Simulation and Inference with BORIS

Simon Firestone

24/06/2019

Contents

Outbreak simulation with BORIS

In this example, we will:

1. Prepare the inputs for simulation
2. Simulate an outbreak with corresponding genomic data

Preparing inputs for BORIS::sim()

The **farm covariate data** should be inputed as a dataframe as specified in the help for `data(sim.epi.input)`. In this example, at the start of the outbreak there is just one latently exposed individual (with `status = 2`), soon to become infectious. The rest of the individuals are susceptible (with `status = 1`). Outbreaks can be forward simulated based on data available at detection when some individuals are known to be infectious (`status = 3`) or even recovered (`status = 4`).

An **animal movement/contact-tracing dataset** is required to be inputted as a dataframe as specified in the help for `data(sim.moves.input)`, irrespective of whether it is used (i.e. whether the option `parsAux$opt_mov` = 1 or = 0. This extension to the model is still under development. If `parsAux$opt_mov` is set to 0, you can just use the example dataset `data(sim.moves.input)`.

```
library(BORIS)

data(sim.epi.input)

head(sim.epi.input)
#>   k  coor_x   coor_y   t_e   t_i   t_r ftype0 ftype1 ftype2 herdn status
#> 1 0 144.901 -36.4348 0e+00 9e+06 9e+06      0      1      0  3209      2
#> 2 1 144.924 -36.4288 9e+06 9e+06 9e+06      0      0      1  1705      1
#> 3 2 144.924 -36.4288 9e+06 9e+06 9e+06      1      0      0   133      1
#> 4 3 144.903 -36.4205 9e+06 9e+06 9e+06      1      0      0   213      1
#> 5 4 144.907 -36.4179 9e+06 9e+06 9e+06      1      0      0   127      1
#> 6 5 144.898 -36.4310 9e+06 9e+06 9e+06      0      0      1     6      1


data(sim.moves.input)

head(sim.moves.input)
#>   from_k to_k t_m
#> 1      0    1   9
#> 2      0    2   9
#> 3      0    3   9
#> 4      0    4   9
#> 5      0    5   9
#> 6      0    6   9
```

Next, enter the key simulation parameters as follows. Details on the parameters can be found in the help for `data(sim.param.key)`.

```
para.key <- data.frame('alpha' = 4e-4,
                       'beta' = 0.2,
                       'mu_1' = 2e-05,
                       'mu_2' = 2e-06,
                       'a' = 3.0,
                       'b' = 2.5,
                       'c' = 21.0,
                       'd' = 4.0,
                       'k_1' = 1.7,
                       'p_ber' = 0.1,
                       'phi_inf1' = 3,
                       'phi_inf2' = 1.5,
                       'rho_susc1' = 0.4,
                       'rho_susc2' = 2,
                       'nu_inf' = 0.2,
                       'tau_susc'= 0.1,
                       'beta_m'= 1.0)
```

Finally, enter some other important settings. Details on these settings can be found in the help for `data(sim.param.aux)`.

```
pars.aux <- data.frame('n' = 100,
                       'seed' = 2468,
                       'n_base' = 7667,
                       'n_seq' = 5,
                       't_max' = 100,
                       'unassigned_time' =  9e+6,
                       'sample_range' = 10,
                       'partial_seq_out' = 0,
                       'n_base_part' = 1000,
                       'n_index' = 1,
                       'coord_type' = 'longlat',
                       'kernel_type' = 'power_law',
                       'latent_type' = 'gamma',
                       'opt_k80' = 1,
                       'opt_betaij' = 1,
                       'opt_movt' = 0,
                       'n_mov' = 60,
                       stringsAsFactors = F)
```

Simulate an outbreak with corresponding genomic data

Then you are ready to run the simulation as follows:

```
sim.out<-sim(epi.inputs = sim.epi.input,
             moves.inputs = sim.moves.input,
             parsKey = para.key,
             parsAux = pars.aux,
             inputPath = "./inputs",
             outputPath = "./outputs")
#> Input path: ./inputs\
#> Output path: ./outputs\
#>
#> Outbreak simulated successfully.
#>
```

This function:

1. Creates input and output directories at the locations specified, or deletes everything existing in those directories.
2. Checks the ranges of key inputs to see that they are within reasonable limits.
3. Writes a set of intput files to the `inputPath`.
4. Runs the simulation storing key outputs into an object `sim.out` and also writing output files to the `outputPath`.

Inspecting some of these outputs

The epidemiological dataset with simulated values for timing of exposure, onset and end of the infectious period for each individual.

```
sim.out$epi.sim[1:6,]
```

```
#>      k  coor_x   coor_y    t_e    t_i     t_r ftype0 ftype1 ftype2 herdn
#> [1,] 0 144.901 -36.4348 0.0000 1.3422  7.6192      0      1      0  3209
#> [2,] 1 144.924 -36.4288 3.9785 5.7881 12.7286      0      0      1  1705
#> [3,] 2 144.924 -36.4288 1.4037 1.7676 11.1723      1      0      0   133
#> [4,] 3 144.903 -36.4205 5.3555 9.1884 13.0687      1      0      0   213
#> [5,] 4 144.907 -36.4179 2.2866 4.6083  9.2533      1      0      0   127
#> [6,] 5 144.898 -36.4310 1.6297 2.6918  4.2218      0      0      1     6
```

The unique ordered identifier (i.e. k) of the simulated infected source of each individual. Those with a simulated source = 9999 have been infected from an external source.

```
head(sim.out$infected_source, 14)
#>  [1] 9999    0    0    2    0    0    0    0    0    0   23   28   63
```

Any with a simulated source = -99 have not been infected in this simulation.

```
which(sim.out$infected_source == -99)
#> [1] 31
```

The percentage of individuals in the population that were infected can be calculated as:

```
inf <- which(sim.out$infected_source != -99)
length(inf)/length(sim.out$infected_source)
#> [1] 0.99
```

The percentage of *total* individuals that were simulated to have actually been sampled is returned as:

```
sim.out$sampled_perct
#> [1] 0.8888889
```

The simulated timing of sampling per individual. Those with the `unassigned_time` of 9e6 have not been sampled.

```
head(floor(sim.out$t_sample), 20)
#>  [1]       2      10       9       5 9000000 9000000       7       5
#>  [9]       7      10 9000000      16 9000000      22      49      52
#> [17]      17 9000000 9000000      19
```

The percentage of *infected* individuals that were simulated to have actually been sampled can be calculated as follows:

```
length(which(sim.out$t_sample[inf] != 9e6))/length(inf)
#> [1] 0.8888889
```

The sequence data itself can be found in the files `subject_x_nt.txt` with the `x` repesenting the identified (`k`) for each individual. The nucleotides bases are represented as 1=A, 2=G, 3=T, 4=C. If an individual has a sequence simulated at more than one time-point (i.e. when it was exposed, then subsequently at the time when it infected another individual(s), and the possibly again if sampled) then each simulated sequence is written to a new line, comma-separated. The times corresponding to these sequences are stored in the files `subject_x_t_nt.txt`.

