
CoaSim/Guile Manual

Using the Guile-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 Guile-Scheme 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 *CoaSim Scheme Bindings* reference manual, distributed with CoaSim in `/usr/local/share/coasim/doc/bindings.html`, or online at <http://www.daimi.au.dk/~mailund/CoaSim/refman/index.html>.

Some familiarity with the Scheme programming language and with coalescent theory is assumed. For people not familiar with Scheme, an excellent tutorial can be found at <http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.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/Guile* manual (<http://www.daimi.au.dk/~mailund/CoaSim/download/getting-started-guile-4.1.pdf>).

Running CoaSim

The Guile-Scheme based CoaSim tool is run from the command line as:

```
> coasim_guile file-1.scm file-2.scm ... file-n.scm
```

where each file is a scheme script; CoaSim will execute these script files in turn, remembering the state between them, a feature that is sometimes useful for setting up a set parameters in one file, and then running simulations in following files. In most cases, though, a single file will suffice for running one or more simulations. In the rest of this manual we will assume that you know

how to execute scripts with 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 make the common case easy, CoaSim provides the function `simulate-sequences`, that simulates an ARG and then automatically extracts the sequences from the leaf-nodes.

Simulating and Saving Sequences

A simple sequence simulation script could look like this:

```
(use-modules (coasim rand))
(let* ((markers (make-random-snp-markers 10 0 1))
      (seqs (simulate-sequences markers 10)))
  (display seqs)(newline))
```

The first line

```
(use-modules ((coasim rand))
```

includes the `(coasim rand)` 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. The `(let* ...)` construction is just a way of creating a block of scheme code, that let us define variables. The variables we define in the code above are `markers`—containing 10 random SNP markers with allowed mutant-frequencies between 0 and 1, created using the `make-random-snp-markers` function—and `seqs`—the simulated sequences. There are 10 sequences, the second argument to `simulate-sequences`, and each sequence contains an allele for each marker in `markers`.

The sequences, as returned by `simulate-sequences`, are represented as a list of lists of alleles. This is a format that is very easy to manipulate by Scheme, but not as useful for most other programs, so printing the sequences in this format, as we do with the `display` 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:

```
(use-modules (coasim rand) (coasim io))
(let* ((markers (make-random-snp-markers 10 0 1))
      (seqs (simulate-sequences markers 10)))
  (call-with-output-file "positions.txt"
    (marker-positions-printer markers))
  (call-with-output-file "sequences.txt"
    (sequences-printer seqs)))
```

Here, the positions are written to the file `positions.txt` and the sequences to the file `sequences.txt`. The `call-with-output-file` function is a way of opening a file for writing; it opens a file port, then calls it second argument with this port, and then closes the port afterwards. The functions `marker-positions-printer` and `sequences-printer` take care of writing the output. They create functions that fit with the interface of the function `call-with-output-file`, and that write the positions and sequences, respectively, to the port opened by the call to `call-with-output-file`.

Simulation Parameters

As called above, `simulate-sequences` 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-sequences`, 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-sequences markers no-sequences :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-sequences markers no-sequences :gamma 250 :Q 100)
```

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

```
(use-modules (coasim rand))
(let* ((markers (make-random-snp-markers 10 0 1))
      (seqs (simulate-sequences markers 10
                               :rho 400 :gamma 250 :Q 100)))
  (display seqs)(newline))
```

Markers

The first argument to the `simulate-sequences` 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-sequences` 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 rand)` module, as in the case of `make-random-snp-markers` above, but can otherwise be ensured by sorting the markers with the `sort-markers` function from the `(coasim markers)` 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 -ary 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 function `trait-marker`:

```
(trait-marker position low-mutation-freq high-mutation-freq)
```

so e.g.

```
(trait-marker 0.5 0.18 0.22)
```

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 function `snp-marker`

```
(snp-marker position low-mutation-freq high-mutation-freq)
```

while micro-satellite markers can be created with the function `ms-marker`

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

```
(ms-marker 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:

```
(let* ((markers (list (snp-marker 0.1 0.2 0.8)
                     (snp-marker 0.2 0.1 0.9)
                     (trait-marker 0.5 0.18 0.22)
                     (snp-marker 0.7 0.2 0.8)
                     (ms-marker 0.8 1.5 10)))
      (seqs (simulate-sequences markers 10 :rho 400)))
      (display seqs)(newline))
```

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.

Explicitly creating the markers in this way can be cumbersome, so CoaSim provides functions for randomly generating markers, as we have already seen, in the module `(coasim rand)`. The three functions:

```
(make-random-trait-marker n low-freq high-freq)
(make-random-snp-marker n low-freq high-freq)
(make-random-ms-marker n  $\theta$  k)
```

of which we have already used the second several times, creates n markers, randomly positioned, of the respective marker types.

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 `sort-markers` from the module `(coasim markers)` on the combined list, or by combining the lists using the function `merge-markers`, also from `(coasim markers)`, e.g.

```
(use-modules (coasim rand) (coasim markers))
(let* ((snp-markers (make-random-snp-markers 10 0.1 0.9))
      (trait-markers (make-random-trait-markers 2 0.2 0.4))
      (unsorted-markers (append snp-markers trait-markers)))
      (sort-markers unsorted-markers))
```

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

```
(use-modules (coasim rand) (coasim markers))
(let* ((snp-markers (make-random-snp-markers 10 0.1 0.9))
      (trait-markers (make-random-trait-markers 2 0.2 0.4))
      (merge-markers snp-markers trait-markers))
```

that simply merges the two sorted lists. Notice, however, that while `sort-markers` will work on any list of markers, `merge-markers` will only work on already sorted lists.

Another variation, that is sometimes useful, is to insert a marker at the right position in a sorted list such that the resulting list is sorted. This can be done with the function `insert-sorted`, also from `(coasim markers)`:

```
(insert-sorted sorted-marker-list marker)
```

the result is a list with all markers from `sorted-marker` and the new marker, sorted. As an example of this, consider the script:

```
(use-modules (coasim rand) (coasim markers))
(let* ((snp-markers (make-random-snp-markers 10 0.1 0.9))
      (disease-marker (trait-marker 0.5 0.2 0.4))
      (markers (insert-sorted snp-markers disease-marker))
      (seqs (simulate-sequences markers 10)))
  (display seqs)(newline))
```

Here we create 10 randomly positioned SNP markers and a single trait marker (here assumed to be a disease susceptibility gene) and place it in the middle of the interval. We then join the SNP markers with the trait marker using `insert-sorted` and simulate 10 sequences from these markers.

