Introduction to MCMC using `RevBayes`

A Simple Example for Building a Hierarchical Model and Sampling under Markov Chain Monte Carlo

Wade Dismukes, Tracy A. Heath, Walker Pett

1 Overview

This tutorial is intended to provide a introduction to the basics of Markov chain Monte Caro (MCMC) using the Metropolis-Hastings algorithm. This will provide a brief introduction to MCMC moves as well as prior distributions. We begin with a simple example of estimating the probability distribution of an archer's ability to shoot at a target, and the distance those arrows land from the center. We will simulate data using this example and attempt to estimate the posterior distribution using a variety of MCMC moves.

1.1 Requirements

1.1.1 Required Software

This tutorial requires that you download and install the latest release of `RevBayes` (Höhna et al. 2017), which is available for Mac OS X, Windows, and Linux operating systems. Directions for downloading and installing the software are available on the program webpage: http://revbayes.com.

The exercise provided also requires additional programs for editing text files and visualizing output. The following are very useful tools for working with `RevBayes`:

- A good text editor – if you do not already have one that you like, we recommend one that has features for syntax coloring, easy navigation between different files, line numbers, etc. Good options include Sublime Text or Atom, which are available for Mac OSX, Windows, and Linux.
- Tracer – for visualizing and assessing numerical parameter samples from `RevBayes`

2 Modeling an Archer's Shots on a Target

We'll begin our exploration of Bayesian inference with a simple archery model. For this model, there is an unknown archer shooting n arrows at a target. The distance d of each arrow from the target's center is measured. Let's assume that the distance of each arrow from the bullseye follows an exponential distribution—i.e., $d \sim \text{Exp}(\mu^{-1})$. Further, we assume the archer has an inherent ability to shoot arrows at an average distance μ. Then, the probability density of each arrow distance d_i is

$$P(d_i \mid \mu) = \frac{1}{\mu} e^{-x_i/\mu}$$

Simple intuition suggests that, given that we observe n arrows, the maximum-likelihood estimate (MLE) of μ is simply the average of all the arrow distances $\bar{d} = \frac{1}{n} \sum_{i=1}^{n} d_i$. This is indeed the maximum likelihood

Figure 1: placeholder figure for a representation of the model
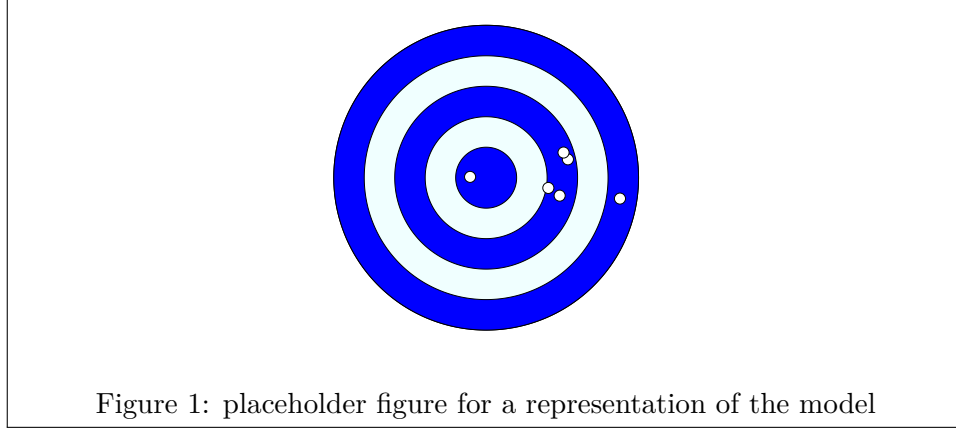
estimate! In fact, given n arrows whose distances follow an exponential distribution, it turns out that the observed average $\bar{d}$ follows a gamma distribution, with parameters n and n/μ.

$$P(\bar{d} \mid \mu, n) = \frac{(n/\mu)^n}{\Gamma(n)} \bar{d}^{n-1} e^{-n\bar{d}/\mu}$$

In this case, the average $\bar{d}$ acts as a *sufficient statistic* for μ. This means that it tells us everything we need to know about μ. Therefore, we will use a $\text{Gamma}(n, n\mu)$ distribution on $\bar{d}$ as the likelihood of our data.

