
Getting Started with CoaSim/Guile

An introduction to the simulator CoaSim

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 CoaSim

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 or Python) for efficient batch simulations. This document is an introduction to the Guile-Scheme based version.

Installing CoaSim

CoaSim is distributed as RPM files or as source code. For most users, we recommend installing from the RPM files, since building the tool from source requires setting up the right build environment and having access to the needed development tools. If you are not familiar with UNIX C++ development—using the Automake suite of tools—we do not recommend that you try building from source.

Installing the RPM Files. The RPM file, `coasim-guile-x.y.z-r.i386.rpm`, contains a binary version of the program, compiled to an Intel x86 Linux platform. To install the Scheme version, run

```
> rpm -Uvh coasim-guile-x.y.z-r.i386.rpm
```

Since the RPM files installs in the directory `/usr/local/`, installing the RPM package requires root access.

Installing from the Source Files. The source code is distributed in a tar-file: `coasim-guile-x.y.z.tar.gz`. To build the Guile version from the source files untar the file, build the Core module

```
> tar xzf coasim-guile-x.y.z.tar.gz
> cd coasim-guile-version/Core
> ./configure
> make
```

and then build the Guile module:

```
> cd ../Guile
> ./configure
> make
```

Running CoaSim

The scheme based version is started from the command-line; the parameters for the simulation or simulations to be run is described in one or more configuration scripts, which are written in the Scheme programming language, see the

[CoaSim/Scheme Manual](#) for documentation. Starting CoaSim with a configuration script `simulation.scm`—for example—is done as:

```
> coasim_guile simulation.scm
```

Run `coasim_guile --help` to get a complete list of command-line options accepted by CoaSim.

Using CoaSim

Running CoaSim will in most cases consist of three steps: Set up the simulation parameters, including the markers (type of marker, mutation rates and similar), demographic parameters, rates for recombination, etc.; running the simulation obtaining an ARG; and extracting the needed information from the ARG (*Ancestral Recombination Graph*), e.g. the resulting sequences, the timing of the various events, or the local coalescent trees embedded in the ARG.

Before using the Guile-Scheme based CoaSim *it is necessary that you have installed the program*, not just compiled the program; at least if you are using any of the scheme modules distributed with CoaSim. If the tool has not been properly installed, it will not be able to locate the modules and the scripts, including several of the included test scripts, will not be able to run. If it is not possible to install the modules globally, you can install them locally but you will then need to set the `GUILLE_LOAD_PATH` environment variable to let CoaSim know where the modules are installed. Consult the Guile manual for further details (<http://www.gnu.org/software/guile/docs/>).

Once CoaSim is successfully installed, it can be run with any number of files as argument:

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

CoaSim will execute these files as scheme programs in turn, remembering the state between them, a feature that is sometimes useful for setting up a set of 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.

Controlling the simulations through scheme makes CoaSim a very flexible and powerful simulation tool. However, with flexibility inevitably associated some complexity, and while running simple simulations through the scheme interface is quite simple, the more complex simulation tasks require a bit of knowledge about the Scheme programming language and the scheme modules supplied with CoaSim. To keep this ‘getting started’ guide short, we do not attempt to explain the scheme interface to CoaSim in detail here—for this we refer to the [CoaSim/Guile Manual](#)—instead we give a short introduction to running very simple simulations, and give an overview of the example scripts distributed with CoaSim.

A very simple use of CoaSim is to simulate coalescent trees. A script for that is shown here:

```
(let* ((m (snp-marker 0.5 0 1))
      (arg (simulate (list m) 10)))
```

```
(tree (car (local-trees arg)))  
(display tree))
```

This might look a bit complicated, but if you break it down in the three phases mentioned at the beginning of this section—setting up parameters, running a simulation, and analysing the result—it is really not.

The `(let* ...)` construction is just a way of creating a block of scheme code, that lets us define variables. The variables we define in the code above are `m`, `arg`, and `tree`. The variable `m` is part of the first phase, setting up parameters. In fact, it is the only part of this phase. The expression

```
(let* ((m (snp-marker 0.5 0 1))  
      ...)  
)
```

defines `m` to be a SNP marker, positions at the middle of the genomic region we consider (position 0.5), with the 1-allele frequency to be between 0 and 1, i.e. unrestricted.