Cases and Controls

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 `split-in-cases-controls-on-marker` function from module `(coasim SNP haplotypes)`.

```
(split-in-cases-controls-on-marker sequence trait-idx)
```

which splits the sequence based on the alleles at an indexed marker. Despite the module name, it is not actually restricted to sequences of SNP haplotypes, but it does assume that the marker at index `trait-idx` is a binary marker with 1 denoting mutants and 0 wild-types. Here, `trait-idx` is the index of the marker to split on, indexing from 0, in sequences. The function returns a list of two lists, the first being the mutants the second the wild-types. Both lists have the marker at `trait-idx` removed, i.e. they contain one less marker than the input sequence. It is possible to disable the removal of the trait alleles in this function by calling the function with the key-word argument `:remove-trait` set to `#f`. In most cases, however, you do not want to keep the trait marker after having determined the case/control phenotype from it, so the default is to remove it.

As an example, consider the following script:

```

(use-modules (srfi srfi-1)) ; get the list-index function
(use-modules (coasim rand)
             (coasim markers)
             (coasim SNP haplotypes))
(let* ((snp-markers (make-random-snp-markers 10 0.1 0.9))
      (disease-marker (trait-marker 0.5 0.2 0.4))
      (markers (insert-sorted snp-markers disease-marker))
      (disease-idx
       (list-index
        (lambda (m) (= 0.5 (position m))) markers))
      (seqs (simulate-sequences markers 10))
      (cases-controls
       (split-in-cases-controls-on-marker seqs disease-idx))
      (cases (car cases-controls))
      (controls (cadr cases-controls)))
      (display cases)(newline)(newline)
      (display controls)(newline)(newline))

```

Here we build a set of randomly placed SNP markers and a single trait marker at the middle of the region, as earlier, and combine them using `insert-sorted`.

```

(use-modules (coasim rand)
             (coasim markers)
             (coasim SNP haplotypes))
(let* ((snp-markers (make-random-snp-markers 10 0.1 0.9))
      (disease-marker (trait-marker 0.5 0.2 0.4))
      (markers (insert-sorted snp-markers disease-marker))
      ...)
      ...)

```

Since we need the index of the disease-marker, to be able to split the sequences, we look that up using the `list-index` function from the module `(srfi srfi-1)`:

```

(use-modules (coasim rand)
             (coasim markers)
             (coasim SNP haplotypes))
(let* (...
      (disease-idx
       (list-index
        (lambda (m) (= 0.5 (position m))) markers))
      ...)
      ...)

```

and given this, we can simulate the sequences, split in cases and controls using `split-in-cases-controls-on-marker` and get the cases as the first element in the resulting list (or in Scheme the `car` of the list) and the controls as the second element (in Scheme, the `cadr` of the list):

```

(use-modules (coasim rand)
             (coasim markers)
             (coasim SNP haplotypes))
(let* (...

```

```

(seq (simulate-sequences markers 10))
(cases-controls
  (split-in-cases-controls-on-marker seqs disease-idx))
(cases (car cases-controls))
(controls (cadr cases-controls)))
...)
```

As an alternative to explicitly looking up the position of the trait marker, you can also insert it into the SNP markers using the function `insert-sorted-idx` from `(coasim markers)`; this function works as `insert-sorted` but returns a list whose `car` is the list of sorted markers, and whose `cadr` is the index the new marker was inserted at. We show the use of this in the following script, where we also place the disease marker at a random position and use the `(coasim io)` functions to write the marker positions and the cases and controls sequences to files. Changes from the previous script are boldface.

```

(use-modules (coasim io)
  (coasim rand)
  (coasim markers)
  (coasim SNP haplotypes))
(let* ((snp-markers
  (make-random-snp-markers 10 0.1 0.9))
  (disease-marker
  (car (make-random-trait-markers 1 0.2 0.4)))
  (m-and-idx
  (insert-sorted-idx snp-markers disease-marker))
  (markers (car m-and-idx))
  (disease-idx (cadr m-and-idx))
  (seqs (simulate-sequences markers 10))
  (cases-controls
  (split-in-cases-controls-on-marker seqs disease-idx))
  (cases (car cases-controls))
  (controls (cadr cases-controls)))
  (call-with-output-file "snp-positions.txt"
  (marker-positions-printer snp-markers))
  (call-with-output-file "trait-position.txt"
  (marker-positions-printer (list disease-marker)))
  (call-with-output-file "cases.txt"
  (sequences-printer cases))
  (call-with-output-file "controls.txt"
  (sequences-printer controls)))
```

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 `split-in-cases-controls-on-marker` using the key-word parameters `:mutant-prob` and `:wild-type-prob`, that sets the probability of a mutant or wild-type, respectively, is selected as a case. To select about 20%

of mutants and only 1% of wild-types, for example, you would use:

```
(split-in-cases-controls-on-marker seqs disease-idx
                               :mutant-prob 0.2
                               :wild-type-prob 0.01)
```

Unphased Genotype Sequences

CoaSim simulates haplotype sequences, but we can combine these to genotype data by pairing sequences. For SNP data, this can be done using the function `haplotypes->genotypes` from module `(coasim SNP genotypes)`:

```
(use-modules (coasim rand) (coasim SNP genotypes))
(let* ((haplotypes
       (simulate-sequences
        (make-random-snp-markers 2 0.1 0.2) 100))
      (genotypes (haplotypes->genotypes haplotypes)))
  (display genotypes) (newline))
```

Using `haplotypes->genotypes`, an even number of haplotype sequences are translated into genotype sequences by combining sequences two and two, translating pairs of alleles by mapping $00 \mapsto 0$, $11 \mapsto 1$, $01 \mapsto 2$, and $10 \mapsto 2$.

