Lab 7. Bayesian inference using RevBayes, continued

Due Sunday, November 20, 2022 at 3 pm

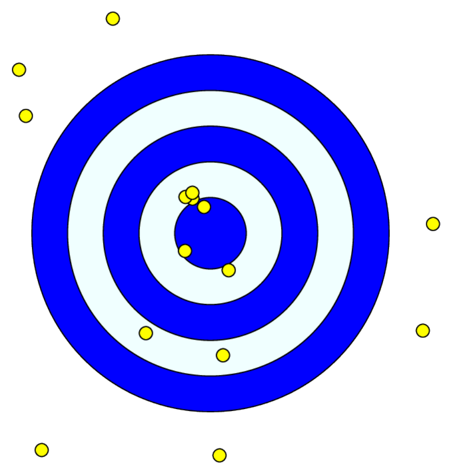
In today's lab, we are going to pick up where we left off last week and continue our RevBayes journey. First, we will use Rev – the language underlying RevBayes – to write our own simple Markov chain Monte Carlo (MCMC) simulation, in order to get a sense of what exactly it is doing and why it works. Next, we will apply our knowledge of Rev and MCMC to phylogenetics, and learn about some challenges that are specific to Bayesian phylogenetic inference: namely, prior choice.

Part 1: Archery and Markov chain Monte Carlo

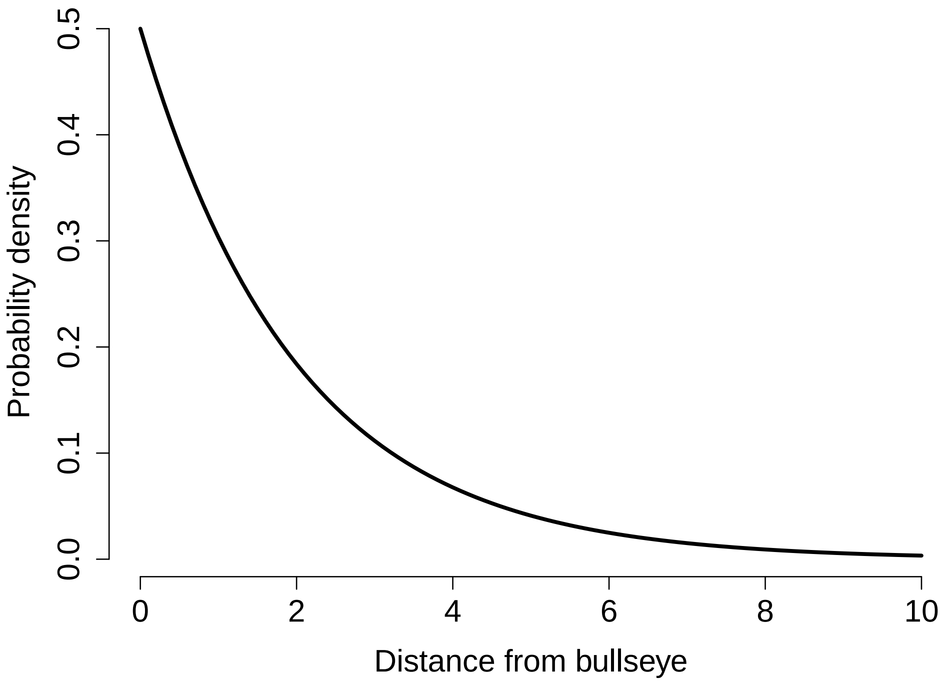
What follows has been adapted from a RevBayes tutorial developed by Prof. Tracy Heath's lab at the Iowa State University. You can access the full version from `https://revbayes.github.io/tutorials/mcmc/archery.html`

Last time, we mentioned that most of the time, we cannot calculate posterior distributions analytically, that is, with pen and paper. Instead, we have to rely on simulations. The idea of estimating parameter values using simulation is surprisingly old and long predates the existence of electronic computers (look up *Buffon's needle* if you are interested), but it wasn't until sufficiently powerful computers became available in the 1990s that one such simulation method, called "Markov chain Monte Carlo" or MCMC for short, exploded in popularity.

To make sure we understand what MCMC is doing, we will once again start with a simple, tractable model. Specifically, we will estimate the probability distribution of an archer's arrows landing at a particular distance from the center of a target:



To construct our model, we will assume that these distances follow an exponential distribution with some unknown mean μ (mu), which corresponds to the *expected distance* from the bullseye. We could interpret this as an estimate of the archer's skill: an experienced archer's μ would be smaller than a beginner's. The exponential distribution tells us that an arrow is much more likely to land at a small distance from the center than at a large distance:



Let's further assume our Bayesian archer shot n arrows at the target, with some average distance $\bar{d}$ from the bullseye. It turns out that $\bar{d}$ follows a gamma distribution with two parameters: the shape, here equal to n, and the rate, equal to $\frac{n}{\mu}$. (We've already encountered the gamma distribution in Lecture 10.3, when modeling rate variation across characters.) The mean m of the gamma distribution can be computed as:

$$m = \frac{\text{shape}}{\text{rate}} = \frac{n}{\frac{n}{\mu}} = \mu$$

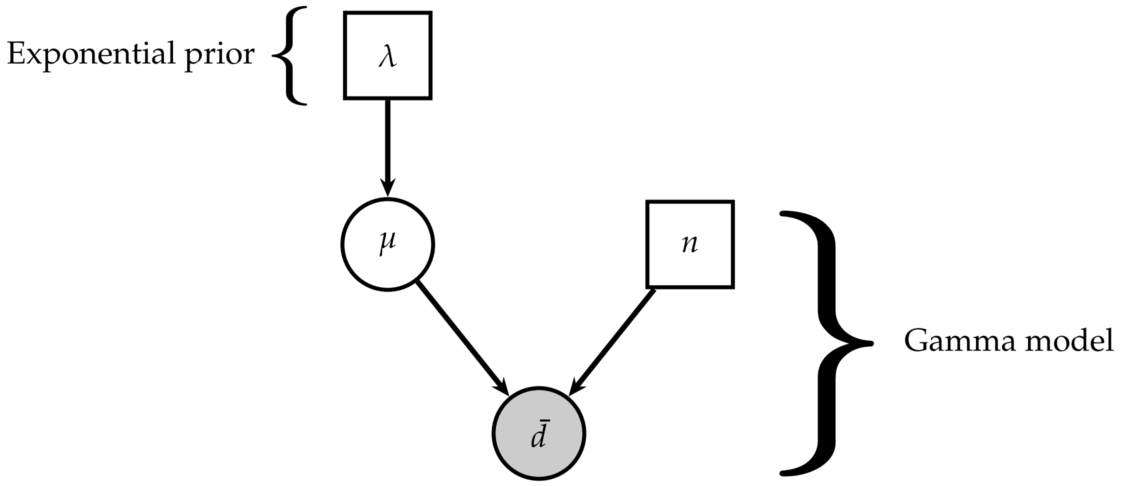
We already know this quantity: it represents the expected distance *from the bullseye*, or the archer's *accuracy*. The variance v of our gamma distribution is equal to

$$v = \frac{\text{shape}}{\text{rate}^2} = \frac{n}{\left(\frac{n}{\mu}\right)^2} = \frac{\mu^2}{n}$$

and represents the archer's *precision*, or dispersion: how far the arrows are expected to land *from each other*.

If we recall what we learned last week, we can immediately tell that our gamma distribution represents a model with two parameters, n and μ. Of these, the number of arrows n is known and fixed, while the expected distance μ is a free parameter that we are going to estimate. To do so, we first need to place a prior distribution on it. In this example, we are going to use an exponential prior distribution, with a single parameter denoted λ (lambda). We are going to set λ to 1, but we could also imagine making this parameter larger (if we had strong reason to believe that the archer is very accurate) or smaller (if we believed the opposite).

Returning to the graphical notation introduced in our previous lab, we can represent this model as follows:



The last thing we are going to need before we start the inference process is some actual data $\bar{d}$. It turns out that RevBayes can be of assistance in this case, too: we are going to cheat and generate our data using the very model we just created, based on some predefined "true" value of μ. I use the word "cheat" in a very facetious manner here, because this workflow is actually perfectly valid and very convenient. In particular, it allows us to directly compare the estimated value of μ to its known true value, which we can't ever do with real data. (If we knew the true values of our parameters, all inference would clearly be superfluous.) With this in mind, let's get started!

Note that the different assignment operators we learned about last time tell RevBayes how different variables should be treated within its own MCMC machinery. Today, however, we are building our own MCMC simulator from scratch, so we can stick with the constant-variable assignment operator, `<-`:

```
lambda <- 1.0
n <- 10
```

Our “true” μ value, `mu_true`, can be whatever we want it to be:

```
mu_true <- 1.0
```

Finally, using the `rgamma()` function, I’m going to make one draw (first argument) from a gamma distribution with a shape of `n` (second argument) and a rate of `n/mu_true` (third argument):

```
arrow_mean <- rgamma(1, n, n/mu_true)[1]
```

Our first step is to initialize the MCMC simulation with some starting value for the estimated parameter *mu*, which we will do by storing a draw from our exponential prior in yet another constant variable:

```
mu <- rexp(1, lambda)[1]
```

Next, let’s do something new: define our own function describing the likelihood of μ. This function will take a proposed value of μ as its input, and return its probability density conditional on the observed data as its output. Once again, the syntax is similar to R: the `function` keyword is used to define a function, and the `return` keyword is used to return the output. Below, we tell RevBayes to treat the likelihood as gamma-distributed if the proposed value is non-negative, and to set it to 0 if the proposed value is negative:

```
function likelihood(mu) {
    if (mu < 0.0) {
        return 0.0
    } else {
        return dgamma(arrow_mean, n, n/mu, log=false)
    }
}
```

We can define another function to describe the prior probability of a proposed μ value. Note that in agreement with our definition of prior probabilities, this function does not depend on the data (`arrow_mean`):

```
function prior(mu) {
    if (mu < 0.0) {
        return 0.0
    } else {
        return dexp(mu, lambda, log=false)
    }
}
```

We can also prepare a log file in which we will store the sampled values of μ:

```
# The file is going to have two columns, called "iteration" and "mu"
write("iteration", "mu", "\n", file="archery.log")
# Write down the initial value of mu, which we have drawn above
write(0, mu, "\n", file="archery.log", append=TRUE)
```

And let's print them to the screen as well:

```
print("iteration", "mu")
print(0, mu)
```

Now that all the ingredients are in place, let's take a detailed look at what our MCMC simulator is going to do:

1. Generate an initial value for μ – **done!**
2. Based on the current value μ, draw a new value (denoted μ', or "mu prime") from some proposal distribution q
3. Calculate the acceptance ratio R as follows:

$$\begin{aligned} R &= \min\left\{1, \frac{p(\bar{d}\,|\,\mu')}{p(\bar{d}\,|\,\mu)} \times \frac{p(\mu')}{p(\mu)} \times \frac{q(\mu)}{q(\mu')}\right\} \\ &= \min\{1, \text{likelihood ratio} \times \text{prior ratio} \times \text{proposal ratio}\} \end{aligned}$$