To write a simulated set of sequences for an individual out to FASTA using the `ape` library:

```
library(ape)

#read in the sequences from individual k=0
fn <- paste0(sim.out$outputPath, "subject_0_nt.txt")
seqs1 <- as.matrix(read.csv(fn, header=F))

#the sequences for this individul are stored in 19 rows
# nucleotides are coded as 1=A, 2=G, 3=T, 4=C
dim(seqs1)
#> [1]   19 7667

#convert the sequences to their nucleotide base letters
seqs1[seqs1==1]<-"a"
seqs1[seqs1==2]<-"g"
seqs1[seqs1==3]<-"t"
seqs1[seqs1==4]<-"c"

#name the rows, which become the tip labels in the fasta file.
#for this we will import the timings associated with each sequence
fn <- paste0(sim.out$outputPath, "subject_0_t_nt.txt")
seqs1_t <- as.numeric(unlist(read.table(fn)))

rownames(seqs1) <- paste0("k0_", round(seqs1_t,3))

#write these out to a fasta file
library(ape)
write.dna(seqs1, file='seqs1.fasta', format = 'fasta', nbcol = -1)

#inspect snps in these data
seq.tab <- apply(seqs1, MARGIN = 2, table)
snp.v<-sapply(seq.tab, length)
snps<-as.numeric(which(snp.v>1))
seqs1[1:8,snps]
#>          V3715 V7484
#> k0_0     "a"   "a"
#> k0_1.402 "g"   "g"
#> k0_1.404 "g"   "g"
#> k0_1.63  "g"   "g"
#> k0_1.662 "g"   "g"
#> k0_1.703 "g"   "g"
#> k0_1.778 "g"   "g"
#> k0_2.003 "g"   "g"
```

There are two similar point mutations from A -> G at roughly the same time in different locations in the genome.

Outbreak reconstruction and inference with BORIS

In this example, we will:

1. Prepare the inputs for outbreak reconstruction
2. Run the reconstruction and inference
3. Inspect diagnostics for the MCMC chain
4. Produce some relevant outputs

Preparing inputs for BORIS::infer()

The following inputs are required to run an outbreak reconstruction and inference:

1. `covariates`: A data.frame including farm covariate data inputed as specified in the help for `data(infer.epi.input)`.
2. `movements`: A data.frame including animal movement or contact-tracing data inputed as specified in the help for `data(infer.moves.input)`.
3. `parsAux`: A data.frame including the parameters for MCMC implementation.
4. `keyInits`: A data.frame including key initial values for parameters. 1. `priors`: A data.frame including prior settings for paramters.
5. `scalingFactors`: A data.frame including scaling factors for parameters.
6. `seed`: An integer seed for for the random number generator.
7. `accTable`: A data.frame including known sources for each individual for simulated. If unknown then enter `9999` for each source.
8. `genomic data`: A .fasta file stored on the `dnaPath`.
9. `t_sampling`: A data.frame of the times that the specimens were collected that were sequenced to provide the genomic data.

In the following example, we will wrangle the data from the previous simulation into the required format, to show the process. The code can also be adapted to empirical inputs from an observed outbreak.

```r
library(BORIS)

#use the epidemiological data from the previous simulation run
epi.data<-as.data.frame(sim.out$epi.sim)

#produce the required temporal fields. Here we are assuming that
#onset of infectiousness typically occurs 24 hours before the
#first clinical signs
epi.data$t_o <- floor(epi.data$t_i) + 1
epi.data$t_s <- round(sim.out$t_sample, 0)
epi.data$t_r <- round(epi.data$t_r, 0)

#wrangle the farm type variables into the required format
epi.data$ftype <- NA
epi.data$ftype[epi.data$ftype0 == 1] <- 0
epi.data$ftype[epi.data$ftype1 == 1] <- 1
epi.data$ftype[epi.data$ftype2 == 1] <- 2

#keep just the desired columns, re-ordered as required
epi.data<-epi.data[,c('k','coor_x','coor_y','t_o','t_s','t_r','ftype','herdn')]
head(epi.data)
#>   k  coor_x   coor_y t_o     t_s t_r ftype herdn
#> 1 0 144.901 -36.4348   2       3   8     1  3209
#> 2 1 144.924 -36.4288   6      11  13     2  1705
#> 3 2 144.924 -36.4288   2       9  11     0   133
#> 4 3 144.903 -36.4205  10       5  13     0   213
#> 5 4 144.907 -36.4179   5 9000000   9     0   127
#> 6 5 144.898 -36.4310   3 9000000   4     2     6

#use the movement data from previously
moves.inputs<-sim.out$moves.inputs
head(moves.inputs)
#>   from_k to_k t_m
#> 1      0    1   9
#> 2      0    2   9
#> 3      0    3   9
#> 4      0    4   9
#> 5      0    5   9
#> 6      0    6   9
```

The options for configuring the MCMC implementation are setup as follows. For details see the help for `data(infer.param.aux)`.

```
pars.aux <- data.frame('n' = nrow(epi.data),
                       'kernel_type' = 'power_law',
                       'coord_type' = 'longlat',
                       't_max' = 100,
                       'unassigned_time' =  9e+6,
                       'processes' = 1,
                       'n_seq' = 5,
                       'n_base' = 7667,
                       'n_iterations' = 1e5,
                       'n_frequ' = 10,
                       'n_output_source' = 1,
                       'n_output_gm' = 1000,
                       'n_cout' = 10,
                       'opt_latgamma' = 1,
                       'opt_k80' = 1,
                       'opt_betaij' = 1,
                       'opt_ti_update' = 1,
                       'opt_movt' = 0,
                       stringsAsFactors = F)
```

The initial values for key inferred parameters are setup as follows. For details see the help for `data(infer.param.key)`.

```
para.key.inits <- data.frame('alpha' = 2e-4,
                             'beta' = 0.1,
                             'lat_mu' = 5,
                             'lat_var' = 1,
                             'c' = 10,
                             'd' = 3,
                             'k_1' = 3,
                             'mu_1' = 3e-05,
                             'mu_2' = 1e-06,
                             'p_ber' = 0.2,
                             'phi_inf1' = 1,
                             'phi_inf2' = 1,
                             'rho_susc1' = 1,
                             'rho_susc2' = 1,
                             'nu_inf' = 0.2,
                             'tau_susc'= 0.1,
                             'beta_m'= 0.5)
```

