GeneRecon Users' Manual

A coalescent based tool for fine-scale association mapping

Thomas Mailund

mailund@birc.au.dk

Bioinformatics ApS

GeneRecon v.1.2

April 2006

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 GeneRecon

GeneRecon is a software package for linkage disequilibrium mapping using coalescent theory. It is based on Bayesian Markov-chain Monte Carlo (MCMC) method for fine-scale linkage-disequilibrium gene mapping using high-density marker maps. GeneRecon explicitly models the genealogy of a sample of the case chromosomes in the vicinity of a disease locus. Given case and control data in the form of genotype or haplotype information, it estimates a number of parameters, most importantly, the disease position. Some of the theory underlying GeneRecon is described in:

> Rannala, B. and Reeve, J.P. (2001) *High-resolution multipoint linkage-disequilibrium mapping in the context of a human genome sequence.* Am. J. Hum. Genet. 69: 159–178.
>
> Morris, A.P., Whittaker, J.C., and Balding, D.J. (2002) *Fine-scale mapping of disease loci via shattered coalescent modeling of genealogies.* Am. J. Hum. Genet. 70:686–707.
>
> Rafnar, T., Thorlacius, S., Steingrimsson, E., Schierup, M.H., Madsen, J.N., Calian, V., Eldon, B.J., Jonsson, T., Hein, J., and Thorgeirsson, S.S. (2004) *The Icelandic Cancer Project—A population-wide approach to studying cancer*. Nature Reviews Cancer. 4:488–492.

GeneRecon is written in C++ and is available as a command-line executable for Linux. The Scheme programming language is used for specifying the input to GeneRecon and controlling its execution.

In the following, we assume that you have successfully installed GeneRecon, if not we refer to the *Getting Started with GeneRecon* manual (`http://www.daimi.au.dk/~mailund/GeneRecon/download/getting-started-1.0.pdf`).

Running GeneRecon

GeneRecon is started from the command-line; the specifics about the data-set to analyze and how it is to be performed, is described in one or more configuration scripts, which are written in the Scheme programming language, see `/usr/local/share/generecon/doc/bindings.html`, if GeneRecon is locally installed, or `http://www.daimi.au.dk/~mailund/GeneRecon/refman/index.html`, online, for documentation. Starting GeneRecon with the configuration script `analysis.scm` is done as:

```
> generecon analysis.scm
```

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

Specifying the sequence data

The first step in configuring GeneRecon for an MCMC calculation is describing the genomic region the calculation should examine, then specifying the haplotype or genotype sequences to be analysed. The region-object specifies properties such as allelic frequencies and positions of markers in the region being analysed, while the sequences hold the haplotype/genotype information. Together, they can be combined into various types of datasets where phenotype is also associated with the sequences.

Specifying a genomic region

A genomic region is created using the function `region` that takes a recombination rate, a mutation rate, and a list of markers as input. These markers, in turn, are created using the `marker` function.

Let us consider a small (toy) example, where we define a region containing three markers, two bi-allelic markers on positions 0.2 and 0.8 respectively, and one three-allelic marker on position 0.45. (The units for the positions are not important, nor are the absolute values, the only important usage GeneRecon has for the positions is their relative positions, i.e. the ordering and the differences in density over the region).

The objects representing these markers are created using the `marker` function, that takes the position and the frequencies for the different alleles as arguments. In GeneRecon, alleles are identified using numbers from zero to the number of alleles for a marker minus one, and the frequencies are given as a list where the first element is the frequency of the first allele, the second element the frequency of the second allele and so forth. Defining the markers like this:

```
(define m1 (marker 0.2  '(0.2 0.8)))
(define m2 (marker 0.45 '(0.1 0.7 0.2)))
(define m3 (marker 0.8  '(0.7 0.3)))
```

Specifies that marker m_1 has frequency 0.2 for allele 0 and frequency 0.8 for allele 1, similarly marker m_2—the three-allelic marker—has frequency 0.1 for allele 0, frequency 0.7 for allele 1, and frequency 0.2 for allele 2, and marker m_3 has frequency 0.7 for allele 0 and frequency 0.3 for allele 1.

