0).
```

```
>>> # try to separate two families?
>>> sample.asPedigree()
>>> #= sim.Pedigree(sample, loci=sim.ALL_AVAIL, infoFields=sim.ALL_AVAIL)
>>> sample.addInfoFields('ped_id')
>>> # return size of families
>>> sz = sample.identifyFamilies(pedField='ped_id')
>>> print(sz)
()
>>> ped1 = sample.extractIndividuals(IDs=0, idField='ped_id')
>>> # print the ID of all individuals in the first pedigree
>>> print([ind.ind_id for ind in ped1.allIndividuals()])
[]
```

6.5.7 Sampling three-generation families (class `ThreeGenFamilySampler`, functions `drawThreeGenFamilySample` and `drawThreeGenFamilySamples`)

A three-generation family consists of two parents, their common offspring, offspring's spouses, and their common offspring (grandchidren). individuals in sampled families have either no or two parents. Functions `drawThreeGenFamilySample` and `drawThreeGenFamilySamples` to draw such families from a population, with restrictions on number of offspring, total number of individuals and number of affected individuals in the Pedigree. These parameters (`numOffspring`, `pedSize` and `numAffected`) could be a fixed number of a range with minimal and maximal acceptable numbers. Example 6.22 draws two three generation families from the pedigree created in Example **??**. The samples are plotted in Figure **??**.

Listing 6.23: Draw three-generation families from a population

```
>>> import simuPOP as sim
>>> from simuPOP.sampling import drawThreeGenFamilySample
>>> pop = sim.loadPopulation('log/pedigree.pop')
>>> sample = drawThreeGenFamilySample(pop, families=2, numOffspring=(1, 3),
...     pedSize=(8, 15), numOfAffected=(2, 5))
```

6.5.8 Sampling different types of samples (class `CombinedSampler`, functions `drawCombinedSample` and `drawCombinedSamples`)

Samples in real world studies sometimes do not have uniform types so it is useful to draw samples of different types from the same population. Although it is possible to draw samples using different functions and combine them, handling of overlapping individuals, namely individuals who are chosen by multiple samplers, can be a headache. The combined sampler of `simuPOP.sampling` is designed to overcome this problem. This sampler takes a list of sampler objects and apply them to a population sequentially. The extracted sample will not have overlapping individuals.

Example 6.24 draws an affected sibpair family and a nuclear family from the pedigree created in Example **??**. The samples are plotted in Figure 6.24.

Listing 6.24: Draw different types of samples from a population

```
>>> import simuPOP as sim
>>> from simuPOP.sampling import drawCombinedSample, AffectedSibpairSampler, NuclearFamilySampler
>>> pop = sim.loadPopulation('log/pedigree.pop')
>>> sample = drawCombinedSample(pop, samplers = [
...     AffectedSibpairSampler(families=1),
...     NuclearFamilySampler(families=1, numOffspring=(2,4), affectedParents=(1,2), affectedOffspring=(1,3))
...     ])
Warning: number of requested Pedigrees 1 is greater than what exists (0).
```

```
Warning: not enough non-overlapping Pedigrees are found (requested 1, found 0).
Warning: number of requested Pedigrees 1 is greater than what exists (0).
Warning: not enough non-overlapping Pedigrees are found (requested 1, found 0).
```

6.5.9 Sampling from subpopulations and virtual subpopulations *

Virtual subpopulations (VSPs) could be specified in the `subPops` parameter of sampling classes and functions. This can be used to limit your samples to individuals with certain properties. For example, you may want to match the age of cases and controls in a case-control association study by selecting your samples from a certain age group. For examples, Example 6.25 draws 500 cases and 500 controls from two a VSP with individual ages between 40 and 60.

Listing 6.25: Draw samples from a virtual subpopulation.