Genotype sequences can also be split in cases and controls using the function `split-in-cases-controls-on-marker`, this time from the module `(coasim SNP genotypes)` rather than `(coasim SNP haplotypes)`:

```
(use-modules (coasim rand) (coasim SNP genotypes))
(let* ((haplotypes
       (simulate-sequences
        (make-random-snp-markers 2 0.1 0.2) 100))
      (genotypes (haplotypes->genotypes haplotypes))
      (cases-controls
       (split-in-cases-controls-on-marker genotypes 1))
      (cases (car cases-controls))
      (controls (cadr cases-controls)))
  (display cases) (newline)
  (display controls) (newline)
  (newline))
```

By default, `split-in-cases-controls-on-marker` 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 `:homozygote-0-prob`, setting the probability of a homozygote 00 being a case, `:homozygote-1-prob`, setting the probability of a homozygote 11 being a case, and `:heterozygote-prob`, setting the probability of a heterozygote 01 being a case. 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:

```
(split-in-cases-controls-on-marker genotypes idx
                               :homozygote-1-prob 0.5
                               :heterozygote-prob 0.1)
```

```
:homozygote-0-prob 0.01)
```

As a short-cut for setting common probabilities, we can use the keyword argument `:disease-model`. Setting the disease model to `'dominant'`:

```
(split-in-cases-controls-on-marker genotypes idx  
:disease-model 'dominant)
```

will set the probability for both homozygote mutants and heterozygotes being cases to 1 and the probability for homozygote wild-types being cases to 0, while setting the disease model to `'recessive'`

```
(split-in-cases-controls-on-marker genotypes idx  
:disease-model 'recessive)
```

will set the probability of homozygote mutants being cases to 1 and the probability of the other two genotypes being cases to 0.

Phased Genotype Sequences

Although we will usually work with either haplotype sequences or unphased genotype sequences—since the first is the actual result of the simulated process and the second the usual experimental data available—CoaSim also has some support for phased genotypes. Phased genotypes are simply haplotypes considered in pairs; a list of an even number of haplotypes can also be seen as a list of phased genotypes of half the size.

In the `(coasim SNP genotypes)` module, the `split-in-cases-controls-on-marker/phased` function does exactly this. As input, it takes a sequence, an index, and optionally a `disease-model`—exactly like the `split-in-cases-controls-on-marker` function described above—but this function considers the sequence a list of haplotypes that should, pairwise, be split as genotypes. The result will, as above, be a list of cases and a list of controls, but these are lists with an even number of haplotypes that, again, can be seen as lists of genotypes of half the length.

Complex Disease Models

Splitting the simulated sequences into cases and controls on a single marker, as we have seen now for both haplotype and (unphased) genotype data, is the common case for simulating data for disease mapping applications. More complex disease models require a bit more programming on the user's side, but a number of functions are available in the `(coasim disease-modelling)` module for assisting in this.

The `split-in-cases-controls-on-marker` function in this module is a generalisation of the functions in the `(coasim SNP haplotype)` and `(coasim SNP genotype)` functions of the same name—or as it is, the functions in these modules are implemented as specialisations of this more general function. The function in this module takes a list of sequences (either haplotypes or genotypes), a marker index, and a predicate function used for determining which sequences are cases and which are controls:

```
(split-in-cases-controls-on-marker
  sequences marker-idx is-case?)
```

The predicate function is called with the allele at the marker index in each sequence, and if it returns `#t` the sequence is considered a case sequence, otherwise it is considered a control sequence. So to emulate the default behaviour of the `(coasim SNP haplotype)` function we could use:

```
(use-modules (coasim disease-modelling))
(let ((is-case? (lambda (a) (= 1 a))))
  (split-in-cases-controls-on-marker
    seqs marker-idx is-case?))
```

Using different variations on the `is-case?` predicate, it is simple to implement the different disease models we saw in the previous two sections, and not surprisingly the functions we used there were implemented in exactly this way.

A slightly more general function is `split-in-cases-controls-on-markers` (notice the plural). It is very similar to `split-in-cases-controls-on-marker` (notice singular), except that it takes a list of marker indices and maps these to the predicate to decide on the classification of cases. For example, to model that both the marker at index 2 and the marker at index 4 must be mutants for a disease to develop, we can use:

```
(use-modules (coasim disease-modelling))
(let ((is-case? (lambda (a2 a4) (and (= 1 a2) (= 1 a4)))))
  (split-in-cases-controls-on-markers seqs '(2 4) is-case?))
```

where the call to `split-in-cases-controls-on-markers` specifies that we are interested in the markers at index 2 and 4, `'(2 4)`—which means that the `is-case?` predicate is called with the alleles at those two markers—and the `is-case?` predicate returns true if both alleles are 1.

Both split functions removes the marker(s) used for testing the disease status from the sequences by default, but as before this can be prevented by setting the `:remove-trait` and `:remove-traits` arguments, for the two functions respectively, to `#f`. If they are left in, they can later be removed using the function `remove-alleles` that takes a list of sequences and a list of indices and removes the markers on those indices from all sequences:

```
(use-modules (coasim disease-modelling))
(remove-allele sequences indices)
```

Rather than simply splitting the sequences in two classes, we can associate a value to each, based on alleles at selected markers, and use that to separate cases from controls, based on some threshold value, or randomly, considering the value a probability of being diseased. The function `qtl-on-markers` can be used to associate values to the sequences. It is called similar to the `split-in-cases-controls-on-markers`, but instead of calling a predicate for each sequence it calls a function for calculating the value that should be associated with the sequence.

As an example, we consider marker 2, 3 and 4 and associate a value to each sequence by letting marker 2 contribute 0.5 if it is a mutant (and 0 otherwise) and markers 3 and 4 contribute 0.2 each for mutant alleles (and 0 otherwise) and having the value be the sum of these:

```
(use-modules (coasim markers)
             (coasim disease-modelling)
             (coasim rand))

(define sequences
  (simulate-sequences (make-random-snp-markers 10 0 1) 10))

(define sequences-and-values
  (let ((seq->value
        (lambda (a2 a3 a4)
          (+ (* .5 a2) (* .2 a3) (* .2 a4)))))
    (qtl-on-markers sequences '(2 3 4) seq->value)))

;; display the sequences and the values associated
;; to each sequence
(define (print-seq-and-value seq-and-value)
  (let ((seq (car seq-and-value))
        (value (cadr seq-and-value)))
    (display "seq: ") (display seq) (display " ")
    (display "value: ") (display value) (newline)))
(for-each print-seq-and-value sequences-and-values)
```

As the split functions, `qtl-on-markers` removes the indices used for the value calculations, and as before the `:remove-traits` can be used to disable this.

