The Jaatha HowTo

Paul R. Staab, Lisha Mathew and Dirk Metzler

Version 2.4-1

1 Introduction

Jaatha is a fast composite likelihood method to estimate model parameters of the evolutionary history of (at the moment) two related species or populations. To do so, it uses SNP data from multiple individuals from both species and – optionally but highly recommended – one or more outgroup sequences. This HowTo describes the method and gives an example of using its implementation as `R` package `jaatha`.

The package itself can be obtained from CRAN using

```
install.packages('jaatha')
```

or downloaded from http://evol.bio.lmu.de/_statgen/software/jaatha. Jaatha runs on R under Windows, OS X (Mac) and Linux with the following restrictions: On Windows the parallelization and finite sites simulation using Seq-Gen is currently not supported.

A more detailed description of the algorithm can be found in Mathew et al. [2013]. Further information about the `R` functions used in this document can be obtained by calling `help()` with the functions name as argument.

We kindly ask you to cite the above paper when using Jaatha in a publication.

2 A demographic model

Before we can apply Jaatha to estimate parameters, we first need to create a model of the evolutionary history of the two species. Jaatha cannot account for the effects of selection, hence we assume a neutral evolution. It can estimate the effects that either "demographic" events – like an expansion of the size of one population or migration between the two populations – or molecular events – like mutation or recombination – have on the genome of the populations. To emphasize that we can not include selection, we refer to a scenario of the evolution of the two species under such events as a "demographic model".

For now, assume that we know that our two species are closely related; hence they must have separated at a certain time in the past. There may still be gene flow ongoing between them to which we will refer to as migration from one population into the other. Hence, we could propose the simple demographic model described in Figure 1.

To specify that model in `R`, we first need to load `jaatha`.

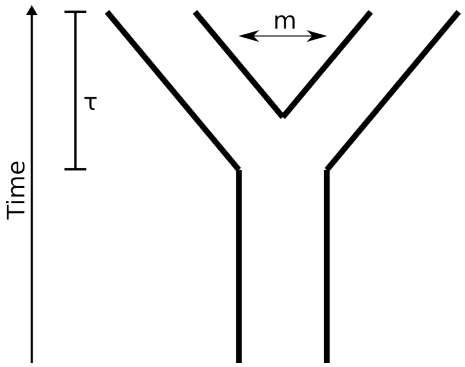

Figure 1: A simple demographic model: The ancestral population splits into two populations τ time units ago, and afterwards individuals migrated from one population to the other with a migration rate M. Mutations are occurring with rate θ and recombination with a known rate ρ.

```
library(jaatha)

## Loading required package:  methods
```

We can now create an 'empty' demographic model `dm` using the `dm.createDemographicModel()` function:

```
dm <- dm.createDemographicModel(sample.sizes=c(12,14),
                                loci.num=50,
                                seq.length=1000)
```

The parameter `sample.sizes` here corresponds to the number of individuals we have sampled from the first population and second population respectively. The second argument `loci.num` states that we are using data from 70 loci while `seq.length` gives the (average) length of each loci[1].

We can now successively add the other assumptions of our model:

```
dm <- dm.addSpeciationEvent(dm, .1, 5, new.time.point.name='tau')
dm <- dm.addSymmetricMigration(dm, .01, 5, new.par.name='M')
dm <- dm.addMutation(dm, 1, 20, new.par.name='theta')
dm <- dm.addRecombination(dm, fixed=20)
```

The first parameter is the demographic model to which we want to add an assumption/feature. The two following numbers represent the range for the corresponding parameter. The lower border has to be strictly greater the zero, as we are using a logarithmic transformation of the parameter space. The parameters are scaled as in the popular simulation program `ms` [Hudson, 2002] that we use for simulations:

[1] This is only used when a finite sites model is assumed or if intra-locus recombination is included.

- The parameter for the *speciation* event is the split time τ, which states how many generations ago the split of the population has occurred. As usual in population genetics, it is measured in units of $4N_1$ generations ago, where N_1 is the (diploid) effective population size of the first population.
- The parameter for the (symmetric) *migration* is the scaled migration rate M, which is given by $M = 4N_1m$, where m is the fraction of individuals of each population which are replaced by immigrants from the other population each generation.
- The *mutation* parameter θ is $4N_1$ times the neutral mutation rate per locus.
- Finally, the *recombination* parameter ρ is $4N_1$ times the probability of recombination between the ends of the locus per generation. Jaatha uses the JSFS as summary statistics, which is not sensitive for recombination events. Hence, estimation of the recombination rate will fail. You should set this to a fixed value reflecting known estimates in your model organism.

