
CoaSim/Python Manual

Using the Python-based CoaSim Simulator

Thomas Mailund
mailund@birc.au.dk

Copyright © 2006 Thomas Mailund • Bioinformatics ApS

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved in all copies.

About This Manual

CoaSim is a tool for simulating the coalescent process with recombination and geneconversion, under either constant population size or exponential population growth. It effectively constructs the ancestral recombination graph for a given number of chromosomes and uses this to simulate samples of SNP and micro-satellite haplotypes or genotypes.

CoaSim comes in two flavours: A graphical user interface version for easy use by novice users, and a script based version—using either Guile-Scheme (<http://www.gnu.org/software/guile/>) or Python (<http://www.python.org>)—for efficient batch simulations. This document is an introduction to the Python based version and will describe how to use CoaSim for various simulation purposes. The manual describes a number of common and more exotic simulation setups and shows how CoaSim can be scripted to conduct such simulations. The manual has intentionally a bit of a tutorial flavour, as we wish to give a feeling of the kind of simulations CoaSim is suited for, and how such simulations are set up and executed in CoaSim, rather than giving a detailed description of all the functions available in CoaSim. For the later, we refer to the online documentation in Python; to get an overview of the module, use

```
>>> import CoaSim
>>> help(CoaSim)
```

or in general

```
>>> help(class-or-function)
```

Some familiarity with the Python programming language and with coalescent theory is assumed. For people not familiar with Python, an excellent tutorial can be found at <http://www.python.org/doc/2.4.2/tut/tut.html>, and for people not familiar with coalescent theory, the text book

Gene Genealogies, Variation and Evolution

A Primer in Coalescent Theory,

Jotun Hein, Mikkel H. Schierup, and Carsten Wiuf,

Oxford University Press, ISBN 0-19-852996-1

is recommended.

We also assume that you have successfully installed CoaSim, if not we refer to the *Getting Started with CoaSim/Python* manual (<http://www.daimi.au.dk/~mailund/CoaSim/download/getting-started-python-1.0.pdf>).

Running CoaSim

The Python based CoaSim tool is available as a Python extension module, that is loaded into Python using the import command:

```
>>> import CoaSim
```

After this, the basic functionality of CoaSim is available for the script. More features are available through additional modules introduced later. In the rest

of this manual we will assume that you know how to execute scripts using CoaSim in this way, and concentrate on how to write the appropriate scripts.

Simple Sequence Simulations

We start out with the most common usage of CoaSim: simulating sequence data. All simulations in CoaSim result in an *Ancestral Recombination Graph* (ARG), but in most cases the ARG is not the desired end-result, rather, the sequences found at the leaves of the ARG are. To extract these sequences from an ARG, we simply get the `sequences` attribute of the simulated ARG object.

Simulating and Saving Sequences

A simple sequence simulation script could look like this:

```
from CoaSim import simulate
from CoaSim.randomMarkers import makeRandomSNPMarkers

markers = makeRandomSNPMarkers(10, 0.0, 1.0)
sequences = simulate(markers, 10).sequences

print sequences
```

The first line

```
from CoaSim import simulate
```

loads the CoaSim module and imports the `simulate` function into the global namespace. The second line

```
from CoaSim.randomMarkers import makeRandomSNPMarkers
```

includes the `CoaSim.randomMarkers` module, which is used for generating random markers (markers on random positions, in this case). The random markers are then used in the next line of the script, where we make 10, randomly positioned SNP markers with allowed mutant-frequencies between 0.0 and 1.0, created using the `makeRandomSNPMarkers` function. These are then used in the next line, where we simulate 10 sequences, the second argument to `simulate`, and each sequence contains an allele for each marker in `markers`.

The call to the `simulate` function really returns an ARG object, but here we ignore that completely and immediately extracts the sequences from it; the line

```
sequences = simulate(markers, 10).sequences
```

corresponds to

```
arg = simulate(markers, 10)
sequences = arg.sequences
```

except that we never declare a named reference, `arg`, to the simulated ARG, but just extract the sequences.

The sequences are represented as a list of lists of alleles. This is a format that is very easy to manipulate by Python, but not as useful for most other programs, so printing the sequences in this format, as we do with the `print` call above, is not that useful. To print the sequences in a more traditional form of a sequence per line, with the sequences printed as space-separated numbers, we can use the (`CoaSim.IO`) module like this:

```
from CoaSim import simulate
from CoaSim.randomMarkers import makeRandomSNPMarkers

markers = makeRandomSNPMarkers(10, 0.0, 1.0)
sequences = simulate(markers, 100, rho=400).sequences

from CoaSim.IO import printMarkerPositions, printSequences
printMarkerPositions(markers, open('positions.txt', 'w'))
printSequences(sequences, open('sequences.txt', 'w'))
```

Here, the positions are written to the file `positions.txt` and the sequences to the file `sequences.txt`.

Simulation Parameters

As called above, `simulate` simulate a basic coalescent tree over the leaf nodes and then apply mutations on the markers. Recombinations, gene conversions and exponential growths can be included by setting the appropriate parameters using keyword arguments to `simulate`, e.g. to simulate with a scaled recombination rate $\rho = 4Nr = 400$ (see *Hein et al.* Sect. 5.5)—which for an effective population size of $N = 10,000$ means that the region from 0 to 1 simulated correspond roughly to 1cM—you would need to call

```
simulate(markers, noSequences, rho=400)
```

Similarly, to enable gene-conversions with rate $\gamma = 4Ng$ and intensity $Q = qL$ (see *Hein et al.* Sect. 5.10), use the keywords `gamma` and `Q` as in

```
simulate(markers, noSequences, gamma=250, Q=100)
```

These parameters can be combined, so e.g. simulating sequences with both gene conversions and recombination can be done using:

```
from CoaSim import simulate
from CoaSim.randomMarkers import makeRandomSNPMarkers

markers = makeRandomSNPMarkers(10, 0.0, 1.0)
sequences = simulate(markers, noSequences,
                    rho=400, gamma=250, Q=100).sequences
```

Markers

The first argument to the `simulate` function is a list of markers. All markers are positioned in the interval 0–1 and to simulate different genetic distances you will have to scale the simulation parameters above. The list of markers

passed to `simulate` must be sorted with relation to the position. This is guaranteed to be the case when the list is created using one of the random-marker functions from the `CoaSim.randomMarkers` module, as in the case of `makeRandomSNPMarkers` above, but can otherwise be ensured by sorting the markers with the `sortMarkers` function from the `CoaSim` module.