Once the sequences have been annotated with values in this way, we can use the values to split them in cases and controls. There are two functions to assist in this: `split-on-threshold` and `split-on-probability`. The `split-onthreshold` function simply considers any sequence with a value above a specified threshold a case and all others controls. As an example, if we wish to consider individuals with a mutation at marker 2 and at least one mutation at either 3 or 4, in the previous example, a case, we can use the threshold 0.6. Markers 3 and 4 alone can at most reach 0.4 and marker 2 alone only 0.5, but with a mutation in marker 2 and at least one in the other two we get at least 0.7. We can split on this, using

```
(define cases-and-controls
  (split-on-threshold sequences-and-values 0.6))
(define cases (car cases-and-controls))
(define controls (cadr cases-and-controls))
```

Alternatively, we can consider the values probabilities of being cases and select the cases based on this probability. This is what the `split-on-probability` does; otherwise it works exactly as the `split-on-threshold` function.

```
(define cases-and-controls
  (split-on-probability sequences-and-values))
```

```
(define cases      (car cases-and-controls))
(define controls  (cadr cases-and-controls))
```

Rather than having a function to compute a value for a list of alleles, it is sometimes more convenient to have a table mapping haplotypes or genotypes to values. The function `table->function` can then be used to translate such a table into a function that can be used with the `qtl-on-markers` function.

```
(define prob-table
  (acons '(1 0) .2
         '(1 1) .4
         '()))
(define seq->value (table->function prob-table))
(define sequences-and-values
  (qtl-on-markers sequences '(2 4) seq->value))
```

The `table->function` function can translate both association lists (created with `acons` as above) or hash-tables (created with the `make-hash-table` and `hash-create-handle!` functions) into functions.

Running Multiple Simulations

So far, we have only considered scripts for running a single simulation, but running several simulations is not much more complicated than running a single simulation. Consider this variation on the very first script we saw, with changes boldfaced:

```
(use-modules (coasim batch) (coasim rand))
(repeat 10
  (let* ((markers (make-random-snp-markers 10 0 1))
         (seqs (simulate-sequences markers 10)))
    (display seqs)(newline)))
```

In this version, we load the module `(coasim batch)` which gives us access to the function `repeat` that let us run a script a fixed number of times, in this case 10.

When we are just printing the results of a simulation, as above, the `repeat` function is simple and easy to use, but if we want to save the results in various files we want to be able to use different files for each simulation. One way of doing this is to have a counter that is incremented with each iteration, and use this counter to create new file names for each simulation. Using the function `repeat-with-index`, also from `(coasim batch)`, this could look like:

```
(use-modules (coasim rand) (coasim io) (coasim batch))
(repeat-with-index (i 1 10)
  (let* ((markers (make-random-snp-markers 10 0 1))
         (seqs (simulate-sequences markers 10))
         (pos-file
          (string-append
           "positions." (number->string i) ".txt")))
    (seq-file
     (string-append
```

```

                "sequences." (number->string i) ".txt"))
  (call-with-output-file
    pos-file (marker-positions-printer markers))
  (call-with-output-file
    seq-file (sequences-printer seqs))))

```

Here the variable `i` iterates from 1 to 10, and for each value a list of markers and sequences are generated and written to files `positions.i.txt` and `sequences.i.txt`, respectively.

We will return to `(coasim batch)` again later in this manual, for constructing more complex scripts, but for now we will once again consider running a single simulation per script.

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-sequences markers no-sequences ...)
```

we use the more general form

```
(simulate-sequences markers population-structure ...)
```

where `population-structure` is a 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

```
(simulate-sequences markers
 '(population 1 :epochs (bottleneck 0.2 1.5) (sample 10)))
```

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, `'(population ...)`—notice the quote, `'`, at the beginning; this is needed for the simulator to be able to process the program rather than having Scheme evaluate it. If you are not familiar with how Scheme handles quotes and S-expressions, do not worry about it, just remember to use the quote for the population structure specification.

The specification starts with a population `'(population ...)`—it always does—and the next parameter specifies the size of the population relative to N :



(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).

it is a parameter f' , similar to the bottleneck parameter. In this particular case it is not really useful, but when we add population sub-structure later on, it will be.

The `epochs` parameter (`(population 1 :epochs (...)) ...`) specifies the bottleneck, and we will get back to it shortly. The final parameter is ‘the rest of the specification’, in this case just the number of samples to simulate from the population. We will see more complicated examples in the next section.

The `:epochs` keyword parameter can take two forms: a single epoch, as above, or a list of epochs. 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:

```
(simulate-sequences markers
  '(population 1
    :epochs (bottleneck 0.2 1.5 3) (sample 10)))
```

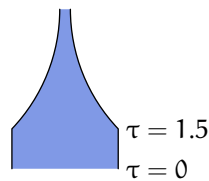
see Fig. 1(b).

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:

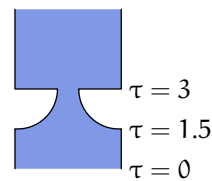
```
(simulate-sequences markers
  '(population 1 :epochs (growth 10 1.5) (sample 10)))
```

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

```
(simulate-sequences markers
  '(population 1
    :epochs (growth 10 1.5 3) (sample 10)))
```



(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).

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.

Since a common case is that the growth is going on in the entire population's lifetime, CoaSim provide a shortcut to specify this, using the keyword `:beta` as in

```
(simulate-sequences
  markers '(population 1 :beta 10 (sample 10)))
```

For convenience, the `:beta` option can also be given to the `simulate-sequences` function directly, when using the simple form where only the sample size is given as the population structure specification, so this

```
(simulate-sequences markers 10 :beta 10)
```

is equivalent to the specification above.

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:

```
(simulate-sequences markers
  '(population 1 :epochs ((bottleneck 0.2 3) (growth 10 0 3))
    (sample 10)))
```

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 ...)` also described above. The nodes—describing population merging events (backward in time, corresponding to splits of populations forward in time)—are specified with the `(merge ...)` construction.

The merge construction takes, as first argument, the time of the merge, backward in time in units of $2N$, followed by 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:

```
(simulate-sequences markers
 '(population 1 (merge 10 (population 1 (sample 20))
 (population 1 (sample 20))))))
```

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” individuals will be from the first population in the population structure specification, the second “sample size” from the next population and so forth.

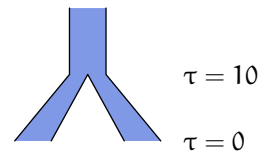


Figure 3: Two populations merging at time $\tau = 10$.

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 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:

```
(simulate-sequences markers
 '(population 2 (merge 10 (population 2 (sample 20))
 (population 1 (sample 20))
 (population 1 (sample 20))))))
```

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

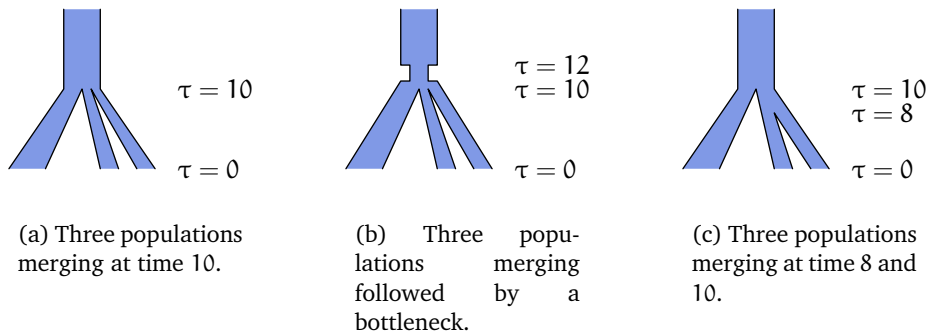


Figure 4: Three different population structures.

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

```
(simulate-sequences markers
  '(population 2 :epochs (bottleneck 0.25 10 12)
    (merge 10 (population 2 (sample 20))
      (population 1 (sample 20))
      (population 1 (sample 20))))))
```

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 sub-populations: $0.25 \cdot 2 = 0.5$, 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

```
(simulate-sequences markers
  '(population 2
    (merge 10
      (population 2 (sample 20))
      (population 1
        (merge 8 (population 1 (sample 20))
          (population 1 (sample 20)))))))
```

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 non-number) as the first parameter to the `(population ...)` construction, thus making the population size the second parameter. Thus, `(population P 1 ...)` assigns the name `P` to a population, of size 1.

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

```
(simulate-sequences markers
  '(population 2
    (merge 10 (population large 2 (sample 20))
              (population small-1 1 (sample 20))
              (population small-2 1 (sample 20))))))
```

giving the larger population the name `large`, and the two smaller populations the names `small-1` and `small-2`, respectively. The populations 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 [start-time end-time])
```

where `[start-time end-time]` is optional 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 `:migration`. 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:²

² Once again, notice the quote, `'`, before the migration list. Without this quote, the simulate

```
(simulate-sequences markers
 '(population 2
   (merge 10 (population large 2 (sample 20))
             (population small-1 1 (sample 20))
             (population small-2 1 (sample 20))))
 :migration '((migration small-1 large 0.5)
             (migration small-2 large 0.5)
             (migration large small-1 1.0)
             (migration large small-2 1.0)
             (migration small-2 small-1 2.0)
             (migration small-1 small-2 2.0)))
```

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:

```
(simulate-sequences markers
 '(population 2
   (merge 10 (population large 2 (sample 20))
             (population small-1 1 (sample 20))
             (population small-2 1 (sample 20))))
 :migration '((migration small-1 large 0.5)
             (migration small-2 large 0.5)
             (migration large small-1 1.0)
             (migration large small-2 1.0)
             (migration small-2 small-1 2.0 5 10)
             (migration small-1 small-2 2.0 5 10)
             (migration small-2 small-1 1.5 0 5)
             (migration small-1 small-2 1.5 0 5)))
```

Simulating the ARG and (Local) Coalescent Trees

Until now we have only considered simulating sequences, but CoaSim actually simulates a full ARG when it is simulating the sequences, and it is possible to access it and extract information about it. In fact, the `simulate-sequences` function is actually a wrapper function doing exactly that:

```
(define (simulate-sequences . params)
 (sequences (apply simulate params)))
```

It builds the ARG by calling the `simulate` function (by applying³ it to the arguments passed to it) and then extracts the sequences from the leaves in the ARG using the function `sequences`. The general function to use—for obtaining other simulation results than the sequences—is `simulate`; a call to `simulate` will return an ARG from which various results can be extracted. Since the function `simulate-sequences` is just a wrapper that calls `simulate` with the parameters it gets, `simulate` takes the same simulation parameters as `simulate-sequences`.

function will not be able to weave the migration list into the population structure.

³The `apply` function is a way of calling a function with a list of parameters, contained in an actual list rather than provided directly to the function. Thus `(apply f '(a b c))` is equivalent to `(f a b c)`.

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 `:keep-empty-intervals` set to `#t`:

```
(define ARG (simulate parameters :keep-empty-intervals #t))
```

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.

Simulating the ARG

As a simple example, we can try simulating an ARG and then extract some statistics from it. We will use three functions for extracting statistics, all from the `(coasim statistics)` module:

- `no-coalescence-events`—returns the number of coalescence events that occurred in simulating the ARG.
- `no-gene-conversions`—returns the number of gene conversions that occurred in simulating the ARG.
- `no-recombinations`—returns the number of recombinations that occurred in simulating the ARG.

Using these could look like this:

```
(use-modules ((coasim statistics)
              :select (no-coalescence-events
                      no-gene-conversions
                      no-recombinations))
              ((ice-9 format) :select (format)))
(let ((ARG (simulate '() 10
                   :rho 40 :gamma 60 :Q 10
                   :keep-empty-intervals #t)))
  (format #t "#recomb: ~d, #gene conv.: ~d, #coalescent: ~d\n"
          (no-recombinations ARG)
          (no-gene-conversions ARG)
          (no-coalescence-events ARG)))
```

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 `:keep-empty-intervals`.

We have also introduced one additional new thing in this example: the `format` function. This function is actually not CoaSim specific, but part of the general Guile library module (`ice-9 format`); it is a convenient way of formatting output, similar to `printf` from C. The first parameter is a port, if the formatted output should be written to that port, `#f` if the formatted output is to be returned as a string, or `#t`—as in the example above—if the output should be output to the current output port, standard out. The second argument is a string describing the output, and the following are values to be formatted. The types and number of these, and the way of formatting them, is determined by the formatting string. For learning more about `format`, we refer to the Guile documentation.

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 use the function `intervals`, as below:

```
(let* ((ARG (simulate '() 10 :rho 40 :keep-empty-intervals #t))
      (inter (intervals ARG))
      ...)
  ...)
```

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

```
(let* ((ARG (simulate '() 10 :rho 40 :keep-empty-intervals #t))
      (inter (intervals ARG))
      (len (lambda (i)
             (- (interval-end i) (interval-start i))))
      (interval-lengths (map len inter))
      ...)
  ...)
```

add the resulting lengths—here by applying function `+` to the length list—and then dividing by the number of intervals:

```
(let* ((ARG (simulate '() 10 :rho 40 :keep-empty-intervals #t))
      (inter (intervals ARG))
      (len (lambda (i)
             (- (interval-end i) (interval-start i))))
      (interval-lengths (map len inter))
      (mean (/ (apply + interval-lengths)
              (length interval-lengths)))
      (display mean)(newline))
```

Simulating Trees

Using functions `interval->tree` and `tree->interval`, we can switch back and forth between local intervals and the coalescence tree for that interval. So to get all the local coalescence trees in the ARG, we can map `interval->tree` over the list of intervals, or alternatively use the function `local-trees` which does exactly this:

```
(define (local-trees ARG) (map interval->tree (intervals ARG)))
```

To simulate a single coalescence tree, we can use code like this:

```
(let* ((ARG (simulate '() 10 :keep-empty-intervals #t))
      (tree (car (local-trees ARG))))
  (display tree))
```

We simulate the ARG and then extract the list of local trees. This will only contain a single tree, since we are simulating without recombination—we have not set the `:rho` parameter, which means that it defaults to 0—and we take this single tree out of the list and display it. Displaying a tree this way will print out the tree in Newick format.