   Note that we read min {} as "the smaller of". Therefore, either the product of the three ratios turns out to be greater than 1, in which case we are always going to accept the newly proposed value μ' (= accept with a probability of 1), or it turns out to be smaller, in which case we will only accept μ' with probability R. What does that mean, you ask? That's where our next step comes in:

4. Draw a random (uniformly distributed) number u from 0 to 1.

```
if $u$ < R:
    Accept the proposal and set $\mu = \mu'$
else:
    Reject the proposal and keep $\mu$ at its current value.
```

5. Record the current value
6. Go back to step 2

All that remains to do is to translate this into Rev. We will use a fixed number of iterations (20,000), storing the sampled parameter values every 20 iterations. We will also use a uniform proposal distribution q. As we will see in a moment, this distribution will be centered on the current value of μ, meaning that the difference between the upper bound of the distribution and μ will be the same as the difference between μ and the lower bound of the distribution. We will call this difference `delta` and set it equal to 1:

```
n_iter = 20000
printgen = 20
delta = 1
```

This `delta` would be an example of what we call a *tuning parameter* in MCMC terminology, because it tunes the behavior of our proposal. Making it very small ensures that the newly proposed value μ' will be very close to the current value μ. This will make it very likely to get accepted, but it will make the exploration of different values slow and difficult. Conversely, making it very large will favor bold jumps far away from the current value, many of which will, however, end up in regions of low posterior probability and get rejected. Modern MCMC implementations are often capable of "auto-tuning" (yes, really) their proposals.

Since we want to repeat the same series of operations many times, the natural construct to use is a `for` loop:

```
for(iter in 1:n_iter) {
    # Step 2: propose a new value of mu
    mu_prime <- mu + runif(n=1, -delta, delta)[1]
```

This is our Rev implementation of the uniform proposal distribution described above. We draw a number ranging from negative `delta` to positive `delta` (here, from -1 to 1), and add it to the current value of μ to generate a new, proposed value μ'.

Next, we will use the functions we defined above to compute the prior probabilities and likelihoods of the proposed and current values, which we will in turn use to compute the acceptance ratio R:

```
    # Step 3: compute the acceptance probability
    R <- likelihood(mu_prime)/likelihood(mu) * prior(mu_prime)/prior(mu)
```

What happened to the third term of the product, the proposal ratio? It turns out that for the uniform proposal distribution we chose to use, it is always going to be equal to 1, so we can just leave it out and move on:

```
    # Step 4: accept or reject the proposal
    u <- runif(1 ,0, 1)[1]
   if (u < R) {
       # Accept the proposal
       mu <- mu_prime
    }
```

Finally, we will store the current value of μ to our log file and print it to the screen. To ensure that we only do this every 20th iteration like we said we would, we will divide the current iteration number (`iter`) by our `printgen` variable, and only proceed if the remainder is equal to 0:

```
    # Step 5: record the current value
    if ( (iter % printgen) == 0 ) {
        # Write the sampled value to a file
        write(iter, mu, "\n", file="archery.log", append=TRUE)
        # Print the sampled value to the screen
        print(iter, mu)
    }
```

End the MCMC simulation by closing the entire `for` loop with a matching right curly brace:

```
}
```

Hit Enter / Return to execute. We're done!

For your convenience, all of the above code has been packaged into a single Rev script available from Canvas (`archery.Rev`). You can execute a script in RevBayes as follows: `source("/path/to/script.Rev")`

---

1) Open your log file in Tracer. What is the posterior mean of μ? How close is it to the true value?

2) Repeat the analysis with `mu_true` set to 0.25, but `lambda` still set to 1. How did this affect the results? Recall our lectures on Bayesian inference. Why do you think the posterior distribution changed? Is the result "wrong" in light of your prior beliefs?

3) In RevBayes, the uniform proposal distribution has been pre-implemented for us as `mvSlide`, which also accepts the `delta` tuning argument. Refer to the previous lab (and to the online version of this tutorial, if you like) and re-write our archery exercise in such a way that it makes use of the standard RevBayes machinery (clamped stochastic variables, moves, monitors, the `mcmc()` function). Submit your answer as a Rev script, i.e., a plain-text file with the `.Rev` file extension that stores all of your commands.

Part 2: Bayesian phylogenetics

We are now familiar enough with RevBayes to try our hand at the task for which it was designed: Bayesian phylogenetic inference.

What follows has been adapted from RevBayes tutorials developed by April M. Wright (Southeastern Louisiana University), Michael J. Landis (Washington University in St. Louis), Sebastian Höhna (Ludwig-Maximilians-Universität), Tracy A. Heath (Iowa State University), and Brian R. Moore (University of California, Davis). You can access the full versions from `https://revbayes.github.io/tutorials/morph_tree/` and `https://revbayes.github.io/tutorials/ctmc/`.