The markers determine how the sequences will be simulated, in the sense that they position the polymorphism along the genomic region being simulated—the relative distance between markers affects e.g. the probability of recombinations occurring between two markers—and determine the model of mutations for the polymorphic sites.

So far, we have only used randomly distributed SNP markers, but we need not only use SNP markers, nor need we stick to randomly positioned markers. `CoaSim` supports three types of markers, that differs in how mutations are placed on the ARG. The built-in¹ marker types are:

Trait markers are binary polymorphisms (think presence or absence of a trait) with a simple mutation model: after simulating the ARG, a mutation is placed uniformly at random on the tree local to the marker position, nodes below the mutation will have the mutant allele while all others will have the wild-type allele. A range of accepted mutant-frequencies can be specified and a simple rejection-sampling scheme is used to ensure it: if, after placing the mutation, the number of mutant leaves are not within the range, the ARG is rejected and the simulation restarted.

SNP markers resemble trait markers in that they are binary polymorphisms, and use the same mutation model as the trait-markers. They differ from the trait-markers in how the mutant-frequency is ensured: If, after the mutation has been placed, the number of mutant leaves does not fall within the accepted range, the mutation is re-placed, but the ARG is not rejected and re-simulated. This places a bias on the markers, but one that resembles the ascertainment bias seen in association studies, where SNPs are chosen to have frequencies in certain ranges.

Micro-satellite markers are k-allele polymorphisms with a different mutation model than the other two. For micro-satellite markers, each edge in the local tree at the marker is considered in turn, and based on a mutation rate and the length of the edge either a mutation is placed on the edge or it is not. If it is, a randomly chosen allele from 0 to $k - 1$ is placed on the child node; if no mutation is placed, the child node gets a copy of the allele at the parent node.

A trait marker can be created using the class constructor `TraitMarker`:

```
TraitMarker(position, lowMutationFreq, highMutationFreq)
```

so e.g.

```
TraitMarker(0.5, 0.18, 0.22)
```

¹Later in this manual we will see how to build our own custom marker types.

would place a trait marker at position 0.5 with accepted mutant-frequencies in the range 0.18–0.22. Quite similarly, SNP markers can be created with the class `SNPMarker`

```
SNPMarker(position, lowMutationFreq, highMutationFreq)
```

while micro-satellite markers can be created with the `MicroSatteliteMarker` class

```
MicroSatteliteMarker(position,  $\theta$ , k)
```

where θ is the scaled mutation rate $\theta = 4N\mu$ and k the number of alleles permitted at the marker (i.e. the alleles on that marker are numbered from 0 to $k - 1$ and each mutant allele is drawn uniformly from this set).

Using these three functions we can explicitly create a list of markers for a simulation:

```
from CoaSim import *
markers = [SNPMarker(0.1, 0.2, 0.8),
           SNPMarker(0.2, 0.1, 0.9),
           TraitMarker(0.5, 0.18, 0.22),
           SNPMarker(0.7, 0.2, 0.8),
           MicroSatteliteMarker(0.8, 1.5, 10)]
seqs = simulate(markers, rho=400).sequences
```

Here we create three SNP markers, on positions 0.1, 0.2 and 0.7, with mutation frequencies in the range 0.2–0.8 for position 0.1 and 0.7 and in the range 0.1–0.9 for position 0.2; a single trait marker at position 0.5, with mutation range 0.18–0.22, and a single micro-satellite marker at position 0.8, with mutation rate 1.5 and with a pool of 10 alleles.

The three marker classes are all from the module `CoaSim`, but rather than importing them into the global namespace (together with `simulate`), we just include the entire `CoaSim` namespace using the `from ... import *` command.

Explicitly creating the markers in this way can be cumbersome, so `CoaSim` provides functions for generating markers at random positions, as we have already seen, in the module `CoaSim.randomMarkers`. The three functions:

```
makeRandomTraitMarkers(n, lowFreq, highFreq)
makeRandomSNPMarkers (n, lowFreq, highFreq)
makeRandomMSMarkers (n,  $\theta$ , k)
```

creates n markers of the respective marker types, randomly positioned.

A list of markers created with these functions is guaranteed to be sorted, but if you choose to combine several lists you must make sure that the resulting list is sorted. This can be done either by calling `sortMarkers` from the module `CoaSim` on the combined list, or by combining the lists using the function `insertSorted`, also from `CoaSim`, e.g.

```
from CoaSim import *
from CoaSim.randomMarkers import *
```

```

SNPMarkers = makeRandomSNPMarkers(10, 0.1, 0.9)
traitMarkers = makeRandomTraitMarkers(2, 0.2, 0.4)

markers = sortMarkers(SNPMarker+traitMarkers)

```

creates a list by appending two marker-lists and then sort the markers using `sortMarkers`, and is equivalent to

```

from CoaSim import *
from CoaSim.randomMarkers import *

SNPMarkers = makeRandomSNPMarkers(10, 0.1, 0.9)
traitMarkers = makeRandomTraitMarkers(2, 0.2, 0.4)

markers, traitIndices = insertSorted(SNPMarkers, traitMarkers)

```

that merges the two sorted lists and returns the resulting list, together with the indices in that list containing the markers from the second list. Notice, however, that while `sortMarkers` will work on any list of markers, `insertSorted` expects the first list to be sorted.

Disease status: Affected and unaffected sequences

Enough about markers, we will now return to the simulation of sequences. A common setting is simulating a set of sequences and then split them into cases and controls based on a trait mutation. We have already seen how to create a trait marker, together with other markers, and simulate a set of sequences over these markers; we have not, however, considered how to split the sequences into cases and controls, based on the allele on the trait marker.

The simplest way to do this is using the disease models and the `split` function from module `CoaSim.diseaseModelling`,

```

from CoaSim.diseaseModelling import split

# ... simulate sequences and set up disease model ...

affectedSeqs, unaffectedSeqs = split(diseaseModel, sequences)

```

which splits the sequence based on a disease model and the alleles at an certain markers determined by the disease model.