Using the module `(generecon common)`, we can define a list of markers slightly simpler[1]—from a list of positions and a list of frequencies for each position—using the function `make-markers`. The markers above can, for example, be defined using

```
(use-modules (generecon common))
(define markers
   (let ((positions '(0.2 0.45 0.8))
         (freq-list
            (list '(0.2 0.8) '(0.1 0.7 0.2) '(0.7 0.3))))
     (make-markers positions freq-list)))
```

[1] If this does not, at first sight, seem simpler than defining the markers one by one as above, consider the benefits of using this function when the positions and/or frequencies are computed or read from a file.

If the position data is given, not as the relative positions of markers but their relative *distance*, the function `distances→positions` can be used to translate this into positions:

```
(use-modules (generecon common))
(define markers
   (let ((positions (distances->positions '(0.25 0.35)))
         (freq-list
            (list '(0.2 0.8) '(0.1 0.7 0.2) '(0.7 0.3))))
     (make-markers positions freq-list)))
```

If, for the (per-generation, per-region) recombination rate, κ, we choose 5×10^{-3}, and for the (per marker)[2] mutation rate, μ, we choose 1×10^{-3}, we can define these as:

```
(define kappa 5e-3)
(define mu 1e-3)
```

Defining a region, then, given κ, μ, and the markers, is as easy as:

```
(define reg (region kappa mu (list m1 m2 m3)))
```

A region created this way, assumes linkage equilibrium between markers among unaffected individuals; the genealogy is explicitly modeled between affected individuals so there, no assumptions are made about linkage equilibrium. Depending on the relatedness of unaffected individuals (the relatedness in the population as such) and the closeness of markers, this assumption might be very far from true. In such cases, GeneRecon can instead model haplotypes as a first-order Markov chain, thus introducing a very simple dependency between markers. If this is desired, we create a Markov chain using the `markov-chain` function. This function takes as input a number of lists, one for each marker minus one, such that list number i specifies the allele frequencies for marker number i, conditional on the allele at marker $i-1$. That is, list number i contains a list for each allele at marker $i-1$, containing frequencies for alleles at marker i. This way, the Markov chain:

```
(markov-chain
  (list (list 0.2 0.5 0.3) (list 0.5 0.4 0.1))
  (list (list 0.7 0.3) (list 0.4 0.6) (list 0.6 0.4)))
```

specifies the conditional probabilities

$$P(m_1 = 0 \mid m_0 = 0) = 0.2 \quad P(m_1 = 1 \mid m_0 = 0) = 0.5 \quad P(m_1 = 2 \mid m_0 = 0) = 0.3$$

$$P(m_1 = 0 \mid m_0 = 1) = 0.5 \quad P(m_1 = 1 \mid m_0 = 1) = 0.4 \quad P(m_1 = 2 \mid m_0 = 1) = 0.1$$

$$P(m_2 = 0 \mid m_1 = 0) = 0.7 \quad P(m_2 = 1 \mid m_1 = 0) = 0.3$$

[2]The mutation rate is, in the current version of GeneRecon, assumed to be the same for each marker, and the parameter μ determines this rate. Notice that this is the mutation rate for the *individual markers*, not the region; the absence of segregating sites over a stretch of the region is not interpreted as the absence of mutations in that stretch, just as the absence of markers selected for the study over that stretch. Each present marker, however, is assumed to evolve under the k-allele model with mutation rate μ.

$$P(m_2 = 0 \mid m_1 = 1) = 0.4 \quad P(m_2 = 1 \mid m_1 = 1) = 0.6$$
$$P(m_2 = 0 \mid m_1 = 2) = 0.6 \quad P(m_2 = 1 \mid m_1 = 2) = 0.4$$

A Markov chain is given to the `region` function as an optional fourth argument, after the list of markers. So extending the example from above with a Markov chain, we would use:

```
(define reg
  (region
     kappa mu (list m1 m2 m3)
     (markov-chain
        (list (list 0.2 0.5 0.3) (list 0.5 0.4 0.1))
        (list (list 0.7 0.3) (list 0.4 0.6) (list 0.6 0.4)))))
```

Specifying sequences

Once the region has been constructed, we can specify sequences over it, as haplotypes (for phased data) or genotypes (for unphased data), using functions `haplotype` and `genotype`, respectively. Both functions take as first argument the region and as second the alleles for each marker in the region; they differ in the way the alleles are specified: a haplotype has a single allele per marker where a genotype has a pair of alleles per marker.