The prior information for inferred parameters are setup as follows. For details see the help for `data(infer.param.priors)`.

```
para.priors <- data.frame('t_range' = 7,
                          't_back' = 21,
                          't_bound_hi' = 10,
                          'rate_exp_prior' = 0.001,
                          'ind_n_base_part' = 0,
                          'n_base_part' = 1000,
                          'alpha_hi' = 0.1,
                          'beta_hi' = 50,
                          'mu_lat_hi' = 50,
                          'var_lat_lo' = 0.1,
                          'var_lat_hi' = 50,
                          'c_hi' = 100,
                          'd_hi' = 100,
                          'k_1_hi' = 100,
                          'mu_1_hi' = 0.1,
                          'mu_2_hi' = 0.1,
                          'p_ber_hi' = 1.0,
                          'phi_inf1_hi' = 500,
                          'phi_inf2_hi' = 500,
                          'rho_susc1_hi' = 500,
                          'rho_susc2_hi' = 500,
                          'nu_inf_lo' = 0,
                          'nu_inf_hi' = 1,
                          'tau_susc_lo' = 0,
                          'tau_susc_hi' = 1,
                          'beta_m_hi' = 5,
                          'trace_window' = 20)
```

The scaling factors (otherwise know as operators or proposal distances) for inferred parameters are setup as follows. For details see the help for `data(infer.param.sf)`. The default is 1. Increase the scaling factor to increase the proposal distance, for instance if the parameter is wandering about and accepting too often (i.e. if the proposal distance is too low). Decrease the scaling factor to decrease the proposal distance, for instance if the parameter is not being accepted often enough (i.e. if the proposal distance is too high and its trace looks like a 'Manhattan skyline'). Set the scaling factor to zero to fix a parameter at its initialising value (so that it doesn't update, such as for `beta_m` when `opt_mov = 0`).

```
para.sf <- data.frame('alpha_sf' = 0.001,
                      'beta_sf' = 0.5,
                      'mu_lat_sf'   = 1.25,
                      'var_lat_sf' = 1.75,
                      'c_sf' = 1.25,
                      'd_sf' = 0.75,
                      'k_1_sf' = 1,
                      'mu_1_sf' = 2.5e-5,
                      'mu_2_sf' = 2.5e-6,
                      'p_ber_sf' = 0.02,
                      'phi_inf1_sf' = 1.75,
                      'phi_inf2_sf' = 1.5,
                      'rho_susc1_sf' = 1,
                      'rho_susc2_sf' = 1.25,
                      'nu_inf_sf' = 0.25,
                      'tau_susc_sf' = 0.25,
                      'beta_m_sf' = 1)
```

Preparing the genomic data for BORIS::infer()

Genomic data can simply be written to the `dnaPath` as a .fasta file prepared with the `ape` library or other phylogenetic software.

Here, we use the simulated outbreak data to show how to utilise the genomes produced earlier.

First, use the sampling times data from the simulation. Those not sampled have the unassigned value.

Then identify and store the data from sequences at the time of sampling on each sampled farm. Other sequences are from the times when transmission events occurred and are typically unobserved.

```
t_sample <- sim.out$t_sample

#which individuals are missing genomes and which were sampled
no.genome <- which(t_sample == pars.aux$unassigned_time)
genome.sampled <- which(t_sample != pars.aux$unassigned_time)

#the sampling times of those with genomes
round(t_sample[-no.genome], 3)
#>  [1]  2.748 10.866  9.423  5.356  7.073  5.958  7.737 10.170 16.703 22.219
#> [11] 49.489 52.991 17.821 19.053  6.741 21.356  5.219  9.517 16.037 11.436
#> [21] 11.037  8.715  4.607 20.537 13.633 17.708 19.584 18.884 22.250 20.003
#> [31]  8.261 36.828 18.566 17.497 20.089 13.276 18.140  7.937 11.192 43.382
#> [41] 16.451 15.005  2.650 47.150 20.505 34.554 44.679 10.806 25.879 19.030
#> [51] 14.766 13.636 24.822 12.700 18.873 18.253 11.424 16.177 18.945 31.042
#> [61] 18.328 17.270  8.058 13.713 15.175 27.699 39.682 19.054  8.078 11.967
#> [71] 13.490 31.187 21.807 12.569 12.744 16.485 16.901  9.001 14.714 18.333
#> [81] 15.765 11.623 21.476 38.838  3.991 18.543 18.800 27.906

#setup a matrix to hold 1 sequence per individual or missing data
seq_mat<-matrix(nrow=nrow(epi.data), ncol=pars.aux$n_base)

#identify and store the data from sequences at the time of sampling
#the next step takes approximately 30 seconds to run
for(i in 1:nrow(epi.data)){
  k <- i-1
  if(i %in% no.genome){
    #these are not used, given t.sample for these = the unassigned time
    #n denotes any base (in the IUPAC system)
    seq_mat[i,]<-rep("n", pars.aux$n_base)
  }else{
    #identify the closest corresponding sample to the time of
    #sampling for each sampled individual
    fn<-paste0(sim.out$outputPath,"subject_",k,"_t_nt.txt")
    ts<-as.numeric(unlist(read.csv(fn, header=F)))
    tsamp <- which.min(abs(ts - t_sample[i]))
    #read in the corresponding nucleotide sequence
    fn<-paste0(sim.out$outputPath,"subject_",k,"_nt.txt")
    nts<-read.csv(fn, header = F, stringsAsFactors = F)
    #store the closest corresponding sample for this individual
    seq_mat[i,]<-as.numeric(nts[tsamp,])
  }
  if((i%%2) == 0) {cat(".",sep=""); flush.console()}  #
}
```

```
#> ..............................................
```

Finalise the sequence data and write it to a *.fasta file.

```
#convert the sequence data to their nucleotide base letters
seq_mat[seq_mat==1]<-"a"
seq_mat[seq_mat==2]<-"g"
seq_mat[seq_mat==3]<-"t"
seq_mat[seq_mat==4]<-"c"

#write row.names to the sequence matrix object, including the 'k'
#and time of sampling. These will be used as tip labels
row.names(seq_mat)<- paste0(epi.data$k, "_", epi.data$t_s)


if(!dir.exists("gen_inputs")) {
      dir.create("gen_inputs")
    }


#write these out to a fasta file in the dnaPath directory
library(ape)
write.dna(seq_mat, file='./gen_inputs/seqs.fasta', format = 'fasta', nbcol = -1)
```

Running the reconstruction and inference

Here we just run one MCMC chain for a very small number of iterations (for demonstration purposes only), whereas in practice multiple chains must be run and inspected appropriately (i.e. for burn-in, convergence and serial autocorrelation) before any inferences are made on the transmission tree or any inferred parameters.