Depending on the disease model—we describe this below—the function splits the input sequences into two: the sequences affected by the disease, and the sequences that are unaffected. By default, the markers affecting the disease, i.e. the markers used by the disease model to determine disease status, are removed. It is possible to disable the removal of the these alleles in this function by calling the function with the key-word argument `keepIndices` set to `True`. In most cases, however, you do not want to keep the trait marker after having determined the affected/unaffected phenotype from it, so the default is to remove it.

The simplest disease model depends only on a single marker, and assigns affected status to all mutants and unaffected status to all wild-type sequences. To create such a disease model, we can use the `singleMarkerDisease` function from `CoaSim.diseaseModelling`:

```
from CoaSim.diseaseModelling import singleMarkerDisease
dm = singleMarkerDisease(traitMarkerIndex)
```

where `traitMarkerIndex` specifies the index in the marker list, containing the marker that affects the disease status.

A complete example of splitting the sequences based on a simple single-marker disease model can look like this:

```
1 from CoaSim import *
2 from CoaSim.randomMarkers import *
3 from CoaSim.diseaseModelling import *
4
5 markers = makeRandomSNPMarkers(10, 0.0, 1.0)
6 traitMarker = TraitMarker(randomPosition(), 0.2, 0.4)
7 markers,idx = insertSorted(markers, traitMarker)
8
9 sequences = simulate(markers, 100, rho=400).sequences
10 dm = singleMarkerDisease(idx)
11 affected, unaffected = split(dm,sequences)
```

Here we simulate sequences for ten randomly placed SNP markers (line 5) and a single randomly placed trait marker (line 6, where the random position is obtained from the function `randomPosition` loaded from the module `CoaSim.randomMarkers`), remembering, when merging the SNP markers and the trait marker in line 7, to get the index of the trait marker. This index is given to `singleMarkerDisease` in line 10 making this marker the determining maker for disease status.

Splitting the sequences into cases and controls based solely on the allele at a trait marker is not always appropriate; when e.g. simulating a disease that is not completely penetrant, such as many common diseases, we might wish to only select mutants with a certain probability. Similarly, when a disease is affected by environmental factors as well as genetic factors, we might select wild-type sequences as cases with a certain probability. Both situations can be handled with `singleMarkerDisease` using the key-word parameters `mutantRisk` and `wildTypeRisk`, that sets the probability of a mutant or wild-type, respectively, is selected as affected. To select about 20% of mutants and only 1% of wild-types, for example, you would use:

```
dm = singleMarkerDisease(idx, mutantRisk=0.2,wildTypeRisk=0.01)
affected,unaffected = split(dm,sequences)
```

Genotype Sequences

`CoaSim` simulates haplotype sequences, but we can combine these to genotype data by pairing sequences. This pairing is done, simply, by taking a even-length

list and combining the first two haplotype sequences into the first genotype sequence, the next two haplotype sequences into the second genotype sequences, and so forth.

The `singleMarkerDisease` model can also handle this form of genotype data, by specifying this using the optional parameter `model`:

```
singleMarkerDisease(idx, model=DiseaseModel.GENOTYPE_MODEL)
```

By default, `singleMarkerDisease` selects homozygote mutant sequences as cases, homozygote wild-type sequences as controls, and heterozygote as cases with probability 0.5 and controls with probability 0.5. This default can be changed using keyword arguments `homozygoteWildTypeRisk`, setting the probability of a homozygote 00 being affected, `homozygoteMutantRisk`, setting the probability of a homozygote 11² being affected, and `heterozygoteRisk`, setting the probability of a heterozygote 01 or 10 being affected. E.g. to select as cases homozygote mutants with probability 0.5, heterozygotes with probability 0.1 and homozygote wild-types with probability 0.01, we would use:

```
singleMarkerDisease(idx, model=DiseaseModel.GENOTYPE_MODEL,
                    homozygoteMutantRisk=0.5,
                    heterozygoteRisk=0.1,
                    homozygoteWildTypeRisk=0.01)
```

A disease model created this way is used in the `split` function just as a disease model for haplotype data, except that disease status, as mentioned above, depends on pairs of sequences rather than the individual sequences.

As a short-cut for common probabilities, the `CoaSim.diseaseModelling` module provides the functions `dominantModel` and `recessiveModel`. Using `dominantModel`:

```
affected, unaffected = split(dominantModel(idx), sequences)
```

will set the probability for both homozygote mutants and heterozygotes being affected to 1.0 and the probability for homozygote wild-types being affected to 0.0, while using the disease model `recessiveModel`

```
affected, unaffected = split(recessiveModel(idx), sequences)
```

will set the probability of homozygote mutants being affected to 1.0 and the probability of the other two genotypes being affected to 0.0.

Complex Disease Models

Splitting the simulated sequences into affected and unaffected sequences, on a single marker, as we have seen now for both haplotype and genotype data, is the common case for simulating data for disease mapping applications, for candidate-gene approaches, at least. More complex disease models require a bit

²In general, any non-0 is considered a mutant by `singleMarkerDisease`, so 11 is not the only encoding of a homozygote mutant. Here we just use this representation for simpler exposition.

more programming on the user's side, but the `CoaSim.diseaseModelling` module and the `split` function does support such setups.

To have the disease status determined by multiple markers, it is necessary to program a custom disease model. There are two ways of doing this: using the generic `DiseaseModel` class and a predicate function, or sub-classing `DiseaseModel`.

The `DiseaseModel` class is the basic disease modelling class, underlying the `singleMarkerDisease` model; its constructor takes either a single marker or a list of markers, for determining the disease model, and a `model` argument—similar to `singleMarkerDisease`—that defaults the class to work on haplotype data, but can be changed to make a genotype disease model.

Using `DiseaseModel` together with a predicate function is done through a second, optional, parameter, `predicate`. This argument should be a function that will be called with the alleles at the specified indices and should return `True` if the corresponding sequence (or sequence pair, in the case of genotype data) should be considered affected, and `False` otherwise. A simple example could look like this:

```
from CoaSim.diseaseModelling import split, DiseaseModel
def pred(a0,a2): return (a0,a2) == (1,1)
aff, unaf = split(DiseaseModel([0,2],predicate=pred), seqs)
```

Here we define a function `pred` that is called with two alleles and return `True` if they are both 1, and `False` otherwise. This is used in the disease model, where the two markers 0 (the first marker in the sequences) and 2 (the third) are used.