Building a haplotype over the region defined above, could look like this:

```
(define h (haplotype reg '(0 2 1)))
```

The haplotype, h, defined here has allele 0 at marker m_1, allele 2 at marker m_2 and allele 1 at marker m_3. A genotype over the region defined quite similarly, except that it takes a list of pairs of alleles:

```
(define g (genotype reg (list (0 . 1) (1 . 2) (0 . 1))))
```

The pairs here are considered unordered; the first element in the pairs are not necessarily on the one underlying chromosome with the second element on the other.

The alleles are given as numbers from 0 to $k-1$ where k is the number of alleles of the marker, i.e. the length of the frequency list for that particular marker. Any missing data is given as -1.

Haplotypes and genotypes can also be defined in a slightly simpler form, similar to markers, using the functions `haplotype-list→haplotype-list` and `genotype-list→genotype-list` in the modules `(generecon SNP haplotype)`, `(generecon SNP genotype)`, `(generecon MS haplotype)`, and `(generecon MS genotype)`, respectively. The `haplotype` modules for phased data and the `genotype` modules for unphased, as described above. The `SNP` (Single Nucleotide Polymorphism) modules are for bi-allelic markers and the `MS` (Micro-Satellite) modules are for k-allelic ($k > 2$) markers.[3]

[3]The distinction between bi-allelic and k-allelic is not important for building haplotype and genotype objects, as such, but the bi-allelic case is slightly simpler (and also becoming the most common case as SNP polymorphism is the most common polymorphism in humans and since

These functions all take a region as the first parameter and for the SNP case a list of alleles (or allele pairs) as the second argument, and build a list of haplotypes (or genotypes). For haplotypes, the code could look like this:

```
(use-modules (generecon common) ; included for make-markers
             (generecon SNP haplotype))
(define markers
   (let ((positions '(0.2 0.45 0.8))
         (freq-list (list '(0.2 0.8) '(0.3 0.7) '(0.7 0.3))))
     (make-markers positions freq-list)))
(define reg (region 5e-3 1e-3 markers))
(define haplotypes
  (haplotype-list->haplotype-list
   reg (list '(0 1 0) '(0 2 1) '(1 0 1))))
```

For SNP genotypes, the pair of markers are *not* encoded as pairs—as they would be if they were build directly using the `genotype` function—but rather as integers -1, 0, 1, and 2 (with -1 being used for missing data): 0 for homozygote 0, 1 for homozygote 1, and 2 for heterozygote.

```
(use-modules (generecon common) (generecon SNP genotype))
(define markers
   (let ((positions '(0.2 0.45 0.8))
         (freq-list (list '(0.2 0.8) '(0.3 0.7) '(0.7 0.3))))
     (make-markers positions freq-list)))
(define reg (region 5e-3 1e-3 markers))

(define genotypes
  (genotype-list->genotype-list
   reg '((0 1 0) (0 2 0) (1 0 2))))
```

For k-allelic data, the functions take an additional argument: a mapping from alleles to indices from 0 to $k-1$ (the numbers that should be used if using `haplotype` or `genotype` directly). This mapping allow us to express alleles using any numbers, not only -1 to $k-1$ (with -1 as usual meaning missing data). The mapping from arbitrary numbers to indices can be constructed using the function `make-index-tables`, found both in the `(generecon MS haplotype)` and the `(generecon MS genotype)` module.

For haplotypes the creation of the mapping and the subsequent list of haplotypes can be coded as this:

```
(use-modules (generecon common) (generecon MS haplotype))
(define markers
   (let ((positions '(0.2 0.45 0.8))
         (freq-list
            (list '(0.3 0.7) '(0.3 0.3 0.4) '(0.7 0.1 0.2))))
     (make-markers positions freq-list)))
(define reg (region 5e-3 1e-3 markers))
```

SNP genotyping technology is highly developed). Therefore, the syntax for dealing with bi-allelic markers has been made slightly simpler, at the cost of having modules for both the SNP and MS data.

```
(define haplotypes
  (let* ((allele-lists '((0 1 5)
                        (0 2 0)
                        (1 4 2)))
         (mapping (make-index-tables allele-lists)))
    (haplotype-list->haplotype-list
     reg mapping allele-lists)))
```

where `mapping`[4] will be a table mapping the alleles in the first column of the `allele-list`, 0 and 1, to the indices 0 and 1, the second column, 1, 2, and 4 to the indices 0, 1 and 2, and the third column, 0, 2, and 5 to the indices 0, 1, and 2. Notice that since the second and third column have three different alleles, there must be at least three frequencies in the `freq-list` for markers two and three.