As the accuracy table (accTable) for evaluating how well the infer is performing, we will use the known infected sources from our prior simulation that generated these data. When inferring for an actual outbreak, it will be unknown which is the true source for each infected individual. In that case, enter `9999` for all infected sources.

```
pars.aux$n_iterations = 1000

infer.out<-infer(covariates = epi.data,
                 moves.inputs = moves.inputs,
                 parsAux = pars.aux,
                 keyInits = para.key.inits,
                 priors = para.priors,
                 scalingFactors = para.sf,
                 seed = 1,
                 accTable = sim.out$infected_source,
                 t.sample = sim.out$t_sample,
                 inputPath = "./inputs",
                 outputPath = "./outputs",
                 dnaPath = "./gen_inputs")
```

Multiple chains can be run:

1. on a single computer: by saving the output from the first run to a new directory, because when `infer()` starts it deletes everything in the input and output directories. Then running again with a different seed.
2. on a SLURM cluster or similar: adapt the following code, then use `.seed` as the `seed` argument for the function `infer()`:

```
task.id <- as.numeric(Sys.getenv('SLURM_ARRAY_TASK_ID'))
task.id

.seed <- c(1, 102, 1003, 10004)[task.id]
.seed
```

Diagnostics and outputs from multiple MCMC chains

Following the MCMC run, it is important to check and exclude burn-in appropriately, and for appropriate mixing and convergence of multiple chains. `Tracer` (http://beast.community/tracer) is a great tool for rapidly evaluating the parameter output of chains, as stored in the file `parameters_current.log`. It allows rapid assessment of convergence of each parameter across multiple chains, effective sample sizes, amount of autocorrelation, and also reading off summary posterior estimates.

Here we will implement some similar analyses in `R`, then continue with code for extracting posterior estimates of the transmission tree and further inferred parameters for each individual.

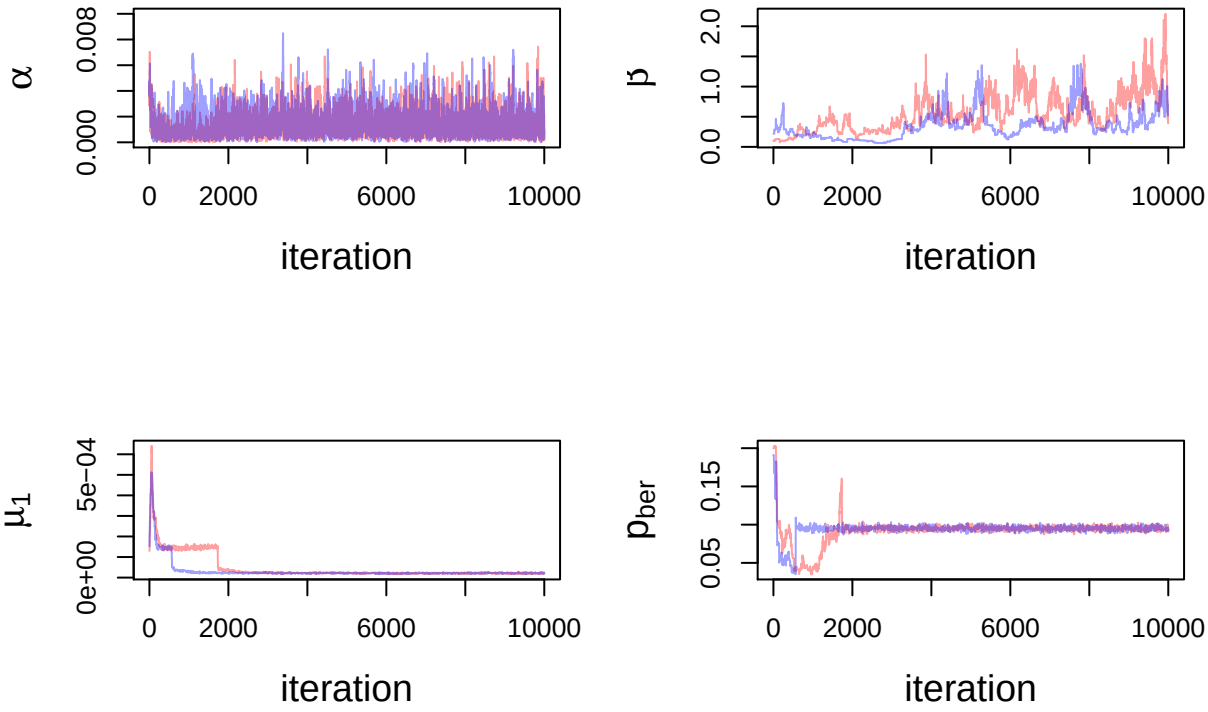
In the last example, we just ran a single chain. The following uses example data from 2 chains of the same inference, each of length 10,000 MCMC cycles. Again, this is for illustative purposes only, in practice hundreds of thousands to millions of iterations are typically required as these are complex inferences.

Inspecting the chains

Inspect the start of the chains to evaluate if and when convergence has occurred. Therefore, where to set burn-in, and if need be the amount of thinning.

Pre-prepared parameter data for each chain have been inputted by reading in the tab delimited `parameters_current.log` files produced by independantly seeded `infer()` runs on the same input data.

```
library(BORIS)
data(paras1)
data(paras2)

burnin=0
thin = 1
end=1e04

x<-seq(burnin+1,end,thin)


par(mfrow=c(2,2))
plot(x, paras1$alpha[x], type='l', col='#ff000060',
     ylim=c(0,0.01),
     xlab='iteration', ylab=expression(alpha), cex.lab=1.4)
lines(x, paras2$alpha[x], col='#0000ff60')

plot(x, paras1$beta[x], type='l', col='#ff000060',
     xlab='iteration', ylab=expression(beta),
     cex.lab=1.4)
lines(x, paras2$beta[x], col='#0000ff60')

plot(x, paras1$mu_1[x], type='l', col='#ff000060',
     xlab='iteration', ylab=expression(mu[1]),
     cex.lab=1.4)
lines(x, paras2$mu_1[x], col='#0000ff60')

plot(x, paras1$p_ber[x], type='l', col='#ff000060',
     xlab='iteration', ylab=expression(p[ber]),
     cex.lab=1.4)
lines(x, paras2$p_ber[x], col='#0000ff60')
```



In this example, the two chains for `alpha` appear to converge very quickly then mix well with most posterior samples between 0 and 0.02.