When working on genotype data, the setup is similar, but the predicate is called with pairs of alleles:

```
1 from CoaSim.diseaseModelling import split, DiseaseModel
2 def pred(p0,p2):
3     a00, a02 = p0
4     a20, a22 = p2
5     return a00!=a20 and a02==a22
6 dm = DiseaseModel([0,2],predicate=pred,
7                   model=DiseaseModel.GENOTYPE_MODEL)
8 af, unaf = split(dm, seqs)
```

Here, `pred` is called with two pairs, `p0` and `p2`, that are unpacked (lines 3 and 4), and used for determining disease status.

In both examples above, the predicate has been deterministic, but using the module `random` from the standard Python library, writing randomized versions is quite straightforward. For example, to modify the first version of the predicate, so it only considers two 1s affected with probability 0.5, we can use:

```
def pred(a0,a2):
    from random import uniform
    if (a0,a2) == (1,1): return uniform(0,1) < 0.5
    else: return False
```

Sub-classing `DiseaseModel`, the other approach to specifying custom disease models, works very similar to using a predicate; here the predicate is simply replaced by the `__call__` special method. The two examples above, specified this way, would look like:

```
from CoaSim.diseaseModelling import split, DiseaseModel
class DM(DiseaseModel):
    def __call__(self, a0, a2):
        return (a0,a2) == (1,1)
af, unaf = split(DM([0,2]), seqs)
```

and:

```
from CoaSim.diseaseModelling import split, DiseaseModel
class DM(DiseaseModel):
    def __init__(self, indices):
        DiseaseModel.__init__(self,indices,
                               model=DiseaseModel.GENOTYPE_MODEL)
    def __call__(self, p0, p2):
        a00, a02 = p0
        a20, a22 = p2
        return a00!=a20 and a02==a22
af, unaf = split(DM([0,2]), seqs)
```

where the later also sets the model to genotypes in the sub-class's constructor, the appropriate place to do this, since the `__call__` method expects genotype input.

Population Structure

In the simulations we have seen so far, we have simulated samples from a single, constant size, population, but CoaSim contains a powerful specification language for demographic structures. Instead of using the form for simulating sequences we have used so far

```
simulate(markers, noSequences).sequences
```

we use the more general form

```
simulate(markers, popStructSpec).sequences
```

where `popStructSpec` (population structure specification) is a small program describing the population structure of the sample.

Epochs: Bottlenecks and Exponential Growth

Simple extensions to the fixed size population model are bottlenecks and periods of exponential growth. A bottleneck is parameterised with the scale factor of the effective population size: $f = N_b/N$, where N is the effective population size outside the bottleneck and N_b the effective population size inside the bottleneck. To specify a bottleneck of $f = 0.2$, starting at time 1.5 (as measured in $2N$), see Fig. 1(a), we use



(a) A bottleneck of size 0.2 starting at time $\tau = 1.5$ and extending back in time.

(b) A bottleneck of size 0.2 starting at time $\tau = 1.5$ and ending at time $\tau = 3$.

Figure 1: A bottleneck of size 0.2 starting at time $\tau = 1.5$ and extending back in time indefinitely (left) or until time $\tau = 3$ (right).

```
from CoaSim.popStructure import Population, Bottleneck, Sample
popStructSpec = Population(1, Sample(10),
                           epochs=[Bottleneck(0.2, 1.5)])
seqs = simulate(markers, popStructSpec).sequences
```

The setup is a tad more complicated than the simple number of samples in the simulation, but we will walk through it: Instead of a simple number we have a program, `popStructSpec`.

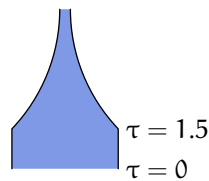
The specification starts with a population `Population(...)`—it always does—and the next parameter specifies the size of the population relative to N : it is a parameter f' , similar to the bottleneck parameter. In this particular case it is not really useful—it is 1 meaning the population size is the same as the global population size—but when we add population sub-structure later on, it will be.

The next parameter to `Population` is another population structure—the population structure specification in this way has a tree form, allowing us to specify trees of population relationships as we will see below—in this case a `Sample`, a leaf in the population specification tree, and representing simply a number of samples from this population; in this case 10 sequences.

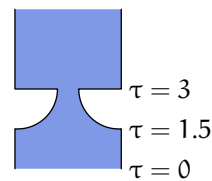
The `epochs` parameter `Population(1, ..., epochs=[...])` specifies the periods of different conditions for the population, such as bottlenecks. The `epochs` parameter takes a list of such periods as argument; in this case we have a single epoch, a bottleneck `Bottleneck(0.2, 1.5)`. This specification says that the size of the bottleneck is 0.2 ($f = 0.2$) and that the bottleneck starts at time 1.5; there is no end time for this particular bottleneck, so in effect it is just a reduction of the effective population size to 20% at time 1.5. To specify a bottleneck of a finite extend, we simply add a third, parameter, e.g. to have the bottleneck starting at time 1.5 and ending at time 3 we use:

```
from CoaSim.popStructure import Population, Bottleneck, Sample
popStructSpec = Population(1, Sample(10),
                           epochs=[Bottleneck(0.2, 1.5, 3)])
seqs = simulate(markers, popStructSpec).sequences
```

see Fig. 1(b).



(a) Exponential growth starting at time $\tau = 1.5$ and extending back in time.



(b) Exponential growth starting at time $\tau = 1.5$ and ending at time $\tau = 3$.

Figure 2: Exponential growth starting at time $\tau = 1.5$ and extending back in time indefinitely (left) or until time $\tau = 3$ (right).

Exponential growth is specified in a similar way, and parameterised by the exponential growth factor, $\beta = 2Nb$, (see *Hein et al.* Sect. 4.3). To have an exponential growth with $\beta = 10$ starting at time 1.5, we use:

```
from CoaSim.popStructure import Population, Growth, Sample
popStructSpec = Population(1, Sample(10),
                           epochs=[Growth(10, 1.5)])
seqs = simulate(markers, popStructSpec).sequences
```

see Fig. 2(a). Once again, an end time can be specified as a third parameter:

```
from CoaSim.popStructure import Population, Growth, Sample
popStructSpec = Population(1, Sample(10),
                           epochs=[Growth(10, 1.5, 3)])
seqs = simulate(markers, popStructSpec).sequences
```

see Fig. 2(b). After a finite period of growth, the population size returns to the size outside the epoch, i.e. the original size before the growth, unless the growth is followed by a bottleneck to reduce the population size.