```
>>> import simuPOP as sim
>>> # create an age-structured population with a disease
>>> import random
>>> pop = sim.Population(10000, loci=10, infoFields='age')
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>> sim.initInfo(pop, lambda: random.randint(0, 70), infoFields='age')
>>> pop.setVirtualSplitter(sim.InfoSplitter(cutoff=(40, 60), field='age'))
>>> sim.maPenetrance(pop, loci=5, penetrance=(0.1, 0.2, 0.3))
>>> #
>>> from simuPOP.sampling import drawCaseControlSample
>>> sample = drawCaseControlSample(pop, cases=500, controls=500, subPops=[(0,1)])
>>> ageInSample = sample.indInfo('age')
>>> print(min(ageInSample), max(ageInSample))
(40.0, 59.0)
```

If a list of sample sizes is given, specified number of samples will be drawn from each subpopulation. For example, if you have an age-structured population when individuals with different ages have different risk to a disease, you might want to draw affected individuals from different age groups and perform association analyses. Function `drawCaseControlSample` cannot be used because both groups are affected, but you can `drawRandomSample` from two VSPs defined by age. Example 6.26 demonstrates how to use this method.

Listing 6.26: Sampling separately from different virtual subpopulations

```
>>> import simuPOP as sim
>>> # create an age-structured population with a disease
>>> import random
>>> pop = sim.Population(10000, loci=10, infoFields='age')
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>> sim.initInfo(pop, lambda: random.randint(0, 70), infoFields='age')
>>> pop.setVirtualSplitter(sim.InfoSplitter(cutoff=(20, 40), field='age'))
>>> # different age group has different penetrance
>>> sim.maPenetrance(pop, loci=5, penetrance=(0.1, 0.2, 0.3), subPops=[(0,1)])
>>> sim.maPenetrance(pop, loci=5, penetrance=(0.2, 0.4, 0.6), subPops=[(0,2)])
>>> # count the number of affected individuals in each group
>>> sim.stat(pop, numOfAffected=True, subPops=[(0,1), (0,2)], vars='numOfAffected_sp')
>>> print(pop.dvars((0,1)).numOfAffected, pop.dvars((0,2)).numOfAffected)
(681, 2126)
>>> #
>>> from simuPOP.sampling import drawRandomSample
>>> sample = drawRandomSample(pop, sizes=[500, 500], subPops=[(0,1), (0,2)])
>>> # virtual subpopulations are rearranged to different subpopulations.
>>> print(sample.subPopSizes())
```

```
(500L, 500L)
```

6.6 Module simuPOP.gsl

simuPOP makes use of many functions from the GUN Scientific Library. These functions are used to generate random number and perform statistical tests within simuPOP. Although these functions are not part of simuPOP, they can be useful to users of simuPOP from time to time and it makes sense to expose these functions directly to users.

Module `simuPOP.gsl` contains a number of GSL functions. Because only a small proportion of GSL functions are used in simuPOP, this module is by no means a comprehensive wrapper of GSL. Please refer to the simuPOP reference manual for a list of functions included in this module, and the GSL manual for more details. Because random number generation functions such as `gsl_ran_gamma` are already provided in the `simuPOP.RNG` class (e.g. `getRNG().randGamma`), they are not provided in this module.

6.7 Module simuPOP.sandbox

The simuPOP sandbox can be understood as the binary version of the simuPOP cookbook. It contains **experimental or specialized** classes and functions that might or might not be formally adopted by simuPOP. For example, a mating scheme in which parents locate their spouses randomly but with probabilities that are related to geographic distances will be slow to implement at the Python level. It might be provided here because it relies on many assumptions such as the existence of certain information fields, and is unlikely to be adopted by simuPOP.

Because of the experiemental nature of this module, **compatibility is not guaranteed for classes and functions provided in this module**. If you use an operator from this module, your script might be locked to a particular version of simuPOP that provides this operator. On the other hand, if you have implemented some C/C++ level classes or functions for your own simulation, it can be a good idea to add them to this module so that they can be distributed with simuPOP (at least in certain versions of simuPOP). Please refer to the simuPOP reference manual for a list of classes and functions provided in the current version of simuPOP.

Chapter 7

A real world example

Previous chapters use a lot of examples to demonstrate individual simuPOP features. However, it might not be clear how to integrate these features in longer scripts that address real world problems, which may involve larger populations, more complex genetic and demographic models and may run thousands of replicates with different parameters. This chapter will show you, step by step, how to write a complete simuPOP script that has been used in a real-world research topic.

7.1 Simulation scenario

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

This example is a simplified version of the `simuCDCV.py` script that simulates this evolution process and observe the allelic spectra of both types of diseases. The complete script is available at the simuPOP online cookbook. The results are published in Peng and Kimmel [2007], which has much more detailed discussion about the simulations, and the parameters used.

7.2 Demographic model

The original paper used a very simple instant population growth model. Under the model assumption, a population with an initial population size N_0 would evolve G_0 generations, instantly expand its population size to N_1 and evolve another G_1 generations. Such a model can be easily implemented as follows:

```
def ins_expansion(gen):
    'An instant population growth model'
    if gen < G0:
        return N0
    else:
        return N1
```

Other demographic models could be implemented similarly. For example, an exponential population growth model that expand the population size from N_0 to N_1 in G_1 generations could be defined as

```
def exp_expansion(gen):
```

```
    'An exponential population growth model'
    if gen < G0:
        return N0
    else:
        rate = (math.log(N1) - math.log(N0))/G1
        return int(N0 * math.exp((gen - G0) * rate))
```

That is to say, we first solve r from $N_1 = N_0 \exp(rG_1)$ and then calculate $N_t = N_0 \exp(rG)$ for a given generation.

There is a problem here: the above definitions treat `N0`, `G0`, `N1` and `G1` as global variables. This is OK for small scripts but is certainly not a good idea for larger scripts especially when different parameters will be used. A better way is to wrap these functions by another function that accept `N0`, `G0`, `N1` and `G1` as parameters. That is demonstrated in Example 7.1 where a function `demo_model` is defined to return either an instant or an exponential population growth demographic function.

Listing 7.1: A demographic function producer

```
>>> import simuPOP as sim
>>> import math
>>> def demo_model(model, N0=1000, N1=100000, G0=500, G1=500):
...     '''Return a demographic function
...     model: linear or exponential
...     N0:   Initial sim.population size.
...     N1:   Ending sim.population size.
...     G0:   Length of burn-in stage.
...     G1:   Length of sim.population expansion stage.
...     '''
...     def ins_expansion(gen):
...         if gen < G0:
...             return N0
...         else:
...             return N1
...     rate = (math.log(N1) - math.log(N0))/G1
...     def exp_expansion(gen):
...         if gen < G0:
...             return N0
...         else:
...             return int(N0 * math.exp((gen - G0) * rate))
...     if model == 'instant':
...         return ins_expansion
...     elif model == 'exponential':
...         return exp_expansion
...
>>> # when needed, create a demographic function as follows
>>> demo_func = demo_model('exponential', 1000, 100000, 500, 500)
>>> # sim.population size at generation 700
>>> print(demo_func(700))
6309
```

Note: The defined demographic functions return the total population size (a number) at each generation beacuse no subpopulation is considered. A list of subpopulation sizes should be returned if there are more than one subpopulations.

7.3 Mutation and selection models

The thoretical model empolyees an infinite allele model where there is a single wild type allele and an infinite number of disease alleles. Each mutation would introduce a new disease allele and there is no back mutation (mutation from disease allele to wild type allele).

This mutation model can be mimicked by a k-allele model with resaonably large k. We initialize all alleles to 0 which is the wild type (A) and all other alleles are considered as disease alleles (a). Because an allele in a $k-$allele mutation model can mutate to any other allele with equal probability, $P(A \rightarrow a) \gg P(a \rightarrow A)$ since there are many more disease alleles than the wild type allele. If we choose a smaller k (e.g. $k = 20$), recurrent and back mutations can on longer be ignored but it would be interesting to simulate such cases because they are more realistic than the infinite allele model in some cases.

A k-allele model can be simulated using the `KAlleleMutator` operator which accepts a mutation rate and a maximum allelic state as parameters.

```
KAlleleMutator(k=k, rates=mu)
```

Because there are many possible disease alleles, a multi-allelic selector (`MaSelector`) could be used to select against the disease alleles. This operator accept a single or a list of wild type alleles (`[0]` in this case) and treat all other alleles as disease alleles. A penetrance table is needed which specified the fitness of each individual when they have 0, 1 or 2 disease alleles respectively. In this example, we assume a recessive model in which only genotype aa causes genetic disadvantages. If we assume a selection pressure parameter s, the operator to use is