For k-allele genotypes, the situation is very similar, except that the allele lists contains pairs of alleles rather than single alleles:

```
(use-modules (generecon common) (generecon MS genotype))
(define markers
   (let ((positions '(0.2 0.45 0.8))
         (freq-list
            (list '(0.3 0.7) '(0.3 0.3 0.4) '(0.7 0.1 0.2))))
     (make-markers positions freq-list)))
(define reg (region 5e-3 1e-3 markers))

(define genotypes
  (let* ((allele-lists '(((0 . 1) (1 . 1) (2 . 5))
                        ((0 . 0) (4 . 2) (2 . 0))
                        ((1 . 1) (1 . 4) (0 . 2))))
         (mapping (make-index-tables allele-lists)))
    (genotype-list->genotype-list
     reg mapping allele-lists)))
```

The four modules mentioned above can also be used to calculate the frequency-lists needed for specifying markers. For bi-allelic markers, the `calc-frequencies` simply takes the list of allele-lists as argument and returns a list of frequencies for 0 and 1:

```
(use-modules (generecon common) (generecon SNP haplotype))
(define allele-lists '((0 1 0) (1 0 1) (0 1 1)))
(define freq-list (calc-frequencies allele-lists))
```

and similar for the genotype case.

For the k-allelic case, the function also takes, as first argument, a list of (ordered) known alleles for each marker. This permits a marker to have more alleles than what appears in the allele lists (although at frequency 0 according to this frequency calculation). To just get the lists of alleles found in the allele lists, the `MS` modules provide the `collect-alleles-lists` functions. Using this, the calculation of frequencies for the k-allelic case could look like this:

[4] The mapping is always sorted.

```
(use-modules (generecon common) (generecon MS haplotype))
(define allele-lists '((0 1 0) (4 0 1) (2 1 6)))
(define known-alleles (collect-alleles-lists allele-lists))
(define freq-list (calc-frequencies known-alleles allele-lists))
```

Building affected/unaffected sets

Given a set of affected haplotype (or genotype) and a set of unaffected haplotypes (genotypes), we can build a data set from these using the method `affected/unaffected-haplotype-set` (`affected/unaffected-genotype-set`). For haplotypes in our toy example, this could look like this:

```
(define affected
  (list (haplotype reg '(0 0 0))
        (haplotype reg '(0 0 1))
        (haplotype reg '(0 0 2))
        (haplotype reg '(1 0 0))
        (haplotype reg '(1 0 1))
        (haplotype reg '(1 0 2))))
(define unaffected
  (list (haplotype reg '(0 1 0))
        (haplotype reg '(0 1 1))
        (haplotype reg '(0 1 2))
        (haplotype reg '(1 1 0))
        (haplotype reg '(1 1 1))
        (haplotype reg '(1 1 2))))

(define au-haplotype-set
  (affected/unaffected-haplotype-set reg affected unaffected))
```

Here we first define our set of affected and unaffected haplotypes as lists, and then construct the data set using these two lists and the region we analyse. The case for genotypes is similar, except that `affected/unaffected-genotype-set` is used.

For an affected/unaffected data set, the likelihood of the affected individuals are calculated from a coalescent tree with the affected haplotypes as leaves (or one haplotype generated from each genotype in the case of genotype data sets), while the likelihood of the unaffected is calculated from a background probability. The initial coalescent tree can be build using the function `random-tree` that takes as input either an affected/unaffected haplotype data set:

```
(define tree (random-tree au-haplotype-set))
```

or an affected/unaffected genotype set:

```
(define tree (random-tree au-genotype-set))
```

There are two alternative tree builders that can be used: `distance-tree` and `weighted-distance-tree`. The first builds a tree from the sequences

by clustering together haplotypes closely related by the hamming distance

$$\text{hamming}(h, h') = \sum_i \delta_{h_i = h'_i} \tag{1}$$

where $\delta_{h_i = h'_i}$ is the indicator variable for equal alleles at marker i:

$$\delta_{h_i = h'_i} = \begin{cases} 1 & h_i = h'_i \\ 0 & h_i \neq h'_i \end{cases} \tag{2}$$

The tree is build by first calculating all pair-wise distances and then proceed by joining the nearest neighbours, n_1 and n_2, and setting the distance from the resulting node, r and any remaining nodes, n, to be the average distance of n and $n_i, i = 1, 2$.