The chains for `beta` are looking like they are soon to converge, however there are two patterns here that indicate **poor mixing** to be aware of if they persist longer into traces. In the first 2000 iterations, both traces for `beta` appear to have a rate of acceptance that is too low. A typical cause of such a pattern is that the scaling factor (or proposal distance) is too large. In this case, however, this is unlikely to be the cause as this mixing pattern self-corrects. Between iterations 4000 and 10000 the traces wander about (another poor mixing pattern), and it appears the rate of acceptance is now too high. In this example, this is just because we haven't run enough iterations, and the chains can still be considered to be burning-in. Possible solutions to such poor mixing patterns could include running the MCMC for more iterations or altering the respective scaling factor (proposal distance).

The traces for `mu_1` initially converge before the 500th iteration, then there is a sizeable jump to a lower posterior estimate in one chain, followed by the other, with convergence achieved after 2000 iterations.

The traces for the parameter `p_ber` also converge by around 2000 iterations.

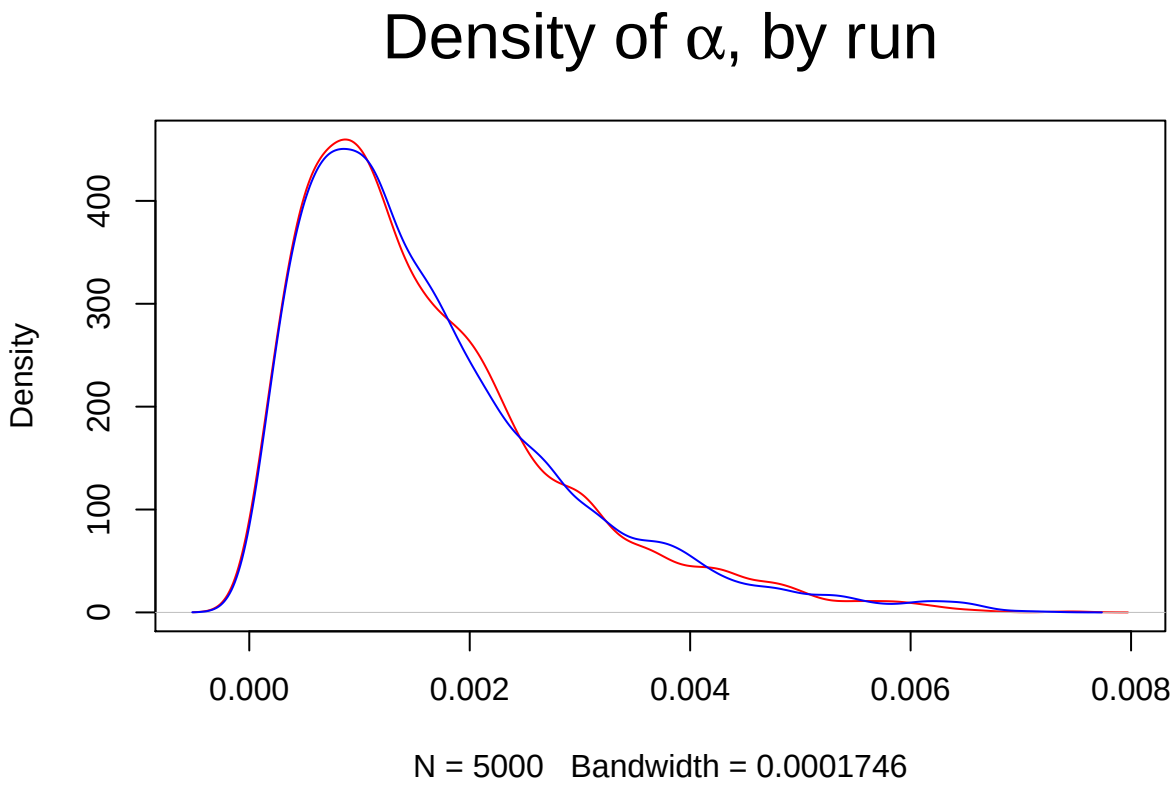
In this example, we will use a burn-in of 5000 iterations, and given little indication of serial autocorrelation will not thin the estimates.

The posterior parameter estimates can thus be derived as follows:

```
burnin=5000
thin = 1
end=1e04
x<-seq(burnin+1,end,thin)

paras<-rbind(paras1[x,], paras2[x,])

#posterior estimates for alpha
round(quantile(paras$alpha, probs=c(0.5, 0.025, 0.975)), 4)
#>    50%   2.5%  97.5%
#> 0.0013 0.0002 0.0047

#posterior estimates for beta
round(quantile(paras$beta, probs=c(0.5, 0.025, 0.975)), 3)
#>   50%  2.5% 97.5%
#> 0.506 0.204 1.311

#posterior estimates for c
round(quantile(paras$c, probs=c(0.5, 0.025, 0.975)), 2)
#>   50%  2.5% 97.5%
#>  9.69  8.54 10.87

#posterior estimates for d
round(quantile(paras$d, probs=c(0.5, 0.025, 0.975)), 3)
#>   50%  2.5% 97.5%
#> 2.172 1.760 2.648

#posterior estimates for mu_1
quantile(paras$mu_1, probs=c(0.5, 0.025, 0.975))
#>         50%        2.5%       97.5%
#> 2.12378e-05 1.77687e-05 2.50796e-05

#posterior estimates for mu_2
quantile(paras$mu_2, probs=c(0.5, 0.025, 0.975))
#>          50%         2.5%        97.5%
#> 2.535750e-06 1.788056e-06 3.547930e-06
```

Density plots produced for each parameter of interest:

```
plot(density(paras1$alpha[x]), col='red', cex.main=2,
  main=expression(paste(plain("Density of "),alpha,plain(", by run"))))
lines(density(paras2$alpha[x]), col='blue')
```



Inferred infected sources

Pre-prepared infected source data for each chain have been inputted by reading in the comma separated `infected_source_current.csv` files produced by independantly seeded `infer()` runs on the same input data.