The next line is the simulation phase, it defines the variable `arg` to hold the ARG that is the result of the simulation.

```
(let* (...  
      (arg (simulate (list m) 10))  
      ...)  
)
```

The first parameter of the simulation is a list of markers—in this case just the single marker `m` from above. The second argument is the number of chromosomes to simulate, in this case 10.

Once the ARG is simulated, we can start the analysis phase; in this example it is very simple: we simply print the coalescent tree. We obtain the tree by taking the first local tree from the ARG (recombinations will split the genomic region into intervals with different trees, but in this example will only get a single tree, so the tree we are interested in is the first and only local tree). Selecting this tree is done with

```
(let* (...  
      (tree (car (local-trees arg))))  
)
```

where `local-trees` extracts the list of local trees from the ARG, and `car` (which is scheme-speak for the head of a list) selects the first tree in the list. The final statement in the script:

```
(let* (...  
      (display tree))
```

simply prints the tree.

As another simple application, consider simulating a list of SNP sequences. A script for simulating 100 sequences with 10 SNP markers in each is shown here:

```
(use-modules ((coasim rand)))
(let* ((markers (make-random-snp-markers 10 0 1))
      (seqs (simulate-sequences markers 100 :rho 400)))
  (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 following lines are similar to the script above

```
(let* ((markers (make-random-snp-markers 10 0 1))
      (seqs (simulate-sequences markers 10 :rho 400)))
  (display seqs)(newline))
```

except that we now make a list of 10 markers, using a function from the module we just included, `make-random-snp-markers`, simulate a set of sequences rather than the ARG, and with 100 rather than 10 chromosomes, and with the recombination rate ρ set to 400 (which for an effective population size N_e of 10,000 roughly correspond to 1cM). Actually, the `simulate-sequences` function is just a short-cut for an ARG simulation using `simulate`, as before, and a function, `sequences`, for extracting the sequences from an ARG. The following script is thus equivalent to the script above:

```
(let* ((markers (make-random-snp-markers 10 0 1))
      (arg (simulate markers 10 :rho 400))
      (seqs (sequences arg)))
  (display seqs)(newline))
```

A script very similar to the tree simulation script.

Printing sequences with the `display` function will print a list of lists; this is a format that is very simple for Scheme to read and manipulate, but is usually not suitable for other analysis tools. 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 :rho 400)))
  (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 and the `marker-positions-printer` and `sequences-printer` take care of writing the output.

Running several simulations is not much more complicated than running a single simulation. Consider the following script that simulates 10 coalescent trees instead of only one:

```
(use-modules ((coasim batch) :select (repeat)))
(repeat 10 (let* ((m (snp-marker 0.5 0 1))
                (arg (simulate (list m) 10))
                (tree (car (local-trees arg))))
            (display tree)))
```

The first line loads a module—just as we loaded the module `(coasim rand)` above—but with a slight variation that only loads a single function, `repeat`, and not the entire module. In most cases you would just load the module, but we only load `repeat` here to give an example of how this is done. Loading single functions from a module, instead of the entire module, can be useful to avoid name-clashing in your scripts.

The next statement calls `repeat` with two arguments, the number of simulations to run, and the code for the simulation; the latter you will recognise as the same code we used above to simulate a single tree.

When conducting a large number of simulations, it is usually not desirable to print all results, but rather to calculate some statistics about the simulations. As a very simple example, instead of printing simulated trees, we can calculate the mean of the total branch length of the trees. For this we again use the module `(coasim batch)`, but rather than using `repeat` we use the function `tabulate` that lets us collect simulation results. The complete script looks like this:

```
(use-modules ((coasim batch) :select (tabulate)))
(let* ((no-iterations 10000)
      (branch-lengths
       (tabulate no-iterations
                (let* ((m (snp-marker 0.5 0 1))
                      (arg (simulate (list m) 10))
                      (tree (car (local-trees arg))))
                  (total-branch-length tree))))))
      (display (/ (apply + branch-lengths) no-iterations))
      (newline))
```

We fix the number of simulations to use and give it the name `no-iterations`, using the `(let* ...)` syntax, we then call `tabulate` with the number of iterations and the code for simulating a tree and getting its branch length. The main difference in the simulation code is that we call the function `total-branch-length` on the tree instead of printing it using `display`.

The result of the `tabulate` call is a list of simulation results, which we store in the variable `branch-lengths`. We then use this list to calculate the mean by summing the values (using the function call `(apply + branch-lengths)`) and dividing it with the number of iterations `(/ ... no-iterations)`, and we finally print the result using `display`.

Calculating the branch-length like this is a bit silly, since we can calculate the

expected branch length for a coalescent tree with n nodes simply as

$$2 \sum_{i=1}^{n-1} \frac{1}{i}$$

but when we add exponential growth, no such simple formula exists. As a final example of running simulations in CoaSim, consider calculating the mean branch length for various growth parameters β :