If we add recombination to the simulation, we could also be interested in knowing the extend of the interval for which the tree is the genealogy. We can get this interval together with the tree like this:

```
(let* ((m (snp-marker 0.5 0 1))
      (ARG (simulate (list m) 10 :rho 40))
      (i (car (intervals ARG)))
      (t (interval->tree i)))
  (display (interval-start i))(display "-")
  (display (interval-end i))(display ": ")
  (display t))
```

Here we use a single marker, at the centre of the genomic region—a SNP marker that will accept any minor allele frequency (0.0–1.0), which essentially means that we do not care about the alleles at the marker; it is the local tree for this marker, and the interval for this tree, that we extract from the simulation.

We simulate with the empty intervals optimisation, which guarantees that only intervals containing markers will be returned by the `intervals` function—in this example it guarantees that `intervals` only return the one interval we are interested in—extract the interval and the corresponding local tree—using the `interval->tree` function—and then display the extracted information.

Using the function `tree-height` we can get the hight of the tree, and for example compute the average height along the genomic sequence:

```
(let* ((ARG (simulate '() 10 :rho 40 :beta 10
                    :keep-empty-intervals #t))
      (trees (local-trees ARG))
      (tree-heights (map tree-height trees))
      (mean (/ (apply + tree-heights) (length tree-heights))))
  (display mean)(newline))
```

or the mean height over a number of simulations

```
(use-modules ((coasim batch) :select (tabulate)))
(let* ((no-iterations 10000)
      (tree-heights
       (tabulate no-iterations
                 (let* ((ARG (simulate '() 10
                                       :beta 10
                                       :keep-empty-intervals #t))
                       (tree (car (local-trees ARG))))
                   (tree-height tree))))))
      (display (/ (apply + tree-heights) no-iterations))(newline))
```

In the last example we use the `tabulate` function described earlier to run a number of iterations and collect the result in a list. We simulate with exponential growth ($\beta = 10$); without growth we have a closed form for the mean tree height— $E(H_n) = 2 \left(1 - \frac{1}{n}\right)$, see *Hein et al.* Sect. 1.9—but for $\beta > 0$ we it makes sense to use simulations to get the expected tree height. In the first example, we simulated with recombination ($\rho = 40$) to get a number of intervals; in the second example we do not allow recombination (ρ is implicitly 0) since we only extract a single local tree any way.

Using the function `total-branch-length`, we can get the sum of branch lengths of the tree, and we can calculate the mean of those for varying β just as for tree heights as above.

If the ARG is simulated with a population structure with migration, the edges of the corresponding trees might move between populations, in the sense that different segments of the same branch will correspond to lineages in different populations. From the tree, as such, there is no way to see which populations the various events and branches belong to since, by default, the migration events are not explicitly represented in the ARG or tree structure.⁴ This can be changed, however, by calling `simulate` with `keep-migration-events` set to `#t`.

This will cause migration events to be represented by special nodes in the ARG that can be examined when exploring the ARG (see below) and will cause printed trees to be annotated with migration information in the form of Newick comments. The comments are on the form:

```
(subtree) : [&migration={migration-information}] length
```

where `migration-information` is a list of the form

$$\&migration = \{p_1, mt_1, p_2, mt_2, p_2, \dots, mt_{k-1}, p_k\}$$

where the p_i s are population numbers (as read from top to bottom of the population specification) and the mt_j s are the times of the migration events (measured in global coalescent time $2N$). The list of migration events is read from

⁴The population structure is important for the simulation of the ARG, but conditional on the ARG the mutations can be placed on the sequences independent of the population structure and which population the various lineages belong to. Consequently, as a memory optimisation, the migration events are not represented in the simulated ARG structure, unless explicitly asked for.

top (parent of the branch) to bottom (child of the branch), so p_1 is the population where the parent node belongs and p_k is the population where the child node belongs.

Exploring the ARG

Aside from the functions mentioned above, for extracting information about the ARG, it is possible to extract information from the ARG by explicitly traversing it 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.

The key functions for traversing the ARG are `root`—which returns the root node of a tree, or the *Most Recent Common Ancestor* (MRCA) of an interval—and `children`—which returns a list of the children of a node; this will be two children nodes for coalescence nodes, one child node for recombination and gene-conversions nodes, and an empty list for leaf nodes.

It is possible to check the type of nodes using the predicates `coalescent-node?`, `gene-conversion-node?`, `leaf-node?`, and `recombination-node?` and through these tread the different nodes differently during a traversal. Three additional functions can be used for extracting information about a node:

- `event-time`—returns the time the node was created (in scaled coalescence time, $2N$), and
- `ancestral?`—predicate that determines if the ancestral material in the node contains a given point.
- `trapped?`—predicate that determines if the a point is trapped between ancestral material in a node.

To accumulate information about all the ARG nodes, it is usually more convenient to use the `fold-nodes` function than to explicitly traverse the nodes using `root` and `children`. The function `fold-nodes` takes as parameters, in addition to the ARG, a function taking two parameters, and an initial value for the accumulation. The function is called, for each node in the ARG, with the node and the value accumulated so far, initially the value passed to `fold-nodes`. As an example, to count the total number of nodes in the ARG, we can use:

```
(fold-nodes ARG (lambda (n count) (+ count 1)) 0)
```

or to count the total number of coalescence nodes:

```
(fold-nodes ARG
  (lambda (n count)
    (if (coalescent-node? n) (+ count 1) count))
  0)
```

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.

Scaling Parameters

If we wish to run simulations under various growth parameters, but the same effective recombination rate—and mutation rate in case of micro-satellites—we need to scale ρ and θ for $\beta > 0$ such that e.g. $\rho_{\beta=0} \cdot E(L_{n,\beta=0}) = \rho_{\beta} \cdot E(L_{n,\beta})$, where ρ_{β} and $\rho_{\beta=0}$ are the recombination rates with and without growth, respectively, and $E(L_{n,\beta})$ and $E(L_{n,\beta=0})$ are the expected total branch lengths with and without growth, respectively, see *Hein et al.* Sect. 1.9.