```
MaSelector(loci=0, wildtype=0, penetrance=[1, 1, 1-s])
```

Note that the use of this selector requires a population information field `fitness`.

This example uses a single-locus selection model but the complete script allows the use of different kinds of multi-locus selection model. If we assume a multiplicative multi-locus selection model where fitness values at different loci are combined (multiplied), a multi-locus selection model (`MlSelector`) could be used as follows:

```
MlSelector([
    MaSelector(loci=loc1, fitness=[1,1,1-s1], wildtype=0),
    MaSelector(loci=loc2, fitness=[1,1,1-s2], wildtype=0)],
    mode=MULTIPLICATIVE
)
```

These multi-locus model treat disease alleles at different loci more or less independently. If more complex multi-locus models (e.g. models involve gene - gene and/or gene - interaction) are involved, a multi-locus selector that uses a multi-locus penetrance table could be used.

7.4 Output statistics

We first want to output total disease allele frequency of each locus. This is easy because `Stat()` operator can calculate allele frequency for us. What we need to do is use a `Stat()` operator to calculate allele frequency and get the result from population variable `alleleFreq`. Because allele frequcies add up to one, we can get the total disease allele frequency using the allele frequency of the wild type allele 0 ($\sum_{i=1}^{\infty} f_i = 1 - f_0$). The actual code would look more or less like this:

```
Stat(alleleFreq=[0,1]),
PyEval(r'"%.2f" % (1-alleleFreq[0][0])')
```

We are also interested in the effective number of alleles [Reich and Lander, 2001] at a locus. Because simuPOP does not provide an operator or function to calculate this statistic, we will have to calculate it manually. Fortunately, this is

not difficult because effective number of alleles can be calculated from existing allele frequencies, using formula

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

where f_i is the allele frequency of disease allele i.

A quick-and-dirty way to output n_e at a locus (e.g. locus 0) can be:

```
PyEval('1./sum([(alleleFreq[0][x]/(1-alleleFreq[0][0]))**2 for x in alleleFreq[0].keys() if x != 0])')
```

but this expression looks complicated and does not handle the case when $f_0 = 1$. A more robust method would involve the `stmts` parameter of `PyEval`, which will be evaluated before parameter `expr`:

```
PyEval(stmts='''if alleleFreq[0][0] == 1:
    ne = 0
else:
    freq = [freq[0][x] for x in alleleFreq[0].keys() if x != 0]
    ne = 1./sum([(f/(1-alleleFreq[0][0])**2 for x in freq])
''', expr=r'"%.3f" % ne')
```

However, this piece of code does not look nice with the multi-line string, and the operator is not really reusable (only valid for locus o). It makes sense to define a function to calculate n_e generally:

```
def ne(pop, loci):
    ' calculate effective number of alleles at given loci'
    stat(pop, alleleFreq=loci)
    ne = {}
    for loc in loci:
        freq = [y for x,y in pop.dvars().alleleFreq[loc].iteritems() if x != 0]
        sumFreq = 1 - pop.dvars().alleleFreq[loc][0]
        if sumFreq == 0:
            ne[loc] = 0
        else:
            ne[loc] = 1. / sum([(x/sumFreq)**2 for x in freq])
    # save the result to the population.
    pop.dvars().ne = ne
    return True
```

When it is needed to calculate effective number of alleles, a Python operator that uses this function can be used. For example, operator

```
PyOperator(func=ne, param=[0], step=5)
PyEval(r'"%.3f" % ne[0]', step=5)
```

would calculate effective number of alleles at locus 0 and output it.

The biggest difference between `PyEval` and `PyOperator` is that `PyOperator` is no longer evaluated in the population's local namespace. You will have to get the variables explicitly using the `pop.dvars()` function, and the results have to be explicitly saved to the population's local namespace.

The final implementation, as a way to demonstrate how to define a new statistics that hides all the details, defines a new operator by inheriting a class from `PyOperator`. The resulting operator could be used as a regular operator (e.g., `ne(loci=[0])`). A function `Ne` is also defined as the function form of this operator. The code is listed in Example 7.2

Listing 7.2: A customized operator to calculate effective number of alleles