As briefly mentioned, a population can contain a list of epochs, and we can use this to, e.g. specify an exponential growth followed (back in time) by a bottleneck, effectively modeling a smaller effective population size before (forward in time) the growth:

```
from CoaSim.popStructure import Population, Sample, \
    Bottleneck, Growth
popStructSpec = Population(1, Sample(10),
                           epochs=[Growth(10, 0, 3),
                                   Bottleneck(0.2, 3)])
seqs = simulate(markers, popStructSpec).sequences
```

or somewhat more succinct:

```
from CoaSim.popStructure import Population as P, Sample as S, \
    Bottleneck as B, Growth as G
popStructSpec = P(1, S(10), epochs=[G(10, 0, 3), B(0.2, 3)])
seqs = simulate(markers, popStructSpec).sequences
```

compressing the population structure specification, using shorter aliases for the population structure classes.

Epochs must be either nested or non-overlapping; nested epochs combine in the obvious way: a bottleneck that scales the effective population size with f_1 , nested inside a bottleneck that scales the effective population size with f_2 will in effect scale the effective population size with $f_1 \cdot f_2$; a bottleneck inside an exponential growth epoch, or vice versa, will model growth in a reduced size, and so on.

Sub-populations

The population structure in the previous section only contained a single population, going through various kinds of epochs, but the population structure specification language allows us to specify trees of populations, and thus specify the history of a set of populations back to their common ancestor population.

The branches on the population tree—describing the period a population exist and the epochs it goes through—are specified with the `Population()` structure described above. The leaves of the tree—describing how many samples to simulate from a currently (at time 0) existing population—are specified with the `Sample()` construction, also described above. The nodes—describing population merging events (backward in time, corresponding to splits of populations forward in time)—are specified with a similar `merge()` construction.

The merge construction takes, as first argument, the time of the merge, backward in time in units of $2N$, followed by a list of two or more population structure specifications. The merge must, of course, occur further back in time than any epochs or merges in the sub-trees specified by the population structures.

Thus, to simulate 20 samples from each of two populations merging at time 10, see Fig. 3, we use:

```
from CoaSim.popStructure import Population as P, Sample as S, \
    Merge as M
popStructSpec = P(1, M(10, [P(1, S(20)), P(1, S(20))]))
seqs = simulate(markers, popStructSpec).sequences
```

The outermost `Population(...)` specifies the population size after (back in time) the merge and the two inner `Population(...)`s the effective population size and the sample size in the sub-populations. For the simulated sequences, the first “sample size” (in this case 20) individuals will be from the first population in the population structure specification, the second “sample size” from the next population and so forth.

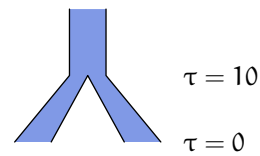


Figure 3: Two populations merging at time $\tau = 10$.

We now see the use of the population size parameter, the first parameter to the `Population(...)` construction: to specify different effective sizes of the populations. CoaSim assumes a fixed global effective size, $2N$ and scales all populations relative to this, using the population size parameter; this works in

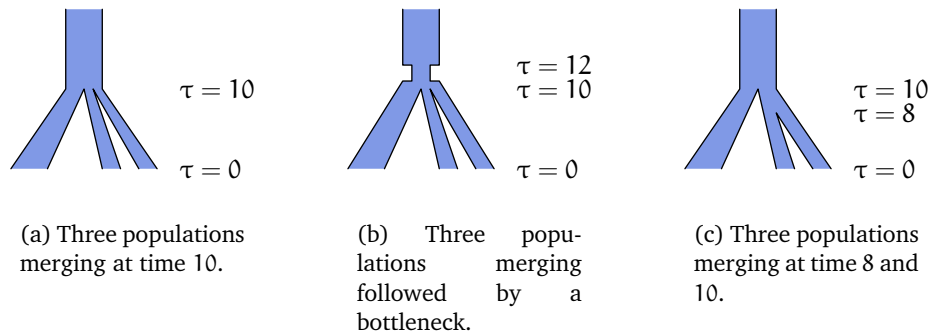


Figure 4: Three different population structures.

exactly the same way as time is scaled for population bottlenecks (and is in fact just syntactic sugar for bottlenecks covering the entire population life time).

As an example we can simulate three subpopulations, one twice the effective size of the other two, merging at time 10 back in time to a population with the same effective size as the larger sub-population at the current time:

```
from CoaSim.popStructure import Population as P, Sample as S,\
    Merge as M
popStructSpec = P(2,M(10,[P(2,S(20)),P(1,S(20)),P(1,S(20))]))
seqs = simulate(markers, popStructSpec).sequences
```

see Fig. 4(a).

The subpopulation size combine with epochs in the expected way, considering that they behave similar to bottlenecks, e.g. a bottleneck with size parameter f_b inside a population with size parameter f_p will result in an effective population size of $f_b \cdot f_p \cdot 2N$. Thus, in

```
from CoaSim.popStructure import Population as P, Sample as S,\
    Merge as M, Bottleneck as B
popStructSpec = P(2,M(10,[P(2,S(20)),P(1,S(20)),P(1,S(20))]),
    epochs=[B(0.25,10,12)])
seqs = simulate(markers, popStructSpec).sequences
```

the population merge is followed, back in time, by a bottleneck from time 10 to time 12 that scales the effective size to half that of the smaller current subpopulations: $0.25 \cdot 2 = 0.5$, the 0.25 factor from the bottleneck and the 2 factor from the population size, see Fig. 4(b).

A population structure tree does not have to be in only two levels, as above, where a merge is followed by an edge to a leaf; each sub-tree can be of the general form containing further merges, as for example

```
from CoaSim.popStructure import Population as P, Sample as S,\
    Merge as M
popStructSpec = P(2,M(10,[P(2,S(20)),
    M(8,[P(1,S(20)),P(1,S(20))])]))
seqs = simulate(markers, popStructSpec).sequences
```

where the original population splits in two (forward in time) 10 time units ago, one half the size of the other, and where the smaller population splits again 8 time units ago, see Fig. 4(c).