Finally we can check your model:

```
dm

## Used simulation program: ms
##
## Parameters to estimate:
##    name lower.range upper.range
## 1   tau        0.10           5
## 2     M        0.01           5
## 3 theta        1.00          20
##
## Fixed parameters:
##   name value
## 1  rho    20
##
## Simulation command:
## ms 26 50 -I 2 12 14 -ej tau 2 1 -em 0 2 1 M -em 0 1 2 M -t theta -r rho 1000
```

Keep in mind that a 'good' model – which is one that approximates the real demographic history but is also as simple as possible – is crucial for obtaining meaningful estimates in the end. Jaatha will always try to find the parameters that make the model fit best to your data. If the model does not fit to the data at all, Jaatha will still return estimates, but they will not be meaningful.

3 Theoretical Background

It is important to understand the key concepts behind Jaatha before we can apply it. Like many estimation methods that rely on simulations, Jaatha tries to find the parameters that best fits [2] to your data by simulating artificial data

[2] for Jaatha, the 'best' parameter combination is the one with the highest composite likelihood

for many different parameter combinations. It uses a learning algorithm to determine how the different parameter values influence the simulated data and uses that knowledge to find the best parameter combination for your data.

You can imagine Jaatha as a method that runs through the parameter space – the space of all possible parameter combinations, in our example a cube with borders from 0.1 to 5, 0.01 to 5 and 1 to 20 – simulating in a small part of the parameter space around the current position (we call this area a *block*). It then searches the new maximum of the current blocks and moves to it, builds a new block around it and so on. The search finally stops when the likelihood cannot be improved anymore or a maximal number of steps has been reached.

To compare the simulated data to the real one, Jaatha uses *summary statistics* of the data. As default, it calculates the Joint Site Frequency Spectrum (JSFS) of the data and further summarizes it by evaluating different sums over the JSFS. Please refer to Naduvilezhath et al. [2011] for a detailed description.

4 Importing Your Data

To run Jaatha, you need to calculate the JSFS of your data set. We do not differentiate between loci and use the JSFS over all loci. You can either just concatenate all loci and the calculate the JSFS for the combined sequence, or calculate the JSFS for every loci and sum them up.

To calculate the JSFS, you can use the function `calculateJsfs()`. This function accepts data imported with the `read.dna()` function from the package *ape*. It assumes that you provide a joined, aligned data set with multiple samples from two populations and – optional but highly recommended – one or more outgroup sequences. Additionally, you must provide the numbers of the sequences in the dataset that belong to population one, population two, and the outgroup, respectively.

Please consult the documentation of *ape* in order to get more information about `read.dna()`. For example, the import of the data could look like this:

```
library(ape)
# The path to the data
sample.file <- system.file('example_fasta_files/sample.fasta',
                           package='jaatha')
# We use system.file() here to get the path of an example file included in the jaatha
# package. You can just give the path to the file as string:

# sample.file <- 'path/to/your/file.fasta'

# Reading the data
sample.data <- read.dna(sample.file, format='fasta',
                        as.character=TRUE)

# Calculating the JSFS
sample.jsfs <- calculateJsfs(sample.data,
                             pop1.rows=3:7,
                             pop2.rows=8:12,
                             outgroup.row=1:2)
```

```
sample.jsfs

##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    1    0    0    0    0
## [2,]    1    1    1    0    0    0
## [3,]    2    0    1    0    0    0
## [4,]    1    1    0    2    0    1
## [5,]    1    0    0    0    3    0
## [6,]    1    0    0    0    0    0
```

For the purpose of this HowTo, we will use a simulated JSFS, for which we know the real parameters:

```
# Real parameters: M = 1, tau = 1 and theta = 10
real.pars <- c(1,1,10)

# Simulate a JSFS with this parameters
sum.stats <- dm.simSumStats(dm, real.pars)
jsfs <- sum.stats$jsfs

# Print the upper left part of the JSFS
jsfs[1:7, 1:7]

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    0  430  179   98   69   27   22
## [2,]  523   90   70   46   40   43   28
## [3,]  252   25   20   24   16   40   17
## [4,]  156   27   20    9   11    8    6
## [5,]   79   18   10   13   12    6    5
## [6,]   85   23   12    6    3   11    7
## [7,]   60   16   19   16   12   11    5
```

5 Running Jaatha

Jaatha is divided into two parts. First we find good starting positions by simulating very coarsely across the entire parameter space. We call this part *initial search*. Afterwards a more thorough *refined search* is performed starting from the best positions of the first step. Before starting the search, we need to set some options like our demographic model, the summary statistics of the real data, and a seed to ensure reproducibility:

```
jaatha <- Jaatha.initialize(dm, jsfs=jsfs, seed=12345, cores=2)
```

For more options refer to `?Jaatha.initialize` or the Jaatha manual. If you are running on Windows, skip the `cores = 2` option.