First, we will load a character matrix into our RevBayes workspace. RevBayes is fine with Nexus files, but wants them to be clean. We will therefore delete the `ASSUMPTIONS` block (that is, the last four lines) from our `Tedford_2009-1.nex` file. We will assign the contents of our dataset to a new workspace object called `morpho`:

```
morpho = readDiscreteCharacterData("/Users/David/Tedford_2009-1.nex")
```

We can immediately create a few "helper" variables that will come in handy later on. From the `mopho` object, we can extract the number of taxa in our analysis (here, 39) and plug it into the formula for the number of branches in our unrooted tree ($2n - 3$). This will tell us how many branch lengths we will have to estimate. We can also get a list of taxon names, which we will use to initialize our tree:

```
num_taxa <- morpho.size()
num_branches := 2 * num_taxa - 3
taxa <- morpho.names()
```

Just like in Lab 6, we will create two more workspace variables to hold our moves and monitors:

```
moves = VectorMoves()
monitors = VectorMonitors()
```

We can also specify our outgroup right away:

```
out_group <- clade("Hesperocyoninae")
```

Now we are ready to place a prior probability distribution on the topology of our tree, which is the main parameter we are interested in. We will assume that all $(2n - 5)!!$ unrooted topologies for n taxa have equal probability, that is, we will specify a *uniform* distribution over topologies:

```
topology ~ dnUniformTopology(taxa, outgroup=out_group)
```

We see that `topology` is a stochastic variable drawn from the specified prior distribution. To estimate it, we need to place some *moves* or *proposals* on it. As it turns out, the topology moves we use in Bayesian phylogenetic inference are no different from those we used for our maximum-parsimony and maximum-likelihood analyses, which we learned about in Lecture 5.4. Here, we will use the nearest-neighbor interchange (NNI) and subtree prune-and-regraft (SPR) moves. As you may recall, SPR is a bit more sophisticated than NNI, so we may want to use it more sparingly: say, one SPR move per every ten NNI moves. We can do this by specifying *weights* for our moves, which tell RevBayes how often they should be applied per iteration, or relative to other moves:

```
moves.append( mvNNI(topology, weight=num_branches) )
moves.append( mvSPR(topology, weight=num_branches/10.0) )
```

From our maximum-likelihood lectures, we know that the probability of a character going from one state to another depends not just on the topology of our tree, but also on its branch lengths. As a result, we have to estimate these as well, and since we are working in the Bayesian framework, if we want to estimate them, we first have to place a prior probability distribution on them. As we'll see further down below, choosing the right prior distribution for branch lengths can be tricky. For now, we are just going to use a uniform distribution from 0 to 5, which indicates that all values from 0 to 5 expected changes per character are equally likely:

```
for (i in 1:num_branches) {
   branch_lengths[i] ~ dnUniform(0, 5)
   moves.append( mvScale(branch_lengths[i]) )
}
```

In the loop above, we draw our branch lengths one by one from the specified prior, and add them to a newly created vector called `branch_lengths`. With each new branch length, we also define a move that will propose new values for it, and append it to the previously created `moves` vector.

Up until now, we have recorded the values of each parameter we estimated. With branch lengths, however, there is a problem: they only have a meaning when associated with a particular topology. Between the two of them, RevBayes and Tracer will try to tell you that, for example, `branch_lengths[26]` (the 26th element of your branch length vector) has a posterior mean of 0.05, but because the 26th branch of one topology can be associated with a completely different clade than the 26th branch of another topology, that doesn't mean anything much in particular. However, one quantity that we can meaningfully compare across different topologies is tree length, or the sum of all branch lengths:

```
tree_length := sum(branch_lengths)
```

Note that we defined `tree_length` as a deterministic variable: it can vary, so it can't be constant, but its value is fully determined by the individual branch lengths – it cannot vary independently of them.

Similarly, a phylogram – which is what we are estimating here – is fully determined by its topology and branch lengths. We will therefore assemble it from the topology and branch lengths we've already specified. Specifically, using the `treeAssembly()` function, we will tell RevBayes to apply the *i*-th element of the `branch_lengths` vector to the branch subtending the *i*-th node in `topology`, and store the result in another deterministic variable:

```
tree := treeAssembly(topology, branch_lengths)
```

An alternative approach

Often, there is more than one way to do things in RevBayes. Above, we went through a 3-step process where we first drew from a *distribution of topologies*, then from a *distribution of branch lengths*, and finally assembled a phylogram out of both draws. We could simplify this approach by drawing directly from a *distribution of phylograms*:

```
tree ~ dnUniformTopologyBranchLength(taxa,
                                       outgroup=out_group,
                                       branchLengthDistribution=dnUniform(0, 5))
```

We can apply the same moves to our phylogram as we did to our topology:

```
moves.append( mvNNI(tree, weight=num_branches) )
moves.append( mvSPR(tree, weight=num_branches/10.0) )
```

Since we specified our branch lengths all at once instead of one by one (by means of the `branchLengthDistribution` argument of the `dnUniformTopologyBranchLength()` function), we also need a move that is capable of operating on all branch lengths at once:

```
moves.append( mvBranchLengthScale(tree, weight=num_branches) )
```

Finally, we can still set up our tree length variable, but we need to do it a bit differently:

```
tree_length := tree.treeLength()
```

At this point, the only thing we need to add to our overall model is the *substitution model*, which gives us the exact probabilities of going from one state to a different state along a branch of a given length. We'll use the same model we learned about when going over maximum likelihood, i.e., Paul Lewis's M*k* (Markov *k*-state) model, with the optional addition of ascertainment bias correction (M*k*v) and among-character rate variation (M*k*+Γ).