The function is called with the same parameters as `random-tree`, i.e.

```
(define tree (distance-tree au-haplotype-set))
```

for haplotypes and

```
(define tree (distance-tree au-genotype-set))
```

for genotypes.

The weighted version, `weighted-distance-tree`, builds the tree in the same way, but uses a distance function where each marker is weighted with its distance from a given locus, l

$$\text{dist}_l(h, h') = \sum_i w_{l,i} \cdot \delta_{h_i = h'_i} \tag{3}$$

where $w_{l,i} = |l - i|/L$ is the distance from locus l to marker i relative to the total length of the region L. Building a tree using `weighted-distance-tree`, requires the locus l as an extra argument, thus

```
(define tree (weighted-distance-tree au-haplotype-set l))
```

for haplotypes and

```
(define tree (weighted-distance-tree au-genotype-set l))
```

for genotypes.

Alternatively to putting all affected individuals in the coalescent tree, it is sometimes preferable to consider only a sub-set of these as part of a tree; we can split the affected in a ”mutation cluster” considered in the tree, and a ”null cluster” considered as part of the background distribution of haplotypes and thus treated as the unaffected are treated (except that haplotypes can be moved between the two clusters during the MCMC). To build a data set that splits the affected in the two clusters, we can use the function `cluster`. It takes the data set as its first argument and the size of the mutation cluster as its second. Thus, to use only four of the affected haplotypes in the set created above we would use:

```
(define au-cluster (cluster au-haplotype-set 4))
```

The cluster function works equally well on haplotype and genotype data sets.

The coalescent tree used by the cluster is, by default, one constructed using `random-tree`, but the default behaviour can be changed using a third parameter specifying the tree builder. To use the distance tree method, the option `'distance-tree` is used:

```
(define au-cluster (cluster au-haplotype-set 4 'distance-tree))
```

and to use the weighted distance tree method, `weighted-distance-tree`, followed by the locus to weight the distance function around, `l`:

```
(define au-cluster
   (cluster au-haplotype-set 4 'weighted-distance-tree l))
```

Reading data from files

Input data does not necessarily need to be specified in the Scheme script. Instead, functions from different modules can be used to read in data from simpler data formats.

In the module `(generecon common)` two functions for reading marker positions are provided: `read-positions`—for reading in the positions of the markers—and `read-distances`—for reading in the relative distances between the markers, to be used with `distances→positions`. Both functions read in data from a simple file format of white-space separated numbers and take as their single argument the name of that file, thus to read the list of positions from `positions.txt` we use:

```
(use-modules (generecon common))
(define positions (read-positions "positions.txt"))
```

The four modules `(generecon SNP haplotype)`, `(generecon SNP genotype)`, `(generecon MS haplotype)`, and `(generecon MS genotype)` have functions for reading haplotype/genotype data. The function `read-haplotype-data` from module `(generecon SNP haplotype)` reads in data in the format expected by the `haplotype-list→haplotype-list` function described above, from a file that contains a haplotype per line, represented as a list of space-separated 0s or 1s, or −1 for missing data. Similarly, the function `read-genotype-data` from module `(generecon SNP genotype)` reads in data in the format expected by the `genotype-list→genotype-list` from a file containing a genotype per line as a space-separated list of −1s, 0s, 1s, and 2s.

The function `read-haplotype-data` from module `(generecon MS haplotype)` reads in data in the format expected by the `haplotype-list→haplotype-list` from a file containing a genotype per line as a space-separated list of alleles (any positive number or −1 for missing data). Finally, `read-genotype-data` from module `(generecon MS genotype)` reads in data in the format expected by the `genotype-list→genotype-list` from a file containing a

genotype per line as a space-separated list of alleles (any positive number or -1 for missing data) with two alleles per marker, the first two being the pair for the first marker, the next two the pair for the second marker etc. All four functions take a single argument: the filename for the data to be read.