```
>>> import simuPOP as sim
>>> class ne(sim.PyOperator):
```

```
...     '''Define an operator that calculates effective number of
...     alleles at given loci. The result is saved in a population
...     variable ne.
...     '''
...     def __init__(self, loci, *args, **kwargs):
...         self.loci = loci
...         sim.PyOperator.__init__(self, func=self.calcNe, *args, **kwargs)
...     #
...     def calcNe(self, pop):
...         sim.stat(pop, alleleFreq=self.loci)
...         ne = {}
...         for loc in self.loci:
...             freq = pop.dvars().alleleFreq[loc]
...             sumFreq = 1 - pop.dvars().alleleFreq[loc][0]
...             if sumFreq == 0:
...                 ne[loc] = 0
...             else:
...                 ne[loc] = 1. / sum([(freq[x]/sumFreq)**2 for x in list(freq.keys()) if x != 0])
...         # save the result to the sim.Population.
...         pop.dvars().ne = ne
...         return True
...
>>> def Ne(pop, loci):
...     '''Function form of operator ne'''
...     ne(loci).apply(pop)
...     return pop.dvars().ne
...
>>> pop = sim.Population(100, loci=[10])
>>> sim.initGenotype(pop, freq=[.2] * 5)
>>> print(Ne(pop, loci=[2, 4]))
{2: 3.9565470135154768, 4: 3.948841408365935}
```

7.5 Initialize and evolve the population

With appropriate operators to perform mutation, selection and output statistics, it is relatively easy to write a simulator to perform a simulation. This simulator would create a population, initialize alleles with an initial allic spectrum, and then evolve it according to specified demographic model. During the evolution, mutation and selection will be applied, statistics will be calculated and outputed.

Listing 7.3: Evolve a population subject to mutation and selection

```
>>> import simuPOP as sim
>>>
>>>
>>> def simulate(model, N0, N1, G0, G1, spec, s, mu, k):
...     '''Evolve a sim.Population using given demographic model
...     and observe the evolution of its allelic spectrum.
...     model: type of demographic model.
...     N0, N1, G0, G1: parameters of demographic model.
...     spec: initial allelic spectrum, should be a list of allele
...         frequencies for each allele.
...     s: selection pressure.
...     mu: mutation rate.
```

```
...     k: k for the k-allele model
...     '''
...     demo_func = demo_model(model, N0, N1, G0, G1)
...     pop = sim.Population(size=demo_func(0), loci=1, infoFields='fitness')
...     pop.evolve(
...         initOps=[
...             sim.InitSex(),
...             sim.InitGenotype(freq=spec, loci=0)
...         ],
...         matingScheme=sim.RandomMating(subPopSize=demo_func),
...         postOps=[
...             sim.KAlleleMutator(k=k, rates=mu),
...             sim.MaSelector(loci=0, fitness=[1, 1, 1 - s], wildtype=0),
...             ne(loci=[0], step=100),
...             sim.PyEval(r'"%d: %.2f\t%.2f\n" % (gen, 1 - alleleFreq[0][0], ne[0])',
...                 step=100),
...         ],
...         gen = G0 + G1
...     )
...
>>> simulate('instant', 1000, 10000, 500, 500, [0.9]+[0.02]*5, 0.01, 1e-4, 200)
0: 0.09 4.91
100: 0.12       2.63
200: 0.09       1.22
300: 0.02       2.85
400: 0.02       2.12
500: 0.05       1.02
600: 0.06       1.51
700: 0.08       1.58
800: 0.09       1.80
900: 0.08       1.79
```

7.6 Option handling

Everything seems to be perfect until you need to

1. Run more simulations with different parameters such as initial population size and mutaion rate. This requires the script to get its parameters from command line (or a configuration file) and executes in batch mode, perhaps on a cluster system.
2. Allow users who are not familiar with the script to run it. This would better be achieved by a graphical user interface.
3. Allow other Python scripts to import your script and run the simulation function directly.

Although a number of Python modules such as `getopt` are available, the simuPOP `simuOpt` module is especially designed to allow a simuPOP script to be run both in batch and in GUI mode, in standard and optimized mode. Example 7.4 makes use of this module.

Listing 7.4: A complete simulation script