Let's inspect the `head` and `tail` of each of these datasets:

```
data(inf1)
head(inf1[,1:10])
#>      1 2    3 4 5 6    7 8 9 10
#> 1 9999 6 9999 0 0 0 9999 2 9 26
#> 2 9999 6 9999 0 0 0 9999 2 9 26
#> 3 9999 6 9999 0 0 0 9999 2 9 26
#> 4 9999 6 9999 0 0 0 9999 2 9 26
#> 5 9999 6 9999 0 0 0 9999 2 9 26
#> 6 9999 6 9999 0 0 0 9999 2 9 26
tail(inf1[,1:10])
#>       1 2    3 4 5 6 7 8 9 10
#> 9995  2 2 9999 5 2 2 2 1 1  0
#> 9996  2 2 9999 5 2 2 2 1 0  0
#> 9997  2 2 9999 5 2 2 2 1 0  0
#> 9998  2 2 9999 5 2 2 2 1 0  0
#> 9999  2 2 9999 5 2 2 2 1 0  0
#> 10000 2 2 9999 5 2 2 2 1 0  0

data(inf2)
head(inf2[,1:10])
#>      1 2    3  4 5 6    7 8 9 10
#> 1 9999 6 9999 95 0 2 9999 2 9 26
#> 2 9999 6 9999 95 0 2 9999 2 9 26
#> 3 9999 6 9999 95 0 2 9999 2 9 26
#> 4 9999 6 9999 95 0 2 9999 2 9 26
#> 5 9999 6 9999 95 0 2 9999 2 9 26
#> 6 9999 6 9999 95 0 2 9999 2 9 26
tail(inf2[,1:10])
#>       1 2    3  4 5 6 7 8  9 10
#> 9995  2 2 9999 10 2 0 0 6 50 10
#> 9996  2 2 9999 10 1 0 0 6 50 10
#> 9997  2 2 9999 10 1 0 0 6 50  5
#> 9998  2 2 9999 10 1 0 0 6 50  5
#> 9999  2 2 9999 10 1 0 0 6 50  5
#> 10000 2 2 9999 10 1 0 0 6 50  5
```

Using the same burn-in as previously decided:

```
burnin=5000
thin = 1
end=1e04
x<-seq(burnin+1,end,thin)

#create lists that store for each individual:
#- a frequence table of iterations featuring each infected source (inf.l)
#- the source with the highest modal posterior support (inf.mode.l)
#- the posterior support for each infected source (inf.support.l)
inf.l<-list()
inf.mode.l<-numeric()
inf.support.l<-numeric()
for(i in 1:nrow(epi.data)){
  inf.l[[i]]<-table(c(inf1[x,i], inf2[x,i]))
  inf.mode.l[i]<-as.numeric(attr(which.max(inf.l[[i]]), 'names'))
  inf.support.l[i]<-
    as.numeric(inf.l[[i]][which.max(inf.l[[i]])]/sum(inf.l[[i]]))
}

#inspect these outputs for individual k=6
inf.l[[2]]
#>
#>    0    2    5    6    8   26   45
#> 2737 6489  611  110   32    5   16
inf.mode.l[2]
#> [1] 2
inf.support.l[2]
#> [1] 0.6489
```

The highest modal posterior support can be combined into an edgelist:

```
el<-data.frame(to=1:nrow(epi.data), from=inf.mode.l)

head(el)
#>   to from
#> 1  1    2
#> 2  2    2
#> 3  3 9999
#> 4  4    0
#> 5  5    0
#> 6  6    2
```

And compared to the inputted `accTable`:

```
#from the simulation
accTable <- sim.out$infected_source

#which are correctly inferred
acc <- ifelse(accTable == el$from, 1, 0)
acc
#>   [1] 0 0 0 0 1 0 1 0 1 1 1 1 0 1 1 1 1 0 0 1 1 0 1 1 1 1 1 0 1 1 0 1 0 0 1
#>  [36] 1 0 1 1 1 1 1 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 0 1 0 1 1 1
#>  [71] 0 1 1 1 1 1 0 1 1 1 1 1 0 0 0 1 0 1 0 1 0 0 1 1 0 1 1 1 0 1

#number correct
sum(acc)
#> [1] 65

round(sum(acc)/length(el$from),2)
#> [1] 0.65
```

The accuracy for those with consensus support or better can be estimated as follows:

```
nsup<-length(which(inf.support.l>0.5))
nsup
#> [1] 68

#proportion of individuals with an inferred source with consensus support
round(nsup/nrow(epi.data),2)
#> [1] 0.68

#accuracy for the inferred sources of these individuals only
round(sum(acc[inf.support.l>0.5])/nsup, 2)
#> [1] 0.82
```

Noting that this inference needs to be run for around 100,000 MCMC iterations for the tree to converge and these example data are based on only 10000 iterations.

Inferred key timings

Pre-prepared data on the inferred timing of exposured of each individual for each iteration in each chain have been inputted by reading in the comma separated `t_e_current.csv` files produced by independantly seeded `infer()` runs on the same input data.

```
data(te1)
round(head(te1[,1:10]), 3)
#>      V1    V2    V3    V4    V5    V6    V7    V8    V9   V10
#> 1 0.573 4.872 0.002 3.055 3.698 1.641 0.185 1.373 6.346 5.326
#> 2 0.905 4.872 0.002 3.055 3.698 1.641 0.185 1.373 6.346 5.326
#> 3 0.905 4.872 0.002 3.055 3.698 1.641 0.185 1.373 6.346 5.326
#> 4 0.905 4.872 0.002 3.055 3.698 1.641 0.185 1.373 6.346 5.654
#> 5 0.905 4.872 0.002 3.055 3.698 1.641 0.185 1.373 6.346 5.654
#> 6 0.905 4.872 0.002 3.055 3.698 1.641 0.185 1.373 6.346 5.654
round(tail(te1[,1:10]), 3)
#>          V1    V2    V3    V4    V5    V6    V7    V8    V9   V10
#> 9995  2.042 1.429 0.606 1.883 1.792 1.163 2.234 4.135 4.360 3.097
#> 9996  2.042 1.429 0.277 1.883 1.792 1.163 2.234 4.135 3.764 3.097
#> 9997  2.042 1.429 0.277 1.883 1.792 1.163 2.234 4.135 3.764 3.097
#> 9998  2.042 1.429 0.277 1.883 1.792 1.163 2.234 4.135 3.764 3.097
#> 9999  2.042 1.429 0.277 1.883 1.792 1.163 2.234 4.135 3.764 3.097
#> 10000 2.042 1.429 0.277 1.883 1.792 1.163 2.234 4.135 3.764 3.097

data(te2)
round(head(te2[,1:10]), 3)
#>   V1    V2   V3 V4    V5    V6    V7    V8    V9   V10
#> 1  0 4.992 0.65  4 2.448 1.462 0.185 1.373 6.048 5.231
#> 2  0 4.992 0.65  4 2.448 1.462 0.185 1.373 6.048 5.231
#> 3  0 4.992 0.65  4 2.448 1.263 0.185 1.373 6.048 5.864
#> 4  0 4.992 0.65  4 2.448 1.263 0.185 1.373 6.048 5.864
#> 5  0 4.992 0.65  4 2.448 1.263 0.185 1.373 6.087 5.796
#> 6  0 4.992 0.65  4 2.448 1.263 0.460 1.373 6.087 5.796
round(tail(te2[,1:10]), 3)
#>          V1    V2    V3    V4    V5    V6    V7    V8    V9   V10
#> 9995  1.905 2.894 0.031 4.916 2.587 3.046 3.268 4.259 4.828 4.820
#> 9996  1.905 2.894 0.031 4.916 4.155 3.046 3.268 4.259 4.828 4.820
#> 9997  1.905 2.894 0.031 4.916 4.155 3.046 3.268 4.259 4.828 3.832
#> 9998  1.905 2.894 0.031 4.916 4.155 3.046 3.268 4.259 4.828 3.832
#> 9999  1.905 2.894 0.031 4.916 4.155 3.046 3.268 4.259 4.828 3.832
#> 10000 1.905 2.894 0.031 4.916 4.155 3.046 3.268 4.259 4.828 3.832
```