5.1 The Initial Search

For the initial search, we divide the parameter space into equally-sized blocks by dividing each of the n parameters ranges into `blocks.per.par` intervals such that we obtain $(\texttt{blocks.per.par})^n$ blocks. Within each block we simulate `sim` data sets with – on a logarithmic scale – uniformly drawn parameter values within each block. To ensure a better sampling of the edges, we simulate in addition data sets for all corner points of each parameter block.

For these data sets we then fit the GLMs and estimate the parameter combination with the maximal score[3]. Each of the blocks provides a single best parameter combination.

In R, the initial search is performed with the command

```
jaatha <- Jaatha.initialSearch(jaatha,
                               sim=100,
                               blocks.per.par=2)

## *** Block 1 of 6 : 0.1-0.707 x 0.01-5 x 1-20
## Best parameters 0.32 0.357 12.98 with estimated log-likelihood -140.3
##
## *** Block 2 of 6 : 0.1-5 x 0.01-0.224 x 1-20
## Best parameters 0.178 0.224 12.98 with estimated log-likelihood -111.8
##
## *** Block 3 of 6 : 0.1-5 x 0.01-5 x 1-4.472
## Best parameters 5 5 4.472 with estimated log-likelihood -669.9
##
## *** Block 4 of 6 : 0.707-5 x 0.01-5 x 1-20
## Best parameters 1.155 1.33 10.4 with estimated log-likelihood -139.8
##
## *** Block 5 of 6 : 0.1-5 x 0.224-5 x 1-20
## Best parameters 1.136 0.805 10.43 with estimated log-likelihood -93.1
##
## *** Block 6 of 6 : 0.1-5 x 0.01-5 x 4.472-20
## Best parameters 0.786 0.972 11.17 with estimated log-likelihood -106
##
##      log.likelihood   tau     M  theta
## [1,]          -93.1 1.136 0.805 10.433
## [2,]         -106.0 0.786 0.972 11.168
## [3,]         -111.8 0.178 0.224 12.983
## [4,]         -139.8 1.155 1.330 10.403
## [5,]         -140.3 0.320 0.357 12.979
## [6,]         -669.9 5.000 5.000  4.472
```

To visualise the estimates for good stating positions sorted by score, type:

```
Jaatha.getStartingPoints(jaatha)

##      log.likelihood   tau     M  theta
```

[3] In this phase, Jaatha uses a score instead of the likelihood for computational reasons. The likelihood is proportional to exp(score). The higher the score, the higher the likelihood.

```
## [1,]          -93.1 1.136 0.805 10.433
## [2,]         -106.0 0.786 0.972 11.168
## [3,]         -111.8 0.178 0.224 12.983
## [4,]         -139.8 1.155 1.330 10.403
## [5,]         -140.3 0.320 0.357 12.979
## [6,]         -669.9 5.000 5.000  4.472
```

Here, there is a big reduction in the likelihoods after the first three blocks. This is suggesting that we use the first three blocks as starting positions for the refined search. For now, we will just use the first point.