```
#!/usr/bin/env python
#
# Author:  Bo Peng
```

```
# Purpose: A real world example for simuPOP user's guide.
#
'''
Simulation the evolution of allelic spectra (number and frequencies
of alleles at a locus), under the influence of sim.population expansion,
mutation, and natural selection.
'''
import simuOpt
simuOpt.setOptions(quiet=True, alleleType='long')
import simuPOP as sim
import sys, types, os, math
options = [
    {'name': 'demo',
     'default': 'instant',
     'label': 'Population growth model',
     'description': 'How does a sim.Population grow from N0 to N1.',
     'type': ('chooseOneOf', ['instant', 'exponential']),
    },
    {'name': 'N0',
     'default': 10000,
     'label': 'Initial sim.population size',
     'type': 'integer',
     'description': '''Initial sim.population size. This size will be maintained
               till the end of burnin stage''',
     'validator': simuOpt.valueGT(0)
    },
    {'name': 'N1',
     'default': 100000,
     'label': 'Final sim.population size',
     'type': 'integer',
     'description': 'Ending sim.population size (after sim.population expansion)',
     'validator': simuOpt.valueGT(0)
    },
    {'name': 'G0',
     'default': 500,
     'label': 'Length of burn-in stage',
     'type': 'integer',
     'description': 'Number of generations of the burn in stage.',
     'validator': simuOpt.valueGT(0)
    },
    {'name': 'G1',
     'default': 1000,
     'label': 'Length of expansion stage',
     'type': 'integer',
     'description': 'Number of geneartions of the sim.population expansion stage',
     'validator': simuOpt.valueGT(0)
    },
    {'name': 'spec',
     'default': [0.9] + [0.02]*5,
     'label': 'Initial allelic spectrum',
     'type': 'numbers',
     'description': '''Initial allelic spectrum, should be a list of allele
            frequencies, for allele 0, 1, 2, ... respectively.''',
     'validator': simuOpt.valueListOf(simuOpt.valueBetween(0, 1)),
```

```
    },
    {'name': 's',
     'default': 0.01,
     'label': 'Selection pressure',
     'type': 'number',
     'description': '''Selection coefficient for homozygtes (aa) genotype.
            A recessive selection model is used so the fitness values of
            genotypes AA, Aa and aa are 1, 1 and 1-s respectively.''',
     'validator': simuOpt.valueGT(-1),
    },
    {'name': 'mu',
     'default': 1e-4,
     'label': 'Mutation rate',
     'type': 'number',
     'description': 'Mutation rate of a k-allele mutation model',
     'validator': simuOpt.valueBetween(0, 1),
    },
    {'name': 'k',
     'default': 200,
     'label': 'Maximum allelic state',
     'type': 'integer',
     'description': 'Maximum allelic state for a k-allele mutation model',
     'validator': simuOpt.valueGT(1),
    },
]

def demo_model(type, N0=1000, N1=100000, G0=500, G1=500):
    '''Return a demographic function
    type: linear or exponential
    N0:   Initial sim.population size.
    N1:   Ending sim.population size.
    G0:   Length of burn-in stage.
    G1:   Length of sim.population expansion stage.
    '''
    rate = (math.log(N1) - math.log(N0))/G1
    def ins_expansion(gen):
        if gen < G0:
            return N0
        else:
            return N1

    def exp_expansion(gen):
        if gen < G0:
            return N0
        else:
            return int(N0 * math.exp((gen - G0) * rate))

    if type == 'instant':
        return ins_expansion
    elif type == 'exponential':
        return exp_expansion

class ne(sim.PyOperator):
    '''Define an operator that calculates effective number of
```

```
    alleles at given loci. The result is saved in a population
    variable ne.
    '''
    def __init__(self, loci, *args, **kwargs):
        self.loci = loci
        sim.PyOperator.__init__(self, func=self.calcNe, *args, **kwargs)

    def calcNe(self, pop):
        sim.stat(pop, alleleFreq=self.loci)
        ne = {}
        for loc in self.loci:
            freq = pop.dvars().alleleFreq[loc]
            sumFreq = 1 - pop.dvars().alleleFreq[loc][0]
            if sumFreq == 0:
                ne[loc] = 0
            else:
                ne[loc] = 1. / sum([(freq[x]/sumFreq)**2 for x in list(freq.keys()) if x != 0])
        # save the result to the sim.Population.
        pop.dvars().ne = ne
        return True