Each of the four modules also have functions for reading in complete affected/unaffected sets. The function `read-affected/unaffected-data` reads a list of affected and a list of unaffected sequences (from two different files) and return these, together with a list of allele frequencies per marker—calculated from the list of unaffected sequences—in a list where the first element is the list of affected sequences, the second element the list of unaffected sequences, and the third element the list of frequencies:

```
(define data (read-affected/unaffected-data
                 "affected.data" "unaffected.data"))
(define affected    (list-ref data 0))
(define unaffected  (list-ref data 1))
(define frequencies (list-ref data 2))
```

The function `read-positions-affected/unaffected-markers` works similarly, but also reads in the positions and creates the region. To do this it needs—in addition to the file names for reading the positions (third argument), affected sequences (fourth argument) and unaffected sequences (fifth argument)—the recombination rate, κ (first argument) and mutation rate, μ (second argument). It returns a list where the first element is the region, the second element the list of affected haplotypes (or genotypes) and the third element the list of unaffected haplotypes (or genotypes):

```
(use-modules (generecon common)
             (generecon SNP haplotype))

(define kappa 5e-3)
(define mu    1e-6)

(define data (read-positions-affected/unaffected-markers
              kappa mu
              "positions.txt" "affected.txt" "unaffected.txt"))
(define reg                  (list-ref data 0))
(define affected-haplotypes   (list-ref data 1))
(define unaffected-haplotypes (list-ref data 2))
```

A similar function, `read-distances-affected/unaffected-markers`, is provided for reading in a list of distances rather than a list of positions.

Running the analysis

The data structures built so far can be combined into a parameter set for the MCMC algorithms using the `parameter-set` function. This function will accept either an affected/unaffected set together with a tree, or a cluster (that contains both the underlying affected/unaffected set and a tree), together with the other parameters for the MCMC calculation.

The other parameters are the genomic region (the first parameter), the initial disease position (the second parameter), and the initial effective population size (the third parameter). A parameter set for haplotypes and no clustering could look like this:

```
(define h-ps
  (parameter-set reg initial-pos initial-pop-size
                 au-haplotype-set au-h-tree))
```

while a parameter set using clustering could be constructed using:

```
(define h-c-ps
  (parameter-set reg initial-pos initial-pop-size au-cluster))
```

Running the MCMC algorithm

Once a parameter set has been constructed, the MCMC calculation can be started using the `run-mcmc` function. This function is actually a wrapper around a whole set of MCMC algorithms, but the appropriate one is selected based on the parameter set. `run-mcmc` takes two arguments besides the parameter set: a "sampler" and the number of iterations to run. The sampler, described in the next section, specify which parameters should be sampled during the MCMC run, and how often. Running the MCMC for $100,000$ iterations with an empty sampler[5] would look like this:

```
(run-mcmc h-ps (sampler '()) 100000)
```

Setting up samplers

A sampler is built using the function `sampler`, that takes as input a list of "hooks" describing parameters to extract—identified by the name of the hook, the first element in the hook description list—and how often to extract them—specified by the next element. To run an MCMC where we only sample the disease locus and likelihood, each every ten iterations, we could use:

```
(define s (sampler
             (list '(disease-locus 10) '(likelihood 10))))
(run-mcmc h-ps s 100000)
```

The supported sampling hooks are:

likelihood The likelihood of the current parameters.

likelihood-curve the likelihood over the region (the likelihood when varying the locus but keeping all other parameters fixed).

tree-prior The contribution to the likelihood from the tree prior.

tree-likelihood The contribution to the likelihood from the tree.

[5]Running the MCMC with an empty sampler means nothing is extracted form the run and that the run is completely pointless, but for the point of exposition we do it here anyway.

background-likelihood The contribution to the likelihood from the haplotypes without parents in the tree.

disease-locus The current disease locus.

population-size The current effective population size.

coalescent-tree The current coalescent tree.

coalescent-tree-height The height of the current coalescent tree.

coalescent-tree-connectedness The connectedness—number of nodes with the parent-bit set (see Morris *et al.*, 2002), over number of nodes with parent bit not set—of the current coalescent tree.

mutation-cluster The haplotypes or genotypes in the mutation cluster.

If a sampling hook is specified with just the two parameters described earlier, the parameters are written to the screen as the MCMC runs. To output the parameters to files, which is often more desirable, a third parameter can be used, giving the name of the file to write to. To write the likelihood to the file `likelihood.txt` and the locus to the file `locus.txt` we would use:

```
(define s
  (sampler (list '(disease-locus 10 "locus.txt")
                 '(likelihood    10 "likelihood.txt"))))
(run-mcmc h-ps s 100000)
```

MCMC options

There are a few MCMC parameters that can be set for fine-control of GeneRecon; these all have reasonable default values for most situations and need not explicitly be set, but in some cases the default values should be changed, and this can be done using the `set-mcmc-option` function. The syntax of the function is

```
(set-mcmc-option option value)
```

and to, e.g. set the maximal steps the effective population size can change with to 50 one can use:

```
(set-mcmc-option 'max-pop-size-change 50)
```

The supported options are:[6]

max-locus-change Maximum change of the disease locus. A fraction of the total region size the locus is allowed to move in one step; i.e. if the region has size 2 and max-locus-change is 0.5, the locus can move at most 1.0 in each move. The default value is 0.3.