The merge times closer to the root must, of course, be larger than the merge times further down the tree, since events closer to the root represent events further back in time. For similar reasons, all epoch times on an edge must be contained in the time interval between the merge events at the end points of the edge; time 0 for edges to leaves; and open ended up for the root-edge.

Migration

It is also possible to simulate migrations between subpopulations existing at the same time. This is done by specifying the rates of migration from one population to another—optionally together with the period for which the migration goes, if this period does not span the entire period for which both populations exist.

To be able to refer to the individual subpopulations when specifying migration rates, we must name them. This is done by adding a name (a string) through the optional name parameter of the `Population(...)` construction.

We can name the subpopulations in the example in the previous section, with two small populations and one large, like this:

```
from CoaSim.popStructure import Population as P, Sample as S, \
    Merge as M, Bottleneck as B
popStructSpec = P(2,M(10,[P(2,S(20), name="large"),
    P(1,S(20), name="small-1"),
    P(1,S(20), name="small-2")]))
seqs = simulate(markers, popStructSpec).sequences
```

giving the larger population the name `large`, and the two smaller populations the names `small-1` and `small-2`, respectively. The population after the merge is not named, and thus cannot be referred to in the migration specification—but since it is the only population in existing after the merge, this would not be meaningful anyway.

With the relevant populations named, we can specify a migration rate using the `Migration` construction which is on the form:

```
Migration(source, destination, rate, startTime, endTime)
```

where `startTime` and `endTime` are optional keyword arguments and will default to the total time where both populations exist.

The `destination` parameter is the population that receives the immigrants, going back in time, and the `source` parameter is the population from which they emigrate, again going back in time.

The `rate` parameter is the migration rate $M = 4Nm$ (see *Hein et al.* Sect. 4.6). Notice that the rate is given in units of the *global* effective population size, the global N , not relative to the individual population sizes. It is, thus, not influenced by bottlenecks or relative subpopulation sizes—to achieve such effects it is necessary to specify different periods of migration with varying rates.

The migration rates are provided to the simulation function as a list through the keyword argument `migrationSpec`. To simulate a migration rate of $M = 0.5$ from each of the two small populations to the larger (back in time, and thus from the larger population to the smaller forward in time), a rate of 1.0 in the other direction, and a rate of 2.0 in each direction between the two smaller populations, we use:

```
from CoaSim.popStructure import Population as P, Sample as S,\
    Merge as M, Bottleneck as B,\
    Migration as Mi

pSpec = P(2,M(10,[P(2,S(20), name="large"),
                 P(1,S(20), name="small-1"),
                 P(1,S(20), name="small-2")]))
mSpec = [Mi('small-1', 'large', 0.5),
         Mi('small-2', 'large', 0.5),
         Mi('large', 'small-1', 1.0),
         Mi('large', 'small-2', 1.0),
         Mi('small-2', 'small-1', 2.0),
         Mi('small-1', 'small-2', 2.0)]

seqs = simulate(markers, pSpec, migrationSpec=mSpec).sequences
```

To specify that the migration rate between the two smaller populations is only 2.0 for the last half (back in time) of the population existence, and 1.5 before that (back in time), we can use the optional start and end time parameters:

```
from CoaSim.popStructure import Population as P, Sample as S,\
    Merge as M, Bottleneck as B,\
    Migration as Mi

pSpec = P(2,M(10,[P(2,S(20), name="large"),
                 P(1,S(20), name="small-1"),
                 P(1,S(20), name="small-2")]))
mSpec = [Mi('small-1', 'large', 0.5),
         Mi('small-2', 'large', 0.5),
         Mi('large', 'small-1', 1.0),
         Mi('large', 'small-2', 1.0),
         Mi('small-2', 'small-1', 2.0, 5, 10),
         Mi('small-1', 'small-2', 2.0, 5, 10),
         Mi('small-2', 'small-1', 1.5, 0, 5),
         Mi('small-1', 'small-2', 1.5, 0, 5)]

seqs = simulate(markers, pSpec, migrationSpec=mSpec).sequences
```

Simulating the ARG and (Local) Coalescent Trees

Until now we have only considered simulating sequences, but CoaSim actually simulates a full ARG in each simulation, we have just so far ignored the ARG and immediately extracted the sequences, using

```
sequences = simulate(markers, noSeqs).sequences
```

rather than

```
arg = simulate(markers, noLeaves)
sequences = arg.sequences
```

There is one slight complication though: since simulating sequences is by far the most common use of CoaSim, it has been optimised for this. This optimisation includes discarding the history of ancestral material intervals not containing any of the markers specified in the simulation parameters. This reduces the memory usage considerably, and speeds up the simulations a bit, but also means that the complete ARG is not available after the simulation is completed. In many cases this is not a problem, but if the full ARG is needed after the simulation the optimisation can be turned off by using the keyword argument `keepEmptyIntervals` set to `True`:

```
arg = simulate(markers, noLeaves, keepEmptyIntervals=True)
```

This is usually recommended when you wish to explore the genealogy between the specified markers—or when you are not interested in markers and only wish to simulate ARGs—but is not needed for e.g. simulating the local coalescent trees at the specified markers.

As a simple example, we can try simulating an ARG and then compute some statistics about it:

```
from CoaSim import *

arg = simulate([], 5, rho=20, gamma=10, Q=2,
              keepEmptyIntervals=True) # remember this!

counter = dict()
for node in arg.nodes:
    try:
        counter[type(node)] += 1
    except KeyError:
        counter[type(node)] = 1

print 'coalescent events:', counter[CoalescentNode]
print 'recombination events:', counter[RecombinationNode]/2
print 'gene-conversion events:', counter[GeneConversionNode]/2
```

Here we simulate an ARG and print the number of recombination events, gene conversions and coalescence events.

Notice that we do not provide any markers to `simulate`—the marker list is the empty list `[]`—but we instead turn off the empty intervals optimisation using `keepEmptyIntervals=True`. We then iterate through the nodes of the ARG—accessed through its `nodes` attribute—and count the different types of nodes (see below). For calculating the the number of recombination and gene-conversion events, we divide the number of corresponding nodes with two, since each recombination and gene-conversion event produces two nodes; for the recombination these correspond to the ancestral material to the left and to

the right of the recombination point, and for gene-conversions they correspond to the gene-conversion tract and the remaining ancestral material.