Again we'll use the same burn-in parameters. This time collating lists with posterior median estimates of the day of exposure for each individual with 95% Bayesian credible intervals.

```
burnin=5000
thin = 1
end=1e04
x<-seq(burnin+1,end,thin)

te.l<-list()
te.q.l<-list()
for(i in 1:nrow(epi.data)){
  te.l[[i]]<-c(te1[x,i], te2[x,i])
  te.q.l[[i]]<-quantile(te.l[[i]], probs=c(0.5, 0.025, 0.975))
}
```

```
#inspecting some of the inferred exposure timings
round(te.q.l[[1]], 2)
#>   50%  2.5% 97.5%
#>  1.86  1.16  2.47
round(te.q.l[[2]], 2)
#>   50%  2.5% 97.5%
#>  2.36  1.17  4.02
round(te.q.l[[3]], 2)
#>   50%  2.5% 97.5%
#>  0.40  0.03  1.28
```

Pre-prepared data on the inferred timing of onset of infectiousness of each individual for each iteration in each chain have been inputted by reading in the comma separated `t_i_current.csv` files produced by independantly seeded `infer()` runs on the same input data.

```
data(ti1)
round(head(ti1[,1:10]), 3)
#>   V1 V2 V3 V4    V5 V6 V7 V8    V9 V10
#> 1  1  5  1  9 4.000  2  2  3 6.732   6
#> 2  1  5  1  9 4.000  2  2  3 6.732   6
#> 3  1  5  1  9 4.000  2  2  3 6.732   6
#> 4  1  5  1  9 4.000  2  2  3 6.732   6
#> 5  1  5  1  9 4.000  2  2  3 6.732   6
#> 6  1  5  1  9 4.652  2  2  3 6.732   6
round(tail(ti1[,1:10]), 3)
#>          V1    V2    V3    V4    V5    V6    V7    V8    V9   V10
#> 9995  2.544 2.494 1.063 5.944 3.605 1.823 6.262 7.921 7.681 5.833
#> 9996  2.544 2.494 1.063 5.944 3.605 1.823 6.262 7.921 7.681 5.833
#> 9997  2.544 2.494 1.063 5.944 3.605 1.823 6.262 7.921 7.681 5.833
#> 9998  2.544 2.494 1.063 5.944 3.605 1.823 6.262 7.921 7.681 5.833
#> 9999  2.544 2.494 1.063 5.944 3.605 1.823 6.262 7.921 7.681 5.833
#> 10000 2.544 2.494 1.063 5.944 3.605 1.823 6.262 7.921 7.681 5.833

data(ti2)
round(head(ti2[,1:10]), 3)
#>   V1 V2 V3 V4 V5    V6 V7 V8    V9 V10
#> 1  1  5  1  9  4 3.146  2  3 7.000   6
#> 2  1  5  1  9  4 3.146  2  3 7.000   6
#> 3  1  5  1  9  4 3.146  2  3 6.125   6
#> 4  1  5  1  9  4 3.146  2  3 6.125   6
#> 5  1  5  1  9  4 3.146  2  3 6.125   6
#> 6  1  5  1  9  4 3.146  2  3 6.125   6
round(tail(ti2[,1:10]), 3)
#>          V1    V2    V3     V4   V5    V6   V7    V8    V9   V10
#> 9995  2.889 4.081 1.748  8.271 7.24 3.336 3.99 7.572 6.682 6.082
#> 9996  2.889 4.081 1.748  8.271 7.24 3.336 3.99 7.572 6.682 6.082
#> 9997  2.889 4.081 1.748  8.271 7.24 3.336 3.99 7.572 6.682 6.082
#> 9998  2.889 4.081 1.748  8.271 7.24 3.336 3.99 7.572 6.682 6.082
#> 9999  2.889 4.081 1.748 10.267 7.24 3.336 3.99 7.572 6.682 6.082
#> 10000 2.889 4.081 1.748 10.267 7.24 3.336 3.99 7.572 6.682 6.082
```

Again we'll use the same burn-in parameters. This time collating lists with posterior median estimates of the day of onset of infectousness for each individual with 95% Bayesian credible intervals.

```
burnin=5000
thin = 1
end=1e04
x<-seq(burnin+1,end,thin)

ti.l<-list()
ti.q.l<-list()
for(i in 1:nrow(epi.data)){
  ti.l[[i]]<-c(ti1[x,i], ti2[x,i])
  ti.q.l[[i]]<-quantile(ti.l[[i]], probs=c(0.5, 0.025, 0.975))
}

#inspecting some of the inferred exposure timings
round(ti.q.l[[1]], 2)
#>   50%  2.5% 97.5%
#>  2.54  2.29  2.74
round(ti.q.l[[2]], 2)
#>   50%  2.5% 97.5%
#>  5.00  2.49  9.20
round(ti.q.l[[3]], 2)
#>   50%  2.5% 97.5%
#>  1.27  0.94  1.94
```

Inferred sequences

A sequence is recorded (in every iteration) whenever an individual is infected, sampled or when it infects another individual. So each individual can have a different number of sequences recorded from other individuals, even from itself in different iterations.