For $E(L_{n,\beta=0})$ we have a formula:

$$E(L_{n,\beta=0}) = 2 \sum_{j=1}^{n-1} \frac{1}{j} \quad (1)$$

but for $E(L_{n,\beta})$ we need to calculate the average total branch lengths over a number of simulations. Given the simulated expected value, we can then rescale ρ and θ accordingly:

$$\rho_{\beta} = \rho_{\beta=0} \cdot \frac{E(L_{n,\beta=0})}{E(L_{n,\beta})} \quad \theta_{\beta} = \theta_{\beta=0} \cdot \frac{E(L_{n,\beta=0})}{E(L_{n,\beta})}$$

In CoaSim, we can calculate the rescaling like this:

```
(use-modules (coasim batch))
(define no-leaves 20)

(define (calc-scale-factor beta)
  (if (= beta 0) 1 ; no need to simulate this one...
      (let ((zero-growth-tree-length
             ;; calculating the tree length without growth
             ;; from closed term formula
             (let loop ((j 1) (sum 0))
               (if (< j no-leaves)
                   (loop (+ j 1) (+ sum (/ 1 j)))
                   (* 2 sum))))
            (growth-tree-length
             ;; simulating the tree length with growth
             (let* ((no-iterations 10000)
                   (branch-sum
                    (fold no-iterations
                          (lambda (val sum) (+ val sum)) 0
                          (let*
                           ((ARG (simulate '() no-leaves
```

```

                                :beta beta
                                :keep-empty-intervals #t))
                                (tree (car (local-trees ARG))))
                                (total-branch-length tree))))
                                (/ branch-sum no-iterations))))
                                (/ zero-growth-tree-length growth-tree-length))))

(define (scaled-simulate beta rho theta)
  (let* ((scale-factor (calc-scale-factor beta))
         (scaled-rho (* scale-factor rho))
         (scaled-theta (* scale-factor theta))
         ...)
    ...))

```

Here we calculate $E(L_{n,\beta=0})$ using (1) and simulate $E(L_{n,\beta})$ by calculating the average branch length over 10,000 iterations. To calculate the average length, we use the function `fold` from module `(coasim batch)`, which works similar to the function `fold-nodes` we saw earlier; given a number of iterations, `no-iterations`, a function for accumulating results

```
(lambda (val sum) (+ val sum))
```

an initial value, 0, and a piece of code for producing the results, we can accumulate results from a number of iterations.

If we run many simulations with the same β , the solution above will be inefficient since we re-calculate the scale factor each time, but it is easy to build a preprocessed table of scale factors and use that:

```

(define (calc-scale-factor beta)
  (if (= beta 0) (cons 0 1) ; no need to simulate this one...
      (let ((zero-growth-tree-length
             ...))
        (cons beta
              (/ zero-growth-tree-length growth-tree-length)))))

(define betas '(0 10 100))
(define scale-factors (map calc-scale-factor betas))

(define (scaled-simulate beta rho theta)
  (let* ((scale-factor (assoc-ref scale-factors beta))
         (scaled-rho (* scale-factor rho))
         (scaled-theta (* scale-factor theta))
         ...)
    ...))

```

Here we change `calc-scale-factor` to return pairs of β and the corresponding scale factor, from which we build an association list—a list of pairs that can be thought of as a mapping from the first elements in the pairs to the second elements—and we look up the scale factors in this association list using the built-in function `assoc-ref`.

Rejection Sampling

There are seven hooks for callbacks in the simulations:

- `:coalescence-callback`—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 lineages just after the event, moving forward in time).
- `:recombination-callback`—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 lineages just after the event, moving forward in time).
- `:geneconversion-callback`—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 lineages just after the event, moving forward in time).
- `:bottleneck-callback`—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.
- `:growth-callback`—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.
- `:migration-callback`—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).
- `:population-merge-callback`—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 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.

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:

```
(define (try-simulate)
  ;; keep trying to simulate until a simulation is NOT aborted
  (let* ((reject-recomb (lambda (n1 n2 k) (throw 'reject)))
        (try (lambda ()
                ;; simulate an ARG, but reject the simulation
                ;; if a recombination occurred
                (simulate '() 10 :rho 2
                        :recombination-callback reject-recomb
                        :keep-empty-intervals #t)))
        (except (lambda (ex-key . ex-args) #f)))
    (let ((res (catch 'reject try except)))
      (if res res (try-simulate)))))
```

Here we use the callback `reject-recomb` to throw the exception `'reject` when a recombination occurs—since `reject-recomb` is installed as the recombination callback through the `:recombination-callback` keyword parameter. The function `try` runs a simulation and returns the ARG if the simulation is not rejected, and the exception handler `except` reports a rejection by returning `#f`. We then evaluate the whole thing using the `catch` function, that wraps the `try` and exception handling, and retry if the result was `#f` rather than an ARG.⁵

By using `try-simulate` we are guaranteed to only obtain ARGs without recombinations, despite the recombination rate $\rho = 5$, and we can run our 10,000 simulations to, say, extract the average tree height, using a loop:

```
(let ((sum-of-heights
      (let loop ((n 0) (sum 0))
        (if (= n 10000) sum ; done
            (let* ((ARG (try-simulate))
                  (h (tree-height (car (local-trees ARG))))
                  (loop (+ n 1) (+ sum h)))))))
  (display (/ sum-of-heights 10000))
  (newline))
```

or using `fold` from `(coasim batch)` again:

```
(use-modules (coasim batch))
(let ((sum-of-heights
      (fold 10000 (lambda (val sum) (+ val sum)) 0
            (tree-height (car (local-trees (try-simulate))))))
  (display (/ sum-of-heights 10000))
  (newline))
```

⁵The reason that we cannot simply try again in the exception handler—which would be the obvious way of doing this—is that we need `try-simulate` to be tail-recursive to avoid stack overflows if many simulations are rejected.

Variable Recombination Rate

In all examples so far, we have used a constant recombination rate, ρ , over the region we simulate. We can think of this as working with a genetic distance rather than physical distance between loci on the region; the distance between two markers is linearly proportional to the expected recombinations between them.

It is well known that the rate of recombination varies across the genome, so if we simulate markers placed on a ‘physical distance’ region, we no longer expect a linear relationship between distance and recombination. We can, however, simulate variable recombination rates simply by scaling the physical distance into genetic distance and simulate with the scaled markers.