Let's start with the latter. You may remember from Lecture 10.3 that we usually assume that the rates of different characters follow a gamma (Γ) distribution, and that we set its rate β equal to its shape α, so that the mean $\frac{\alpha}{\beta}$ equals 1. This leaves us with α as the only parameter we need to estimate. We will therefore treat α as a stochastic variable, draw it from some prior distribution, and place a move on it:

```
# Let's use a broad uniform distribution from 0 to 1,000,000:
alpha ~ dnUniform(0, 1E6)
moves.append( mvScale(alpha, weight=2.0) )
```

You may also recall that to simplify the calculations, we break up – or "discretize" – the continuous gamma distribution into a number of rate categories (usually four). This is easy to do in RevBayes:

```
# The three arguments are: (1) shape, (2) rate, (3) number of categories
char_rates := fnDiscretizeGamma(alpha, alpha, 4)
```

Just like before, we want to make sure that each character receives a rate matrix of the right dimensions: a binary character should be described by a 2-by-2 matrix, a three-state character by a 3-by-3 matrix, etc. You still remember what a nightmare it was to convince IQ-TREE to do this for us. The good news is that with RevBayes, doing this is a lot easier. As long as we know the maximum number of character states in our matrix, we can iterate over different state numbers (2, 3, 4, etc.), and for each number i, we can extract all the i-state characters in our matrix and assign them an i-by-i rate matrix:

```
max_num_states <- 6

# Define a helper variable to count distinct rate matrices:
j = 1

# We will start from 2, because there are no constant (1-state) characters
# in the Tedford et al. matrix
for (i in 2:max_num_states) {
```

We will create temporary copies of our character matrix, which we can freely modify without the original data being affected. Each copy will form one element of a vector called `partitions`. To make sure that the 2-state partition forms the 1st element, the 3-state partition forms the 2nd element, etc., we will make sure that the index is one smaller than the current number of states:

```
    partitions[i - 1] <- morpho

    # Only keep those characters whose number of states equals i
    partitions[i - 1].setNumStatesPartition(i)
```

We could imagine a case in which our dataset contains, say, 2-state, 3-state, and 5-state characters, but no 4-state characters. To prevent our loop from stumbling over such a case, we will check whether the number of *i*-state characters is greater than zero, and only move on if it is:

```
        # How many i-state characters do we have?
        nc = partitions[i - 1].nchar()
        if (nc > 0) {
            Q[j] := fnJC(i)
```

What happened here? We created an M*k* rate matrix with $k = i$ states. From Paul Lewis's 2001 paper linked in Lab 5, you may remember that the M*k* model for morphological data was derived by generalizing an earlier Jukes-Cantor (JC) model for DNA sequences. This explains why the function for setting up an M*k* model is called `fnJC()` in RevBayes. It takes a single argument corresponding to the desired number of states, and since the M*k* model has no free parameters, its rate matrix is fully determined by this number, making it a deterministic variable. Once the rate matrix has been created, we assign it to the *j*-th element of a vector called `Q`.

Next, we are going to put it all together:

```
            characters[j] ~ dnPhyloCTMC(tree=tree, Q=Q[j], type="Standard",
                                        siteRates=char_rates,
                                        coding="variable")
```

A lot has happened here! We built a stochastic variable for the character data by drawing from a distribution called the *phylogenetic continuous-time Markov chain*, or `dnPhyloCTMC` for short. This distribution ties together everything we've specified so far: the tree topology with branch lengths (jointly specified by `tree`), the substitution model (specified through its rate matrix `Q`), and the model of how rates of change vary across characters (`char_rates`). We additionally told RevBayes that it would be dealing with morphological ("standard") data, and asked it to correct for ascertainment bias by specifying that only variable characters should be considered (`coding="variable"`).

Now we just have to clamp this variable to our observed character data:

```
            characters[j].clamp(partitions[i - 1])

            # Increment counter
            j = j + 1

            # Close the if-conditional and the for-loop
        }
    }
```