Each iteration, the timing of recorded sequences is captured in the output file `seqs_t_current_csv` with 1 row per individual, comma-separated.

```
data(nt.t1)

head(nt.t1)
#>                                                    V1
#> 1                          0,1.23898,1.907,3,3.05547
#> 2                                 4.99155,5.6523,11
#> 3 0.720324,1.14676,1.1981,1.37252,1.73364,6.93066,9
#> 4                                 3.05547,5,12.0619
#> 5                                            1.907
#> 6                                          1.14676

nt.t.l<-apply(nt.t1, MARGIN=1,
              FUN = function(x) as.numeric(unlist(strsplit(x, ","))))

# the timings corresponding to the sequences for the first three individuals
# in the first iteration
nt.t.l[[1]]
#> [1] 0.00000 1.23898 1.90700 3.00000 3.05547
nt.t.l[[2]]
#> [1]  4.99155  5.65230 11.00000
nt.t.l[[3]]
#> [1] 0.720324 1.146760 1.198100 1.372520 1.733640 6.930660 9.000000

# the timings corresponding to the sequences for the first
# individual in the first three iterations
nt.t.l[[pars.aux$n*0+1]]
#> [1] 0.00000 1.23898 1.90700 3.00000 3.05547
nt.t.l[[pars.aux$n*1+1]]
#> [1] 2.49883 2.98227 3.00000 3.08243 3.73211 3.82546 3.83107
nt.t.l[[pars.aux$n*2+1]]
#> [1] 2.76420 2.79978 3.00000 3.00476 3.36984

#this indiviual was sampled at
t.sample<-round(sim.out$t_sample, 0)
t.sample[1]
#> [1] 3
```

For instance in the above example, in the first iteration, the first individual was infected at t=0, then infected 3 further individuals at times t = 1.24, 1.91 and 3.06 days, and was sampled at t=3.00 days.

The number of sequences per individual, per iteration, can thus be derived as:

```
nt.size<-sapply(nt.t.l, length)

nt.size[[pars.aux$n*0+1]]
#> [1] 5
nt.size[[pars.aux$n*1+1]]
#> [1] 7
nt.size[[pars.aux$n*2+1]]
#> [1] 5
```

As can the total number of sequences, and then the sequence sets recorded per iteration, which is used to lookup specific sequences:

```
#total number of sequences recorded
sum(nt.size)
#> [1] 3140

#matrix to hold number of sequences per individual (column) per iteration (row)
nt.size.mat<-matrix(nt.size, byrow = T, ncol=pars.aux$n)
nt.size.mat[1:2,1:10]
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#> [1,]    5    3    7    3    1    1    9    4    4     4
#> [2,]    7    5    5    2    1    1    5    3    3     3

#total number of sequences stored per iteration
nt.size.mat.sizes <- apply(nt.size.mat, MARGIN = 1, FUN = sum)
nt.size.mat.sizes
#>  [1] 279 287 286 286 286 286 286 286 286 286 286
```

This can then be used to lookup specific inferred sequences, per iteration, per individual.

The corresponding inferred sequences are stored in the file `seqs_current.csv`.

```
data(nt.seq1)

#1 sequence per row, corresponding to the timings in nt.t1
dim(nt.seq1)
#> [1] 3140 7667
```

For the first individual in the first iteration, the 5 sequences are stored as:

```
nt.size[[1]]
#> [1] 5
nt.seq1[1:5,1:10]
#>   V1 V2 V3 V4 V5 V6 V7 V8 V9 V10
#> 1  2  4  3  4  2  1  3  3  2   4
#> 2  2  4  3  4  2  1  3  3  2   4
#> 3  2  4  3  4  2  1  3  3  2   4
#> 4  2  1  3  2  2  1  3  3  4   4
#> 5  2  1  3  2  2  1  3  3  4   4
```

For the second individual in the first iteration, the 3 sequences are stored as follows:

```
nt.size[[1]]
#> [1] 5
nt.size[[2]]
#> [1] 3
nt.seq1[6:8,1:10]
#>   V1 V2 V3 V4 V5 V6 V7 V8 V9 V10
#> 6  2  4  3  2  2  3  3  3  4   4
#> 7  2  4  3  2  2  3  3  3  4   4
#> 8  2  1  3  2  2  1  3  3  4   4
```

For the second individual in the second iteration, there are 5 sequences recorded:

```
nt.size[[pars.aux$n*1+2]]
#> [1] 5
```

And these start at row 287 in the sequence data, i.e.:

```
#count sequences in the first iteration
.row.nos <- nt.size.mat.sizes[1]
#add sequences for the first individual in the 2nd iteration
.row.nos <- .row.nos + nt.size[[pars.aux$n*1+1]]
#count sequences for the second individual in the 2nd iteration
.row.nos <- .row.nos + 1:(nt.size[[pars.aux$n*1+2]])
.row.nos
#> [1] 287 288 289 290 291

nt.seq1[.row.nos,1:10]
#>     V1 V2 V3 V4 V5 V6 V7 V8 V9 V10
#> 287  2  1  3  2  2  1  3  3  4   4
#> 288  2  1  3  2  2  1  3  3  4   4
#> 289  2  1  3  2  2  1  3  3  4   4
#> 290  2  1  3  2  2  1  3  3  4   4
#> 291  2  1  3  2  2  1  3  3  4   4
```

For the second individual in the third iteration, there are 4 sequences recorded:

```
nt.size[[pars.aux$n*2+3]]
#> [1] 4
```

And these start at row 572 in the sequence data, i.e.:

```
#count sequences in the earlier iterations
.row.nos <- sum(nt.size.mat.sizes[1:2])
#add sequences for the first individual in the 3nd iteration
.row.nos <- .row.nos + nt.size[[pars.aux$n*2+1]]
#count sequences for the second individual in the 3nd iteration
.row.nos <- .row.nos + 1:(nt.size[[pars.aux$n*2+2]])
.row.nos
#> [1] 572 573 574 575 576

nt.seq1[.row.nos,1:10]
#>     V1 V2 V3 V4 V5 V6 V7 V8 V9 V10
#> 572  2  1  3  2  2  1  3  3  4   4
#> 573  2  1  3  2  2  1  3  3  4   4
#> 574  2  1  3  2  2  1  3  3  4   4
#> 575  2  1  3  2  2  1  3  3  4   4
#> 576  2  1  3  2  2  1  3  3  4   4
```