def simuCDCV(model, N0, N1, G0, G1, spec, s, mu, k):
    '''Evolve a sim.Population using given demographic model
    and observe the evolution of its allelic spectrum.
    model: type of demographic model.
    N0, N1, G0, G1: parameters of demographic model.
    spec: initial allelic spectrum, should be a list of allele
        frequencies for each allele.
    s: selection pressure.
    mu: mutation rate.
    k: maximum allele for the k-allele model
    '''
    demo_func = demo_model(model, N0, N1, G0, G1)
    print(demo_func(0))
    pop = sim.Population(size=demo_func(0), loci=1, infoFields='fitness')
    pop.evolve(
        initOps=[
            sim.InitSex(),
            sim.InitGenotype(freq=spec, loci=0)
        ],
        matingScheme=sim.RandomMating(subPopSize=demo_func),
        postOps=[
            sim.KAlleleMutator(rates=mu, k=k),
            sim.MaSelector(loci=0, fitness=[1, 1, 1 - s], wildtype=0),
            ne(loci=(0,), step=100),
            sim.PyEval(r'"%d: %.2f\t%.2f\n" % (gen, 1 - alleleFreq[0][0], ne[0])',
                step=100),
        ],
        gen = G0 + G1
    )
    return pop

if __name__ == '__main__':
    # get parameters
```

```
par = simuOpt.Params(options, __doc__)
if not par.getParam():
    sys.exit(1)

if not sum(par.spec) == 1:
    print('Initial allelic spectrum should add up to 1.')
    sys.exit(1)
# save user input to a configuration file
par.saveConfig('simuCDCV.cfg')
#
simuCDCV(*par.asList())
```

Example 7.4 uses a programming style that is used by almost all simuPOP scripts. I highly recommend this style because it makes your script seld-documentary and work well under a variety of environments. A script written in this style follows the following order:

1. First comment block

   The first line of the script should always be

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

   This line tells a Unix shell which program should be used to process this script if the script to set to be executable. This line is ignored under windows. It is customary to put author and date information at the top of a script as Python comments.

2. Module doc string

   The first string in a script is the module docstring, which can be referred by variable `__doc__` in the script. It is a good idea to describe what this script does in detail here. As you will see later, this docstring will be used in the `simuOpt.getParam()` function and be outputed in the usage information of the script.

3. Loading simuPOP and other Python modules

   simuPOP and other modules are usually imported after module docstring. This is where you specify which simuPOP module to use. Although a number of parameters could be used, usually only `alleleType` is specified because other parameters such as `gui` and `optimized` should better be controlled from command line.

4. Parameter description list

   A list of parameter description dictionaries are given here. This list specifies what parameters will be used in this script and describes the type, default value, name of command line option, label of the parameter in the parameter input dialog in detail. Although some directionary items can be ignored, it is a good practice to give detailed information about each parameter here.

5. Helper functions and classes

   Helper functions and classes are given before the main simulation function.

6. Main simulation function

   The main simulation function preforms the main functionality of the whole script. It is written as a function so that it can be imported and executed by another script. The parameter processing part of the script would be ignored in this case.

7. Script execution part conditioned by `__name__ == '__main__'`

   The execution part of a script should always be inside of a `if __name__ == '__main__'` block so that the script will not be executed when it is imported by another script. The first few lines of this execution block are almost always

```
par = simuOpt.Params(options, __doc__)
if not par.getParam():
    sys.exit(1)
```

which creates a simuOpt object and tries to get parameters from command line option, a configuration file, a parameter input dialog or interactive user input, depending on how this script is executed. Optionally, you can use