5.2 The Refined Search

Now we can conduct the more thorough refined search described above to improve the likelihood approximations.

```
jaatha <- Jaatha.refinedSearch(jaatha, best.start.pos=1, sim=50)

## *** Search with starting Point in Block 1 of 1 ****
## -----------------
## Step No 1
## Best parameters 1.136 0.805 10.43 with estimated log-likelihood -89.01
##
## -----------------
## Step No 2
## Best parameters 1.196 0.923 9.68 with estimated log-likelihood -72.86
##
## -----------------
## Step No 3
## Best parameters 1.243 0.923 9.666 with estimated log-likelihood -70.27
##
## -----------------
## Step No 4
## Best parameters 1.281 0.929 9.525 with estimated log-likelihood -74.85
##
## -----------------
## Step No 5
## Best parameters 1.287 0.922 9.445 with estimated log-likelihood -73.68
##
## -----------------
## Step No 6
## Best parameters 1.329 0.903 9.533 with estimated log-likelihood -73.12
##
## -----------------
## Step No 7
## Best parameters 1.287 0.911 9.565 with estimated log-likelihood -72.65
##
## -----------------
## Step No 8
## Best parameters 1.282 0.938 9.666 with estimated log-likelihood -71.88
```

```
##
## -----------------
## Step No 9
## Best parameters 1.162 0.878 9.796 with estimated log-likelihood -75.42
##
## -----------------
## Step No 10
## Best parameters 1.282 0.971 9.63 with estimated log-likelihood -71.21
##
## -----------------
## Step No 11
## Best parameters 1.162 0.943 9.82 with estimated log-likelihood -71.94
##
## -----------------
## Step No 12
## Best parameters 1.171 0.961 9.811 with estimated log-likelihood -73.94
##
## *** Finished search ***
## Seems we can not improve the likelihood anymore.
## Calculating log-composite-likelihoods for best estimates:
## * Parameter combination 1 of 10
## * Parameter combination 2 of 10
## * Parameter combination 3 of 10
## * Parameter combination 4 of 10
## * Parameter combination 5 of 10
## * Parameter combination 6 of 10
## * Parameter combination 7 of 10
## * Parameter combination 8 of 10
## * Parameter combination 9 of 10
## * Parameter combination 10 of 10
##
##
## Best log-composite-likelihood values are:
##    log.cl block   tau      M theta
## 8  -74.17     1 1.171 0.9609 9.811
## 9  -74.21     1 1.282 0.9714 9.630
## 10 -74.55     1 1.162 0.9434 9.820
## 3  -74.73     1 1.281 0.9294 9.525
## 7  -75.10     1 1.282 0.9379 9.666
```

Here we use just one starting position for the sake of simplicity (e.g. `best.start.pos` is set to one). In a real analysis, it is recommended to start independent search from multiple starting positions (use a value greater than one), to avoid getting stuck in local maxima.

During the search, we build a block with `half.block.size` in each direction (on a logarithmic scale) at each step, and perform simulations for `sim` random parameter combinations within this block (plus one for very corner). We use this information to estimate the composite maximum likelihood parameters within this block and take this value as new starting position for the next step.

The algorithm stops when the score has not changed more than `epsilon` for

five consecutive steps or step `max.steps` is reached. To avoid getting stuck in local maxima, the `weight` option decreases the weight of simulations of previous blocks.

Finally the log composite likelihoods for the best ten parameter combinations are approximated using `sim.final` simulations. These are values printed at the end of the search. This matrix can also be accessed via

```
likelihoods <- Jaatha.getLikelihoods(jaatha)
print(likelihoods[1:3, ])

##    log.cl block   tau      M theta
## 8  -74.17     1 1.171 0.9609 9.811
## 9  -74.21     1 1.282 0.9714 9.630
## 10 -74.55     1 1.162 0.9434 9.820
```

6 Parallelization

On Linux and OS X, Jaatha can distribute the simulations on multiple CPU cores. To use this feature, set the `cores` option during initialization. The value of the option specifies the number of cores you want to use. Jaatha will distribute its simulation evenly this cores. Since version 2.2 the number of cores no longer affects Jaatha's seeding system, e.g. you will get the same results for the same seed no matter how many cores you use.

There is known problem with our parallelization approach on OS X. If you get errors like

```
The process has forked and you cannot use this CoreFoundation
functionality safely. You must exec().
```

use the plain command line version of R instead of the GUI.

7 Finite sites models

As described in Mathew et al. [2013], you can use finite sites mutation models in Jaatha. However, this currently only works on Linux and OS X and requires that Seq-Gen [Rambaut and Grassly, 1997] is installed on your system. On Debian GNU/Linux and it's derivates (Ubuntu, Mint) you can install it running `apt-get install seq-gen` as root. Otherwise download the current version from http://tree.bio.ed.ac.uk/software/seqgen and compile it according to the instruction within the package. Jaatha will search for the Seq-Gen executable using your PATH variable. If you get an error that it is not able to find it, you can specify the path to the executable using the `Jaatha.setSeqgenExecutable` function.

To create a finite sites model, you must specify a mutation model using the `dm.setMutationModel` function and add an outgroup to your model using `dm.addOutgroup`. Optionally, you can than add a mutation rate heterogeneity to your model using `dm.addMutationRateHeterogenity`.