As another example, we can try to extract the local intervals of the ARG, that is, the intervals of the ARG not broken up by a recombination event. To get the list of these intervals, we access the ARG's `intervals` attribute, similar to how we get the nodes and sequences:

```
from CoaSim import *
arg = simulate([], 5, rho=20, gamma=10, Q=2,
               keepEmptyIntervals=True) # remember this!
intervals = arg.intervals
```

Again, we need to turn off the empty intervals optimisation, since otherwise we would only get the intervals containing markers; in this case no interval will contain a marker since we have provided no markers to the `simulate` function.

From the list of intervals, we could, for instance, calculate the average interval length. To do this we use functions `interval-start` and `interval-end` to get the start point and end point of the interval, respectively, and then obtain the length by subtracting the start point from the end point:

```
lengths = [i.end - i.start for i in intervals]
print 'average interval length:', sum(lengths) / len(lengths)
```

Simulating Trees

To each interval there correspond a local tree-genealogy. This tree can be obtained from the interval through the interval's `tree` attribute:

```
trees = [i.tree for i in intervals]
```

and likewise the interval for a tree can be reached from the tree's `interval` attribute:

```
intervals = [t.interval for t in trees]
```

To simulate a list of local genealogies across the simulated genomic region, and printing intervals and trees, we could then use:

```
from CoaSim import *
arg = simulate([], 5, rho=1, keepEmptyIntervals=True)
intervals = arg.intervals
for i in intervals:
    print i, ':', i.tree
```

Printing a tree this way will print out the tree in Newick format. To simulate a single coalescence tree, we can use code like this:

```
arg = simulate([], 5, keepEmptyIntervals=True)
tree = arg.intervals[0].tree
print tree
```

We simulate the ARG and then extract the list of intervals. This will only contain a single interval, since we are simulating without recombination—we have not set the `rho` parameter, which means that it defaults to 0—and we take the corresponding single tree out of the list and prints it.

Through the attribute `height` of the tree we can get its height, and for example compute the average height along the genomic sequence:

```
from CoaSim import *
arg = simulate([], 5, rho=20, keepEmptyIntervals=True)
trees = [i.tree for i in arg.intervals]
heights = [t.height for t in trees]
print 'average tree height:', sum(heights) / len(heights)
```

or the mean height over a number of simulations:

```
from CoaSim import *
noIterations = 1000
heights = []
for i in xrange(noIterations):
    arg = simulate([], 5, keepEmptyIntervals=True)
    tree = arg.intervals[0].tree
    heights.append(tree.height)
print 'Average tree height:', sum(heights) / len(heights)
```

Using the attribute `branchLength`, we can instead get the sum of branch lengths of the tree, and we can calculate the mean of those just as for tree heights as above.

Exploring the ARG

It is possible to explicitly traversing the and extracting the information through custom-made functions. Since we rarely have to explicitly traverse the ARG—most interesting simulation results can be extracted using other methods—we will not go into details about this general traversal, but just give short pointers to the relevant functions.

We have already seen how to iteration through all nodes in the ARG, using the `nodes` attribute. This gives access to the nodes in an arbitrary order; see below for how to access local graph structure from the nodes. The nodes can be of four different types: `LeafNode`, `CoalescentNode`, `RecombinationNode`, and `GeneConversionNode`, and we have already seen how this information can be used above.

In addition to checking the type of a node, we can get the time of the event producing the node through the `eventTime` attribute:

```
1 from CoaSim import *
2
3 arg = simulate([], 5, keepEmptyIntervals=True)
4 coalescenceTimes = []
5 for n in arg.nodes:
6     if isinstance(n, CoalescentNode):
7         coalescenceTimes.append(n.eventTime)
```

```

8
9 coalescenceTimes.sort()
10 for ct in coalescenceTimes:
11     print ct

```

Here, we collect the coalescence times by iterating through the nodes (lines 5–7), considering all coalescent nodes (line 6), and collecting their event times (line 7).

For all node types, two predicates, `isAncestral` and `isTrapped` can be used to check if a given position is ancestral, or (non-ancestral and) trapped between ancestral material, respectively. For recombination nodes, the attribute `recombinationPoint` gives the recombination point, and for gene-conversion nodes, the attribute `conversionInterval` gives the start and end-point of the gene-conversion.

For a more structured exploration of the ARG, the attribute `children` can be used to get the children of a node, between 0 (for leaf nodes) and 2 (for coalescent nodes). For a tree, the attribute `root` gives root node of a tree, or the *Most Recent Common Ancestor* (MRCA) of the tree.

Advanced Usage

In this section we give a few examples of more advanced usage of CoaSim. This should by no mean be considered an exhaustive list of the possibilities for CoaSim simulations, but only give a taste of the possibilities that are available when programming scripts for the simulator. Through the functions described in the reference manual, CoaSim is highly programmable, and very flexible in the kinds of simulations it can run.

Simulation Callbacks (and their use in e.g. Rejection Sampling)

There are seven hooks for callbacks in the simulations:

- `coalescenceEvent`—called with the single node that is the result of a coalescent event, and the number of lineages at the time of the coalescent (i.e. the number of linages just after the event, moving forward in time).
- `recombinationEvent`—called with the two nodes that is the result of a recombination event, and the number of lineages at the time of the recombination (i.e. the number of linages just after the event, moving forward in time).
- `geneConversionEvent`—called with the two nodes that is the result of a gene conversion event, and the number of lineages at the time of the gene conversion (i.e. the number of linages just after the event, moving forward in time).
- `bottleneckEvent`—called with the number of the affected population, a boolean flag indicating whether the event indicates entering or leaving

the bottleneck, the time of the event and the (total) number of lineages left at the time of the event.

- `growthEvent`—called with the number of the affected population, a boolean flag indicating whether the event indicates entering or leaving the growth period, the time of the event and the (total) number of lineages left at the time of the event.
- `migrationEvent`—called with the two populations (their number in the specification tree), the time of the migration, and the number of lineages at the time of the gene conversion (i.e. the number of lineages just after the event, moving forward in time).
- `mergeEvent`—called with a list of the populations being merged, the time of the merge, and the (total) number of lineages left at the time.