```
par.saveConfig('file.cfg')
```

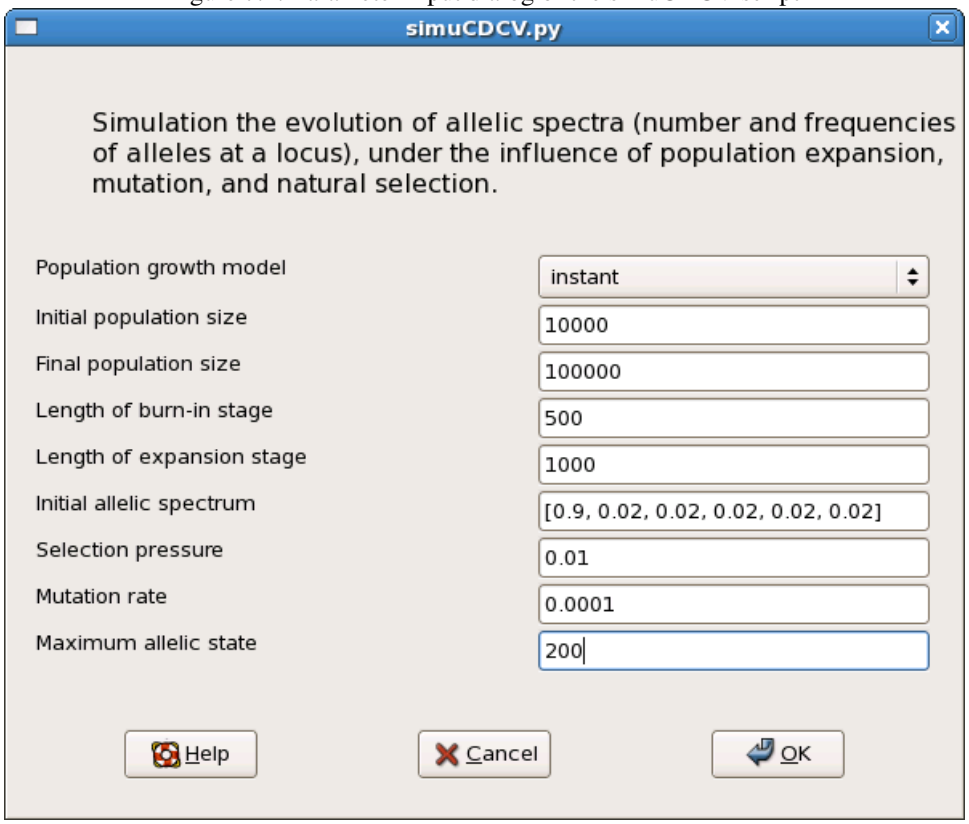
to save the current configuration to a file so that the same parameters could be retrieved later using parameter `--config file.cfg`.

After simply parameter validation, the main simulation function can be called. This example uses `simuCDCV(*par.asList())` because the parameter list in the `par` object match the parameter list of function `simuCDCV` exactly. If there are a large number of parameters, it may be better to pass the `simuOpt` object directly in the main simulation function.

The script written in this style could be executed in a number of ways.

1. If a user executes the script directly, a Tkinter or wxPython dialog will be displayed for users to input parameters. This parameter is shown in Figure 7.1.

Figure 7.1: Parameter input dialog of the simuCDCV script



2. The help message of this script could be displayed using the Help button of the parameter input dialog, or using command `simuCDCV.py -h`.

3. Using parameter `--gui=False`, the script will be run in batch mode. You can specify parameters using

```
simuCDCV.py --gui=False --config file.cfg
```

if a parameter file is available, or use command line options such as

```
simuCDCV.py --gui=False --demo='instant' --N0=10000 --N1=100000 \
    --G0=500 --G1=500 --spec='[0.9]+[0.02]*5' --s=0.01 \
    --mu='1e-4' --k=200
```

Note that parameters with `useDefault` set to `True` can be ignored if the default parameter is used. In addition, parameter `--optimized` could be used to load the optimized version of a simuPOP module. For this particular configuration, the optimized module is 30% faster (62s vs. 40s) than the standard module.

4. The simulation function could be imported to another script as follows

```
from simuCDCV import simuCDCV
simuCDCV(model='instant', N0=10000, N1=10000, G0=500, G1=500,
    spec=[0.9]+[0.02]*5, s=0.01, mu=1e-4, k=200)
```

Index

F
function
    loadPopulation, 41

G
GenoStruTrait
    chromName, 21
    chromType, 21
    infoField, 21
    infoFields, 21
    locusPos, 21
    numChrom, 21
    numLoci, 21
    ploidy, 21
    ploidyName, 21
genotypic structure, 21

I
index
    absolute, 10
    relative, 10

M
mating scheme, 149
moduleInfo, 18

O
operator
    Stat, 40

P
Population, 28
    Population, 40
    save, 41
    vars, 40

R
r, 14

S
setRNG, 18
SplitSubPops, 67