Ordered characters

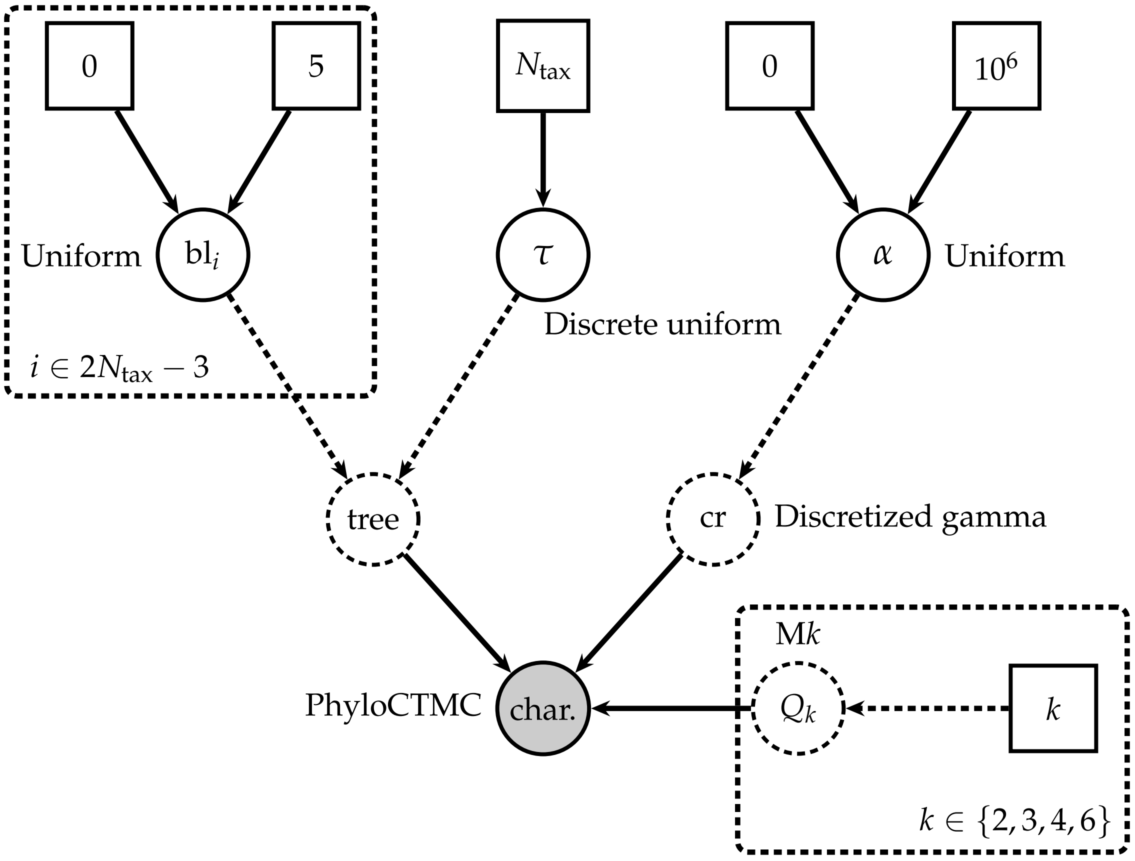
So far we have assumed that all our multistate characters are unordered, and as such correctly described by the M*k* model. This happens to be true of the Tedford et al. dataset we are using as an example, but other matrices may contain a mixture of ordered and unordered characters. This is how we would deal with such a situation in RevBayes:

```
ordered_char_indices <- [14, 23, 32, 38, 41, 57]

# Create temporary copies of the character data
ordered <- morpho
unordered <- morpho

# Drop all characters and put back only those that are ordered
ordered.excludeCharacter(1:morpho.nchar())
ordered.includeCharacter(ordered_char_indices)

# Drop the ordered characters but keep the rest
unordered.excludeCharacter(ordered_char_indices)

idx = 1
ix = 1

# Only multistate characters can be ordered, so we start iterating from 3
for (i in 3:max_num_states) {
    ordered_partitions[i - 2] <- ordered
    ordered_partitions[i - 2].setNumStatesPartition(i)
    nco = ordered_partitions[i - 2].nchar()
    if (nco > 0) {
        Q_ord[idx] := fnOrderedRateMatrix(i)
        char_ord[idx] ~ dnPhyloCTMC(tree=tree, Q=Q_ord[idx], type="Standard",
                                    siteRates=char_rates, coding="variable")
        char_ord[idx].clamp(ordered_partitions[i - 2])
        idx = idx + 1
    }
}

# Back to unordered characters, which can be binary: start iterating from 2
for (j in 2:max_num_states) {
    unordered_partitions[j - 1] <- unordered
    unordered_partitions[j - 1].setNumStatesPartition(j)
    nc = unordered_partitions[j - 1].nchar()
    if (nc > 0) {
        Q_unord[ix] := fnJC(j)
        char_unord[ix] ~ dnPhyloCTMC(tree=tree, Q=Q_unord[ix], type="Standard",
                                     siteRates=char_rates, coding="variable")
        char_unord[ix].clamp(unordered_partitions[j - 1])
        ix = ix + 1
    }
}
```

We're done! Now we just combine everything into a single model:

```
mymodel = model(tree)
```

Let's see if we can graph this overall model using the notation we introduced in Lab 6:



Here, N_{tax} is the number of taxa, bl_i is the i-th branch length, τ is the tree topology, "cr" are the character rates, and k is the number of states. For clarity, the stochastic variables are annotated with the type of prior distribution from which they were drawn. The dashed rectangles represent vectors, or more generally repetition: they tell us that, for example, the process of drawing a branch length from the uniform prior distribution is repeated $2N_{\text{tax}} - 3$ times, i.e., once for every branch in our tree.

At this point, we should define our monitors:

```
monitors.append( mnScreen(printgen=10) )
monitors.append( mnModel(filename="Tedford_phylo.log", printgen=10) )
monitors.append( mnFile(filename="Tedford_phylo.trees", printgen=10, tree) )
```

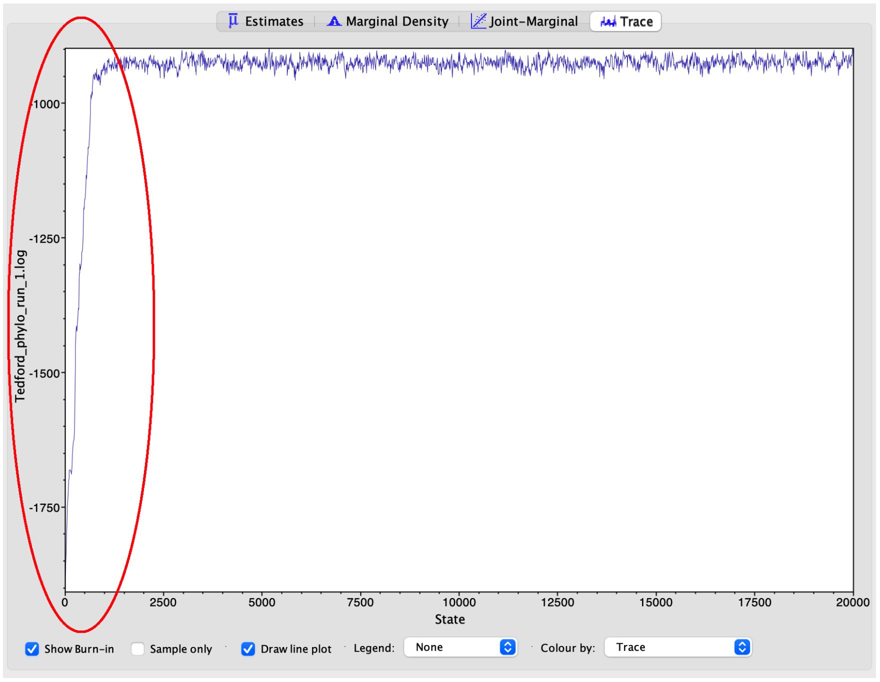
The first two of these should look familiar: they just print the values of continuous parameters (such as `alpha` or `tree_length`) to the screen or to a specified log file, respectively.