[6]Some of these options require some familiarity with the algorithm, as described in Morris *et al.* 2002.

min-locus-marker-dist The closes the disease locus is allowed to be to a marker. The minimal distance between the disease locus and any marker allowed. If this is zero, the disease locus can be placed on a marker, if it is 0.1 it can never be closer than 0.1 and so on. By default, this is 0; for some datasets, GeneRecon will calculate extremely high likelihoods close to markers, and in such cases this parameter can be used to fix it.

max-allele-freq-change The maximal change of allele frequencies allowed. The maximal amount the frequency for a single allele at a marker can change in a single step. By default, 100%—i.e. the frequencies can change arbitrary (within the bound that they should be frequencies and sum to 1).

min-pop-size Minimal population size. The minimal value the population size is allowed to reach. By default 500.

max-pop-size Maximal population size. The maximal value the population size is allowed to reach. By default $100,000$.

max-pop-size-change Maximal population size change. The maximal amount the population size is allowed to change in a single step. By default 1000.

max-waiting-time-change Maximal effective population size change. The maximal amount the waiting time in a coalescent tree can change. By default 0.4.

mcmc-temperature Temperature for accepting more proposed changes. The acceptance probability is normally $\exp(L'-L)$ where L' is the log-likelihood of the proposed locus and L the log-likelihood of the current position. With a temperature, t, it will instead be $\exp((L'-L)/t)$. To get a "flatter" curve, use a $t > 1$. By default it is 1.0, corresponding to no temperature.

It is also some times desirable to control the weighting of the different types of changes. Reasonable defaults are used if the weighting is not explicitly set, but changing the weights might sometime improve mixing. The change weighting can be explicitly set using the `set-mcmc-weight`function. The interface is the same as for setting the MCMC options:

```
(set-mcmc-weight change weight)
```

e.g.

```
(set-mcmc-weight 'locus-change 10) ; set locus change to 10
                                   ; rather than 1
```

and the supported changes are:

locus-change Weight for changing the disease locus. The default value is 1.

allele-frequency-change Weight for changing the allele frequency of markers. The default value is the number of markers.

population-size-change Weight for changing the effective population size. The default value is 1.

missing-allele-change Weight for changing the allele at a site where the allele is unknown. The default value is the number of such sites.

genotype-phase-change Weight for changing the phase of a genotype. The default value is the number of heterozygote sites.

branch-position-change Weight for changing the tree topology by moving a branch. Default is the number of branches: $2(n-1)$ where n is the number of leaves in the coalescent tree.

parent-bit-change Weight for changing the parent bit. Default is the number of branches: $2(n-1)$ where n is the number of leaves in the coalescent tree.

event-ordering-change Weight for changing the ordering of events. Default is $n-2$ where n is the number of leaves in the coalescent tree.

waiting-time-change Weight for changing a waiting time. Default is the number of waiting times: $n-1$ where n is the number of leaves in the coalescent tree.

clustering-change Weight for changing the mutation cluster. Default is the size of the cluster.

Analysing the results

GeneRecon does not, in itself, provide any support for analysing the results of running the MCMC. Instead we can use statistical software such as e.g. the R package (see http://www.r-project.org) to visualise the output from GeneRecon computations. If we have run a chain where we sample the likelihood and locus, as in the example above, we can for example plot the likelihood as a function of the number of iterations as:

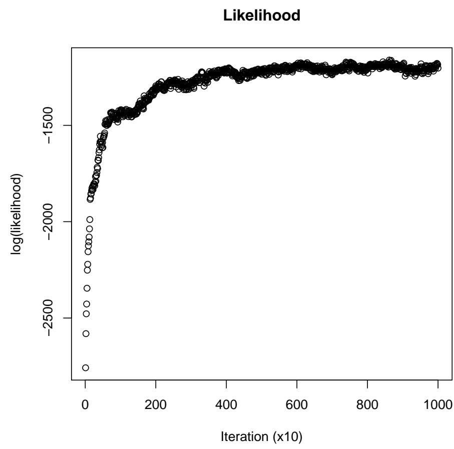
```
> likelihood <- scan(file="likelihood.data")
> plot(likelihood)
```