From Bayes' theorem, the *posterior distribution* of μ given $\bar{d}$, $P(\mu \mid \bar{d})$, is:

$$\overbrace{P(\mu \mid \bar{d})}^{\text{posterior distribution}} = \frac{\overbrace{P(\bar{d} \mid \mu)}^{\text{likelihood}} \times \overbrace{P(\mu)}^{\text{prior}}}{\underbrace{P(\bar{d})}_{\text{marginal likelihood}}}$$

The take-home message here is that, if we're interested in doing Bayesian inference for the archery model, we need to specify a *likelihood function* and a *prior distribution* for μ. In virtually all practical cases, we cannot compute the posterior distribution directly and instead use numerical procedures, such as a Markov chain Monte Carlo (MCMC) algorithm. Therefore, we will also have to write an MCMC algorithm that samples parameter values in the frequency of their posterior probability.

We'll use a simple exponential distribution as a prior on the parameter of the model, μ. The exponential distribution has one parameter α representing our prior belief about the mean arrow distance (Figure 2). Different choices for α represent different prior beliefs.

Figure 3 shows the graphical model for the archery model. This nicely visualizes the dependency structure in the model. We see that the parameter α is drawn in a solid square, representing that this variable is constant. From this variables, we see an arrow going into the variable μ. That simply means that μ depends on α. More specifically, μ is a stochastic variable (shown as a solid circle) and drawn from an exponential distribution with parameter α. Then, we have another constant variable, n. Finally, we have the observed data $\bar{x}$ which is drawn from a gamma distribution with parameters μ and n, as can be seen by the arrows going into x. Furthermore, the solid circle of $\bar{x}$ is shaded which means that the variable has data attached to it.
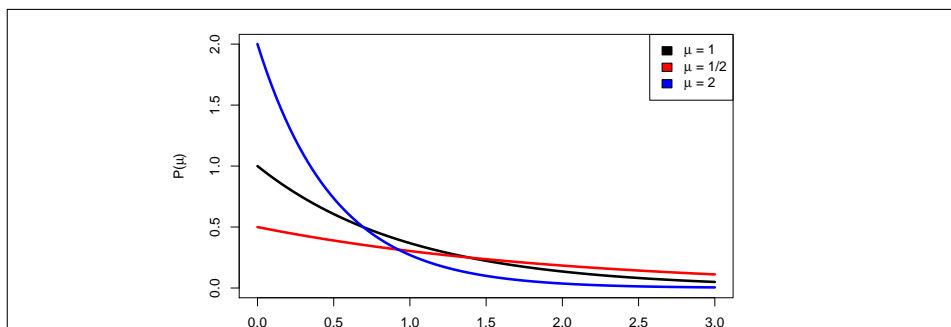
3   Writing an MCMC from Scratch

Figure 2: An exponential distribution with one parameter α. This distribution is used as a prior distribution on the average arrow distance μ. Here we show different curves for the exponential distribution when using different parameters.
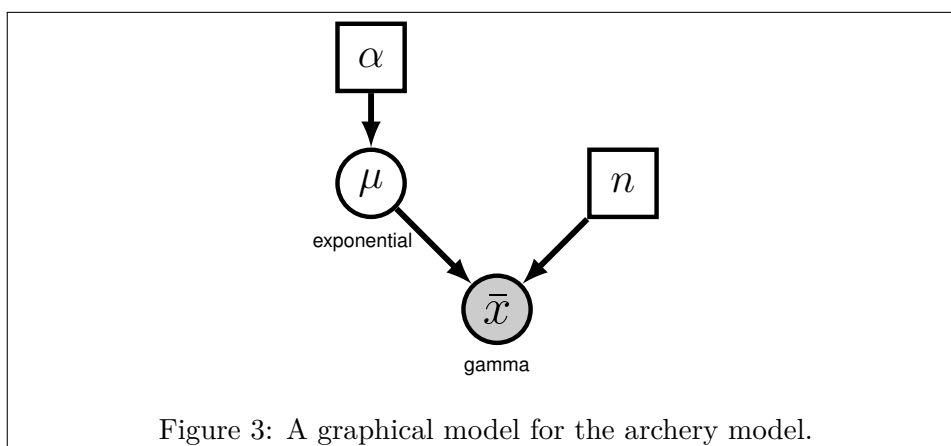