What's new here is the third monitor, which records the values of only those parameters that we pass in as arguments. In this case, the monitor will print trees annotated with branch lengths to a separate file.

One issue we need to address at this point is the fact that MCMC analyses start from different randomly picked initial points, and when exploring the parameter space, they may have trouble crossing the "valleys" of low posterior probability that separate the high-probability "peaks". As a result, there is a real danger that a single MCMC run may get stuck at a particular peak. However, as we will learn in upcoming lectures, the whole idea behind the MCMC is that the number of samples collected from a particular region of the parameter space is proportional to its posterior probability: for example, if 95% of sampled trees contain clade x, we would conclude that the posterior probability of clade x is 95%. If the simulation gets stuck, this will no longer hold true. To address this problem, we will run two MCMC analyses at the same time. If both of them converge on the same distribution (i.e., if they end up sampling very similar parameter values), we can be more confident that this distribution really is the posterior.

```
   mymcmc = mcmc(mymodel, monitors, moves, nruns=2)
```

It takes some time for the MCMC to converge on the posterior distribution: since the starting point is usually chosen at random, it is likely to be quite lousy, and the number of samples collected in the region around this starting point will likely be out of proportion to its posterior probability. If we were to plot the (log) posterior probability against the number of collected samples, we would find that during this initial wandering, the analysis climbs rapidly toward higher-probability regions:



In MCMC theory, we call this the "burnin", and since the samples collected during this period are not representative of the posterior distribution, we discard them. This can be done after the fact, but we can also tell RevBayes in advance to not even bother to collect samples from, say, the first 2000 iterations. We can also use this period to tune our proposals (see above):

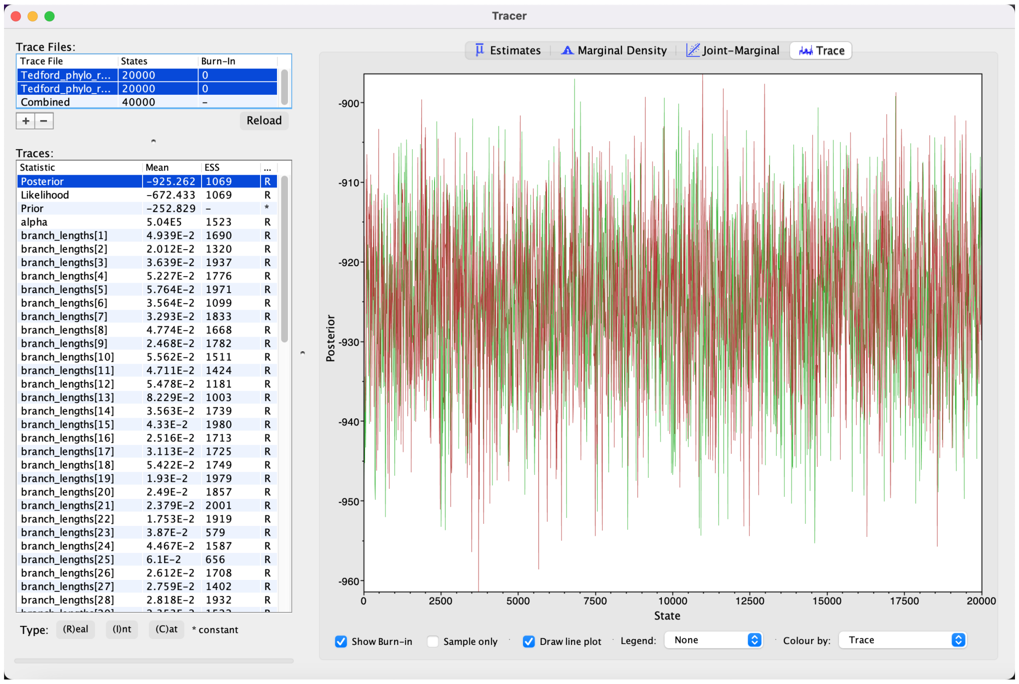
```
mymcmc.burnin(generations=2000, tuningInterval=10)
mymcmc.run(generations=20000)
```

For your convenience, all of the above code has been packaged into a single Rev script available from Canvas (`Tedford_phylo.Rev`).

The analysis we just set up takes about 20 minutes to finish. Once it's done, RevBayes will have created the following files in your working directory:

- `Tedford_phylo_run_1.log`
- `Tedford_phylo_run_1.trees`
- `Tedford_phylo_run_2.log`
- `Tedford_phylo_run_2.trees`

Load the two files with the `.log` extension into Tracer, highlight both of them in the top left pane, and make sure you've selected the "Trace" tab in the right pane. The trace shows the parameter values sampled over the course of the analysis:



Trace Files:

Trace File	States	Burn-In
Tedford_phylo_r...	20000	0
Tedford_phylo_r...	20000	0
Combined	40000	–

Traces:

Statistic	Mean	ESS	...
Posterior	-925.262	1069	R
Likelihood	-672.433	1069	R
Prior	-252.829	–	*
alpha	5.04E5	1523	R
branch_lengths[1]	4.939E-2	1690	R
branch_lengths[2]	2.012E-2	1320	R
branch_lengths[3]	3.639E-2	1937	R
branch_lengths[4]	5.227E-2	1776	R
branch_lengths[5]	5.764E-2	1971	R
branch_lengths[6]	3.564E-2	1099	R
branch_lengths[7]	3.293E-2	1833	R
branch_lengths[8]	4.774E-2	1668	R
branch_lengths[9]	2.468E-2	1782	R
branch_lengths[10]	5.562E-2	1511	R
branch_lengths[11]	4.711E-2	1424	R
branch_lengths[12]	5.478E-2	1181	R
branch_lengths[13]	8.229E-2	1003	R
branch_lengths[14]	3.563E-2	1739	R
branch_lengths[15]	4.33E-2	1980	R
branch_lengths[16]	2.516E-2	1713	R
branch_lengths[17]	3.113E-2	1725	R
branch_lengths[18]	5.422E-2	1749	R
branch_lengths[19]	1.93E-2	1979	R
branch_lengths[20]	2.49E-2	1857	R
branch_lengths[21]	2.379E-2	2001	R
branch_lengths[22]	1.753E-2	1919	R
branch_lengths[23]	3.87E-2	579	R
branch_lengths[24]	4.467E-2	1587	R
branch_lengths[25]	6.1E-2	656	R
branch_lengths[26]	2.612E-2	1708	R
branch_lengths[27]	2.759E-2	1402	R
branch_lengths[28]	2.818E-2	1932	R

Type: (R)eal (I)nt (C)at * constant

A well-behaved analysis, like the one shown above, should not exhibit any large-scale trends (no increase or decrease). Instead, the trace should resemble white noise or – as it's also sometimes described – a "hairy caterpillar" randomly oscillating around a more or less stable value. Moreover, the traces from both runs (here shown in red and green) should overlap: if they don't, your runs got stuck on different peaks. In our analysis, we should examine the traces for at least the following statistics: `Posterior`, `Likelihood`, `Prior`, `alpha`, the four elements of the `char_rates` vector, and `tree_length`. We should also check that their effective sample sizes (ESS) exceed 200 after combining both runs.

---

4) Re-run the analysis above on your pasta dataset. If all of this feels a bit overwhelming, don't worry. Start by examining the `Tedford_phylo.Rev` script. Think about which parts of the code are generally applicable, and which you'll need to tailor to your own data. (Hint: there are probably fewer of the latter than you might think!) Refer to the section on ordering if appropriate. After you're done, use the information above to assess whether or not your analysis has converged. Was the specified burnin sufficient, or are there more samples that you need to discard? Justify your answer.

This is all nice, but we still want to see the tree. However, we have $2 \times (20,000/10) =$ 4,000 of them. (If you don't see why, spend some time thinking about this.) This is a very important point: *the result of a Bayesian phylogenetic analysis is the posterior distribution of trees*. We can choose a single tree to represent this distribution, just like we often choose a single number – say, the mean – to represent the distribution of some continuous variable. However, that's just a summary of the result, not the result itself. There are several different summary trees we can compute. The strict consensus, which we used for maximum parsimony, is not often used in Bayesian inference, but the 50% majority-rule consensus remains pretty common. Here, we will explore two additional summaries:

1. To obtain the *maximum clade credibility* (MCC) tree, we first calculate the frequency at which every clade present in a given tree occurs in the entire posterior sample. As we learned above, this frequency approximates the clade's posterior probability. We then assign each tree a score equal to the product of the clade frequencies, and choose the tree with the highest score.

2. We can also simply pick the tree with the highest posterior probability. That would be the so-called *maximum a posteriori* (MAP) tree. It is analogous to the maximum likelihood tree, except that the quantity we are maximizing is no longer just the likelihood, but rather the likelihood multiplied by the prior.

We will first load the trees outputted by RevBayes back into the program, and assign them to a new type of workspace variable called "tree trace":

```
trace1 = readTreeTrace("Tedford_phylo_run_1.trees", treetype="non-clock",
                        burnin = 0)
trace2 = readTreeTrace("Tedford_phylo_run_2.trees", treetype="non-clock",
                        burnin = 0)
trace_combined = [trace1, trace2]
```

The `treetype="non-clock"` argument just tells RevBayes that we want the trees to be treated as phylograms (with branch lengths in expected changes per character) rather than chronograms (with branch lengths in units of calendar time).

Now it's just a matter of telling RevBayes where we want the summary trees to be printed:

```
mcc_tree = mccTree(trace=trace_combined, file="my_MCC_tree.tre")
map_tree = mapTree(trace=trace_combined, file="my_MAP_tree.tre")
```

---

5) Use FigTree to visualize your MCC and MAP trees. Expand the "Node Labels" menu and choose "posterior" from among the "Display" options: this will show you the posterior probability of each clade. Save an image of at least one of the two summary trees and insert it into your answer sheet. How does it compare to your most parsimonious and maximum-likelihood trees from the previous weeks? Briefly describe its topology and posterior probabilities.

6) Change the upper bound of the uniform prior on branch lengths from 5 to 0.5, and re-run the analysis. What happens? Save an image of your summary tree(s) – MAP, MCC, or both – and include it in your report. Is this tree "better" or "worse" than the original one?

7) In Lab 5, we played around with the branch lengths of our tree, treating them as equal, proportional, or completely independent among the individual partitions. We can do this in RevBayes as well. Based on your newly acquired knowledge of Rev, can you come up with a piece of code that would give each partition a separate set of branch lengths? Give it your best shot!