The result is shown in Figure 1(a). Plotting the disease position as a function of the number of iterations:

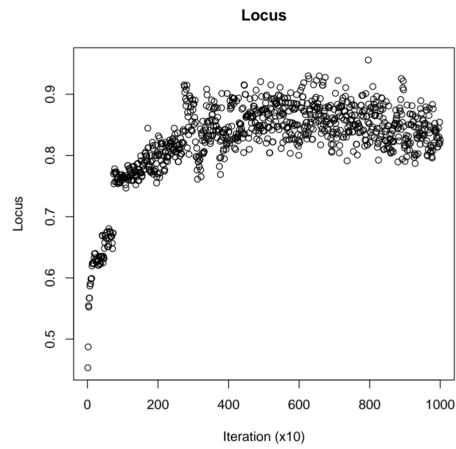
```
> locus <- scan(file="locus.data")
> plot(locus)
```

The result is shown in Figure 1(b).

The probability of a disease causing mutation at a given position is proportional to the number of times the MCMC has visited that position after an initial burn-in period. The posterior distribution is obtained by binning the sampled disease positions. Several bin sizes should be tried when displaying the data,

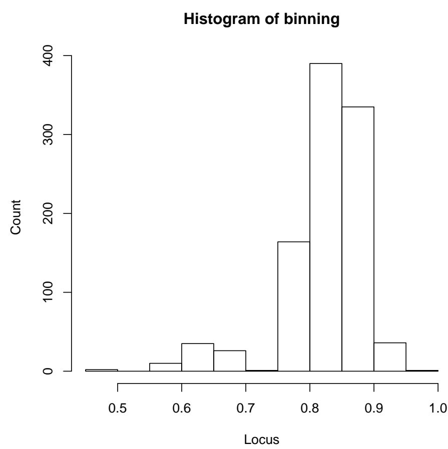


(a) The likelihood as a function of the number of iterations.

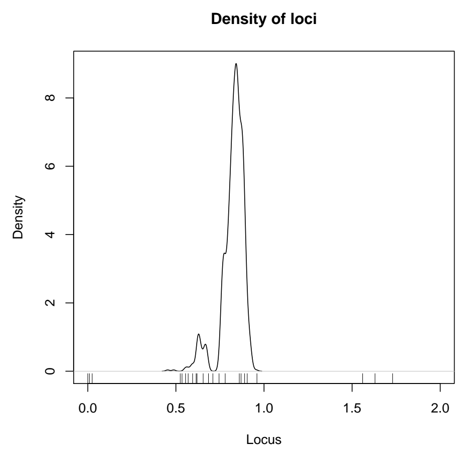


(b) The disease position as a function of the number of iterations.

Figure 1: The sampled values from an MCMC run.



(a) Histogram of the counts of binned positions.



(b) The posterior density of the disease locus.

Figure 2: Summaries of the sampled disease locus parameter.

in order to avoid binning-artifacts. Using R we can bin the values using the function `hist` (for histogram), that both performs the binning and plots a histogram of the bin counts

```
> hist(locus)
```

see Figure 2(a). Alternatively we can plot the posterior density using the `density` function

```
> plot(density(locus), xlim=c(0,2))
> positions <- c(0, 0.009, 0.0248, 0.5248, 0.5348, 0.5548,
+                0.5698, 0.5948, 0.6148, 0.6198, 0.6548,
+                0.6848, 0.7098, 0.7448, 0.7798, 0.8598,
+                0.8698, 0.8898, 0.9048, 0.9598, 1.5598,
+                1.6298, 1.7298)
> rug(positions)
```

see Figure 2(b). Here we also indicated the positions of the markers using the `rug` function.

The Markov Chain only samples from the posterior distribution when mixing properly. Mixing ensures a thorough search in the parameter space and minimizes the likelihood of being trapped in local maxima. The mixing behavior of the Markov Chain can currently be estimated by visual inspection of the output after a burn-in period.

To ensure that the results of running GeneRecon are correct, it is usually necessary to run several chains and verify that they converge to the same posterior distribution—if the result of running the MCMC is random, the resulting posterior distributions should be random, whereas if the resulting posteriors agree, they are likely to represent a true signal in the data.

Contact

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