Figure 3: A graphical model for the archery model.

3.1 Tutorial Format

This tutorial follows a specific format for issuing instructions and information.

> The boxed instructions guide you to complete tasks that are not part of the RevBayes syntax, but rather direct you to create directories or files or similar.

Information describing the commands and instructions will be written in paragraph-form before or after they are issued.

All command-line text, including all Rev syntax, are given in **monotype font**. Furthermore, blocks of Rev code that are needed to build the model, specify the analysis, or execute the run are given in separate shaded boxes. For example, we will instruct you to create a new variable called **n** that is equal to **10** using the **=** operator like this:

```
n = 10
```

It is important to be aware that some PDF viewers may render some characters given as `Rev commands` differently. Thus, if you copy and paste text from this PDF, you may introduce some incorrect characters. Because of this, we recommend that you type the instructions in this tutorial or copy them from the scripts provided.

> Make yourself familiar with the example script called **archery_MH.Rev** which shows the code for the following sections. Then, start a new and empty script in your text editor and follow each step provided as `Rev commands` below.

3.2   The Metropolis-Hastings Algorithm

Though `RevBayes` implements efficient and easy-to-use Markov chain Monte Carlo algorithms, we'll begin by writing one ourselves to gain a better understanding of the moving parts. The Metropolis-Hastings MCMC algorithm (Metropolis et al. 1953; Hastings 1970) proceeds as follows:

1. Generate initial values for the parameters of the model (in this case, μ).

2. Propose a new value (which we'll call μ') for some parameters of the model, (possibly) based on their current values

3. Calculate the acceptance probability, R, according to:

$$R = \min\left\{1, \frac{P(\bar{d} \mid \mu')}{P(\bar{d} \mid \mu)} \times \frac{P(\mu')}{P(\mu)} \times \frac{q(\mu)}{q(\mu')}\right\}$$

4. Generate a uniform random number u between 1 and 0.

   if $u < R$: then accept the move and set $\mu = \mu'$.

   else (if $u \geq R$): the value of mu does not change and the move is rejected: $\mu = \mu$.

5. Record the values of the parameters.

6. Return to step 2 many, many times, keeping track of the value of μ.

3.3   Reading in the data

Since we do not have access to archery data, we will simulate the the shots of our archer using the simulation tools in `RevBayes`. By simulating the data, we can also evaluate how well our moves and prior model perform—*i.e.,* how robust and accurate are our estimators. After completing this exercise, feel free to repeat it and alter the true values to see how they influence the posterior distribution.

```
# Simulate some data (i.e. shoot some arrows)
# First we need the number of arrows to shoot
n = 10
# Then we need a true mean distance
```

```
mu_true = 1
# Simulate the observed mean distance of the arrows shot from a gamma distribution
arrow_mean = rgamma(1, n, n/mu_true)[1]
```

The Rev code above uses the **rgamma()** function to simulate a single observed **arrow_mean** from $n = 10$ arrows shot on target. The **[1]** following the **rgamma()** function is needed because this function always returns a *vector* even when we only request a single value. Thus, in order to treat **arrow_mean** as a single value, we have to request the first element of the vector returned by that function.