8 Confidence Intervals

From Version 2.2 on, Jaatha also offers the function `Jaatha.confidenceIntervals` to calculate parametric bootstrap confidence intervals for the estimated parameters. Bootstrapping requires to rerun the complete analysis multiple times, in the case of parametric bootstrapping on simulated data. This means that in order to get confidence intervals, you need to execute about 50 to 100 Jaatha runs, which can be quite time consuming. Therefore, we recommend doing this only on a Linux/Mac multicore system (the more cores the better), where the runs can be executed at least partially in parallel. Set the `cores` option to the number of cores you want to use (the individual Jaatha runs will always run on the single core). We usually use up to 32 cores on powerful computation servers for this. The confidence intervals are bias-correct and accelerated (BCa), as described in Chapter 14.3 of Efron and Tibshirani [1994].

9 More on Demographic Models

The demographic model system can also deal with population size changes. There are two functions that add a change in the size of a population to the model, `dm.addSizeChange` and `dm.addGrowth`. 'SizeChanges' are instantaneous changes in the population size, while 'Growth' refers to an exponential increase or decrease in population size over time. Please keep in mind that models are always specified looking backwards in time. Hence a size change affects the time from the size changes time point on further backwards into the past. The parametrisation of a model including both growth and time changes can be quite challenging. As an example, here is the code for model S1.1 in Mathew et al. [2013]:

```
nLoci <- 100
rho <- 5

dm <- dm.createDemographicModel(c(25,25), nLoci, 1000)
dm <- dm.addMutation(dm, 5, 20, new.par.name="theta")
dm <- dm.addSpeciationEvent(dm, 0.017, 20, new.time.point.name="tau")
dm <- dm.addSymmetricMigration(dm, 0.005, 5, new.par.name="m")
# Everything but a fixed recombination rate does not make sense
dm <- dm.addRecombination(dm, fixed=rho)

# "SizeChange"s are instanious sizes changes (ms' -n, -N, -en and -eN)
dm <- dm.addSizeChange(dm, 0.05, 10, population=2, at.time="0", new.par.name="q")

# Parameters s1 + s2 have to be created separately, because they are no direct
# parameters of any command
dm <- dm.addParameter(dm, par.name="s1", 0.05, 10)
dm <- dm.addParameter(dm, par.name="s2", 0.05, 10)
dm <- dm.addSizeChange(dm, par.new=FALSE, parameter="s1+s2", population=1, at.time="tau")

# Growth is an exponential population expansion over time (e.g. ms' -g, -G, -eg and -eG)
dm <- dm.addGrowth(dm, par.new=FALSE, parameter="log(1/s1)/tau", population=1)
```

```
dm <- dm.addGrowth(dm, par.new=FALSE, parameter="log(q/s2)/tau", population=2)
```

Forwards in time, this model consists of an ancestral population of relative size $s1 + s2$, which splits into two population at time τ. These population have size $s1$ and $s2$, and grow to their present relative sizes 1 and q.

References

Bradley Efron and Robert Tibshirani. *An introduction to the bootstrap*. Chapman & Hall, New York, 1994. ISBN 0412042312 9780412042317.

Richard R. Hudson. Generating samples under a wright-fisher neutral model of genetic variation. *Bioinformatics*, 18(2):337 –338, February 2002. doi: 10.1093/bioinformatics/18.2.337.

Lisha A. Mathew, Paul R. Staab, Laura E. Rose, and Dirk Metzler. Why to account for finite sites in population genetic studies and how to do this with jaatha 2.0. *Ecology and Evolution*, 2013. doi: 10.1002/ece3.722. URL http://onlinelibrary.wiley.com/doi/10.1002/ece3.722/abstract.

Lisha Naduvilezhath, Laura E Rose, and Dirk Metzler. Jaatha: a fast composite likelihood approach to estimate demographic parameters. *Molecular Ecology*, 20(13):2709–2723, July 2011. doi: 10.1111/j.1365-294X.2011.05131.x.

Andrew Rambaut and Nicholas C Grassly. Seq-gen: An application for the monte carlo simulation of DNA sequence evolution along phylogenetic trees. *Comput Appl Biosci*, 13(3):235–238, January 1997. doi: 10.1093/bioinformatics/13.3.235. URL http://bioinformatics.oxfordjournals.org/content/13/3/235.