These are called during the ARG simulation, whenever one of the respective events happens. They can be used to gather information from the simulation—although much of the same information can be obtained by traversing the ARG after the simulation—but also used for aborting a running simulation in a rejection sampling setup.

To get a callback, we pass an object implementing a method with the corresponding name to the `callbacks` parameter of `simulate`. For example, to count the different types of events, we can use:

```
1 class callbacks(object):
2     def __init__(self):
3         self.counts = {'coa':0, 'rec':0, 'gc':0,
4                       'bn_enter':0, 'bn_leave':0,
5                       'g_enter':0, 'g_leave':0,
6                       'mig':0, 'me': 0}
7     def coalescentEvent(self, n, k):
8         self.counts['coa'] += 1
9     def recombinationEvent(self, n1, n2, k):
10        self.counts['rec'] += 1
11    def geneConversionEvent(self, n1, n2, k):
12        self.counts['gc'] += 1
13    def bottleneckEvent(self, pop, entering, t, k):
14        if entering:
15            self.counts['bn_enter'] += 1
16        else:
17            self.counts['bn_leave'] += 1
18    def growthEvent(self, pop, entering, t, k):
19        if entering:
20            self.counts['g_enter'] += 1
21        else:
22            self.counts['g_leave'] += 1
23    def migrationEvent(self, pop1, pop2, t, k):
24        self.counts['mig'] += 1
25    def mergeEvent(self, pops, t, k):
26        self.counts['me'] += 1
27
```

```

28 import CoaSim
29 from CoaSim.popStructure import Population as P, Merge as M,\
30     Sample as S, Bottleneck as B,\
31     Growth as G, Migration as Mi
32
33 popSpec = P(1,M(1.5,[P(1,S(2),name='1'),P(1,S(2),name='2')]),
34     epochs=[B(.2,1.5,2),G(10,2)])
35 migSpec = [Mi('1','2',0.1),Mi('2','1',0.2)]
36 cb = callbacks()
37
38 CoaSim.simulate([], popSpec, migSpec, rho=40,Q=10,gamma=2,
39     keepEmptyIntervals=True, # NB!
40     callbacks=cb)

```

We specify the callbacks in class `callbacks`, lines 1–26, and pass it to `simulate` in line 40. In this particular example, the callback object implements all callback types. In general this is not necessary; any subset of the callbacks can be implemented.

Say we want to simulate 10,000 coalescent trees simulated under recombination rate $\rho = 2$, but conditional on no recombination in fact did happen. We could, of course, simulate ARGs till completion and reject ARGs containing recombination nodes, but it is much more efficient to abort the simulation once the first recombination occurs.

To reject a simulation when the first recombination event occurs, we can install a callback for recombination events that just throws an exception. We can then wrap the simulation in a function that tries to perform the simulation, and if the exception is throw just restarts with another try:

```

1 from exceptions import Exception
2 from CoaSim import *
3
4 class reject(Exception): pass
5 class RejectionSampler(object):
6     def recombinationEvent(self,n1,n2,k):
7         raise reject()
8 rejectionSampler = RejectionSampler()
9
10 noSamples = 0
11 heights = []
12
13 while noSamples < 10000:
14     try:
15         arg = simulate([], 5, rho=2, keepEmptyIntervals=True,
16             callbacks=rejectionSampler)
17         tree = arg.intervals[0].tree
18         heights.append(tree.height)
19         noSamples += 1
20     except reject:
21         pass
22
23 print 'Average tree height:', sum(heights)/len(heights)

```

We define a rejection exception and a callback to throw it in case of a recombination in lines 4–8. We then wrap the simulation in a `try:...except:...` block and sample only trees where the `reject` exception were not raised, i.e. where no recombination occurred.

Writing Custom Markers

We have seen the three types of built-in markers supported by CoaSim earlier, but it is also possible to define new marker types yourself. To create a custom marker, we sub-class `CustomMarker` from the `CoaSim` module. The class should implement two methods: `defaultValue` and `mutate`. The `defaultValue` method determines the ancestral value for the marker; typically this is a fixed value, but depending on the implementation it can be drawn from some distribution. The `mutate` method is called with the parent allele and the length of an edge, and should return the child allele based on this.

A very simple marker, where the ancestral allele is always 0, each edge contains a mutation (irregardless of edge length) changing alleles 0 to 1 and 1 to 0 would look like this:

```
from CoaSim import CustomMarker
class MyMarker(CustomMarker):
    def __init__(self, pos):
        CustomMarker.__init__(self, pos) # don't forget this!
    def defaultValue(self):
        return 0
    def mutate(self, parentAllele, edgeLength):
        if parentAllele == 0: return 1
        else: return 0
```

Below is shown how a custom marker can be used to define a step-wise mutation model for micro-satellite markers. Here, the initial allele value is always set to 0 and the whole piece of code is an implementation of the mutation model. The `mutate` function uses two helper functions, one for selecting the waiting time for the next mutation, and one for, conditional on a mutation occurring, mutating an allele into another. Waiting times are selected from an exponential distribution, and `mutate` iteratively selects waiting times and apply mutations until the next waiting time moves past the length of the edge being processed.

```
from random import uniform, expovariate
class StepWise(CustomMarker):
    '''The step-wise mutation model for micro-satellites.'''
    def __init__(self, pos, theta):
        '''Initialize marker at position pos with mutation
        rate theta -- the intensity of the waiting time for
        the next mutation event will be theta/2.'''
        CustomMarker.__init__(self, pos) # don't forget!
        self.lambd = float(theta)/2
    def _mutateTo(self, allele):
        '''Choose a new allele -- randomly one step up
```

```

    or one step down.'''
    if uniform(0.0,1.0) < 0.5: return allele-1
    else: return allele+1

def _wait(self):
    '''Draw the waiting time for the next mutation.'''
    return expovariate(self.lambd)

def defaultValue(self):
    '''Initial value -- here always 0.'''
    return 0

def mutate(self, parentAllele, edgeLength):
    '''Step-wise mutations...'''
    time = self._wait()
    allele = parentAllele
    while time < edgeLength: # mutate till end of the edge
        allele = self._mutateTo(allele)
        time += self._wait()
    return allele

```

Contact

For comments or questions regarding CoaSim, please contact Thomas Mailund, at mailund@mailund.dk or mailund@birc.au.dk.