If we split the (physical distance) interval $0 - 1$ into segments, with each segment getting a different recombination rate, we can rescale the interval $0 - 1$ by making the length of each segment proportional to the original length of the segment times the recombination rate of the segment. Let l_i be the length of segment i and r_i the rate of segment i , and let $bp_i = \sum_{j < i} l_j$. Then, for any locus x : $bp_i \leq x < bp_{i+1}$, we can rescale x to

$$x' = \frac{r_i \cdot (x - bp_i) + \sum_{j < i} l_j \cdot r_j}{\sum_j l_j \cdot r_j}.$$

The locus is scaled according to the total scaled range of the previous segments, from 0 to bp_i , by adding $\sum_{j < i} l_j \cdot r_j$, and then the expected recombination in the current segment, $r_i \cdot (x - bp_i)$. The scaled loci should all be moved to the interval $0 - 1$ for CoaSim to be able to handle them, thus the final scaling constant, $1 / \sum_j l_j \cdot r_j$.

The function `rescale-markers!` from module `(coasim markers)` performs exactly this rescaling. It takes a list of markers and a list of recombination rates covering the interval $0 - 1$, and scales the markers positions. The list of recombination rates is a list of the form

```
{((length-1 rate-1) (length-2 rate-2) ... (length-n rate-n))}
```

where the lengths must sum to 1. This list provides the l_j s and r_j s above. The list of markers, which must be sorted wrt. their position, provides the x s above and are rescaled.

For example, to give the third quarter of the interval $0 - 1$ a recombination rate ten times as high as the rest of the interval, we could use

```
(define recomb-rates '((0.5 1) (0.25 10) (0.25 1)))  
(rescale-markers! markers recomb-rates)
```

The actual value of the rates in this list is not so important, only the relative rates: all loci are scaled down to the interval $0 - 1$ as described above, and the total recombination on this interval is determined by the ρ parameter given to the `simulate` function. The actual recombination rate of a segment with length l and rate r would be $\rho \cdot \frac{r \cdot l}{\sum_j r_j \cdot l_j}$, thus for the third quarter of $0 - 1$ to have recombination rate 400 (1cM assuming the effective population size

is 10,000) we should simulate with $\rho = \frac{400 \cdot (0.5 \cdot 1 + 0.25 \cdot 10 + 0.25 \cdot 1)}{0.25 \cdot 10} = 520$. On the other hand, to let the entire interval have recombination rate $\rho = 400$, the first half would be scaled to rate $400 \cdot \frac{0.5 \cdot 1}{3.5} = 57.14$, the third quarter would be scaled to rate $400 \cdot \frac{0.25 \cdot 10}{3.5} = 285.71$, and the final quarter would be scaled to rate $400 \cdot \frac{0.25 \cdot 1}{3.5} = 28.57$.

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, you call the function `custom-marker` with a position, an ancestral allele value (or a function for creating it, if the ancestral allele should be random), and a function for handling mutations:

```
(custom-marker pos allele-value mutate)
```

The ancestral allele value, `allele-value`, should either be an integer, or a function taking no arguments and returning an integer value. The `mutate` function is called with three parameters for each edge in the ARG: the parent node of the edge, the child node of the edge, and the parent allele of the edge. It is then responsible for returning the allele for the child; this will be the same as for the parent if no mutation occurs on the edge.

Below is shown how the `custom-marker` function 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.

```
(define (step-ms-marker pos theta)
  (let* ((msec (cdr (gettimeofday)))
        (random-state (seed->random-state msec))

        (waiting-time
         ;; waiting time for next mutation
         (lambda ()
           (let ((mean (/ 2 theta)));mean is 1/i where the
              ;intensity i is theta/2
              (* mean (random:exp random-state))))))

        (mutate-to
         ;; randomly mutating up or down
         (lambda (parent-allele)
           (if (< (random 1.0 random-state) 0.5)
               (- parent-allele 1)
               (+ parent-allele 1))))))
```

```

(mutate
  ;; mutation function to use in the custom marker
  (lambda (parent child parent-allele)
    (let loop ((allele parent-allele)
              (time-left
                (- (- (event-time parent)
                     (event-time child))
                  (waiting-time))))
      (if (< time-left 0)
          allele
          (let ((new-allele (mutate-to allele))
                (next-time (waiting-time)))
            (loop new-allele
                  (- time-left next-time)))))))
(custom-marker pos 0 mutate))

```

The step-wise model, as implemented above, is also available in the module (`coasim stepwise`), implemented similar to above.

Customising CoaSim

CoaSim is highly programmable—as we have seen in the later parts of this manual—and at some point you might wish to extend it with a few of your own custom functions. You can, of course, include those functions in the various simulation scripts you write, but it is more convenient to keep the general functions in a single copy—to avoid the inevitable problems with redundancy—and just have them available for use in the simulation scripts.

Start-up File

Upon start-up, CoaSim checks if the file `~/ .coasim/startup.scm` exists, and if so, loads it in and execute it before any other script is run. Since any code in this file will be executed before any scripts, it is an ideal place to put your custom functions if you want them to be available to all scripts. As `~/ .coasim/startup.scm` is *always* executed, it can hurt runtime performance to put time-consuming computations here, so that is best avoided.

Local Modules

The CoaSim Scheme modules we have used in this manual—the modules on the form (`coasim module-name`)—are globally installed together with CoaSim, but it is also possible to install modules locally. Before searching globally for a module—using a path that depends on the platform, but is most often `/usr/local/share/coasim/scheme`—it searches in the directory `~/ .coasim/`; by placing a module in this directory you can include it in your scripts using `use-modules`.

For example, to create a module, (`stepwise`), containing the step-wise mutation model micro-satellite described earlier, we can save the code

```
(define-module (stepwise))

(define-public (step-ms-marker pos theta)
  (let* ((msec (cdr (gettimeofday)))
        ;; ... same as before ...
```

in the file `~/ .coasim/stepwise.scm`. The `define-module` in the beginning of the file informs the scheme interpreter that this really is a module, with the right name, and the `define-public`, rather than just `define`, exports the function from the module to the script including the module. This module can now be included in your scripts, similarly to the globally installed modules, by the `use-modules` function:

```
(use-modules (stepwise))
```

For this particular case, the step-wise mutation model, you can also use the `(coasim stepwise)` module distributed with CoaSim, but in general, custom modules can be used to extend the functionality of the simulator in a flexible way.

Contact

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