```
(use-modules ((coasim batch) :select (tabulate)))

(define betas '(0 10 20))
(define (mean-branch-length beta)
  (let* ((no-iterations 1000)
        (branch-lengths
         (tabulate no-iterations
                   (let ((arg (simulate (list (snp-marker 0.5 0 1))
                                         10 :beta beta)))
                     (total-branch-length (car (local-trees arg)))))))
        (/ (apply + branch-lengths) no-iterations)))

(display (map mean-branch-length betas))(newline)
```

This looks very similar to the simulation code above, but we have now defined a list of β values:

```
(define betas '(0 10 20))
```

and made the mean branch lengths calculation into a function of β :

```
(define (mean-branch-length beta)
  ...)
```

The way that β is specified to the simulation is through the keyword argument `:beta` value in

```
(simulate (list (snp-marker 0.5 0 1)) 10 :beta beta)
```

The final line in the script ‘maps’ the `mean-branch-length` function over the β values, which means that it applies the function on each β and collect the results in a list, and it then prints this list using `display`.

Calculating the mean branch length in this way is inefficient, since we build a list of all simulated values and then sum the elements; it would be more efficient to calculate the sum during the iterations. This can also be done, using the `fold` function rather than `tabulate`:

```
(use-modules ((coasim batch) :select (fold)))

(define betas '(0 10 20))
(define (mean-branch-length beta)
  (let* ((no-iterations 10)
        (branch-sum
         (fold no-iterations (lambda (val sum) (+ val sum)) 0
```

```

      (let* ((m (snp-marker 0.5 0 1))
            (arg (simulate (list m) 10 :beta beta))
            (tree (car (local-trees arg)))
            (branch-length
              (total-branch-length tree)))
            (branch-length)))
    (/ branch-sum no-iterations))

(display (map mean-branch-length betas)) (newline)

```

The `fold` method, which will be familiar to people with exposure to functional programming, builds a value from the simulations using a function for combining the result so far with the latest simulated value, and an initial value. In the example above, the function

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

adds a new value to the sum so far, with the initial value 0. Folding with this pair of combination function and initial value lets us calculate the sum of branch lengths without building a list of the simulated values.

For further examples, see `/usr/local/share/coasim/test-input/` containing the files (in order of increasing complexity):

- `tree.scm`—the tree example from above.
- `interval-and-tree.scm`—simulation of a tree and its local interval under recombination.
- `sequences.scm`—the sequence example from above.
- `arg-statistics.scm`—extracting the number of different kinds of events from a simulation.
- `mean-interval-length.scm`—calculates the mean length of local intervals for an ARG.
- `mean-tree-height-1.scm`—calculate the mean height of trees along the genomic region.
- `mean-tree-height-2.scm`—calculate the mean height of trees over multiple simulations.
- `snp-haplotype.scm`—a more detailed example of simulating SNP sequences.
- `snp-haplotype-split.scm`—an example of simulating SNP sequences and splitting them into cases and controls, based on a trait-marker.
- `more-trees.scm`—the simulation and printing of several trees, from above.
- `more-sequences.scm`—simulation of several sequences, writing them to different files.

- `mean-branch-length.scm`—the calculation of the mean branch length of trees, from above.
- `more-mean-branch-length.scm`—the calculation of the mean branch length of trees with various β values, from above.
- `fold-mean-branch-length.scm`—the calculation of the mean branch length of trees with various β values using `fold`, as above.
- `scaling.scm`—rescaling ρ and θ according to simulated average branch lengths.
- `ms.scm`—an example of simulating sequences and outputting them in Hudson’s `ms` format.
- `rejection-sampling.scm`—example of using callbacks to do rejection sampling.

Contact

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