3.4   Initializing the Markov chain

We have to start the MCMC off with some initial values for all of the parameters. One way to do this is to randomly draw values of the parameters (just μ, in this case) from the prior distribution. We'll assume a simple exponential prior distribution; that is, one with $\alpha = 1$.

```
# Initialize the chain with some starting value
alpha = 1.0
mu = rexp(1, alpha)[1]
```

3.4.1   Likelihood function

Next we will specify the likelihood function, which will compute the probability of our data given the prior model. We use the gamma distribution for the likelihood. Since the likelihood is defined only for values of μ greater than 0, we return a likelihood of 0.0 if μ is negative:

```
# Define the likelihood function on the mean
function likelihood(mu){
    if(mu < 0.0)
        return 0.0

    return dgamma(arrow_mean, n, n/mu, log=false)
}
```

In Rev, we can create a *user-defined function* using the **function** keyword. In our function definition above, **likelihood()** takes a single value as an argument that is expected to be the mean (μ) value. All other parameters in our function are expected to be defined before **likelihood()** is called.

3.4.2   Prior distribution

Similarly, we need to define a function for the prior distribution. Here, we use the exponential probability distribution for the prior on μ:

```
# Define the prior function on the mean
function prior(mu){
    if(mu < 0.0)
        return 0.0

    return dexp(mu, alpha, log=false)
}
```

3.4.3 Monitoring parameter values

Additionally, we are going to monitor, *i.e.,* store, parameter values into a file during the MCMC simulation. For this file we need to write the column headers in the first line of our output file, which we will name **archery_MH.log** (you may have to change the newline characters from **"\n"** to **"\r\n"** if you're using a Windows operating system.):

```
# Prepare a file to log our samples
write("iteration","mu","\n",file="archery_MH.log")
write(0,mu,"\n",file="archery_MH.log",append=TRUE)
```

We'll also monitor the parameter values to the screen, so let's print the initial values:

```
# Print the initial values to the screen
print("iteration","mu")
print(0,mu)
```

3.5 Writing the MH Algorithm

At long last, we can write our MCMC algorithm. First, we define how often we print to file (*i.e.,* monitor); this is called thinning if we do not choose to save every value of our parameter to file. If we set the variable **printgen=1**, then we will store the parameter values at every iteration; if we instead choose **printgen=10**, then we'll only save the values every 10th step in our Markov chain.

```
printgen = 10
```

We will repeat this resampling procedure many times and iterate the MCMC using a **for** loop (*e.g.,* step 6 in Sect. 3.2). We will start this part by defining the number of iterations for our MCMC (**reps = 1000**) and writing the first line of our **for** loop.

```
# Write the MH algorithm
reps = 10000
for(rep in 1:reps){
```

In `Rev`, the contents of every **for** loop must be enclosed within a set of 'curly braces' **{...}**. Our loop will not be complete until we finish it and add the closing brace. Additionally, it is good style to make our loops readable by indenting the contents within the **{...}**. We recommend that you use 4 spaces to represent these indents.

For our MCMC algorithm, the first thing we do is generate a new value of μ' to evaluate (step 2 of Sect. 3.2). We'll simply propose a new value of μ from a the prior distribution. Note that in this first example we do not condition new parameter values on the current value.

```
# Propose a new value of p
mu_prime <- rexp(n=1, alpha)[1]
```

Next, we compute the proposed likelihood and prior probabilities using the functions we defined above, as well as the acceptance probability, R (step 3 of Sect. 3.2):

```
# Compute the acceptance probability
R = ( likelihood(mu_prime)/likelihood(mu) ) * ( prior(mu_prime)/prior(mu) )
```

Then, we accept the proposal with probability R and reject otherwise (step 4 of Sect. 3.2):

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

Finally, we store the current value of μ in our log file (step 5 of Sect. 3.2). Here, we actually check if we want to store the value during this iteration.

```
    if ( (rep % printgen) == 0 ) {
        # Write the samples to a file
        write(rep,mu,"\n",file="archery_MH.log",append=TRUE)
        # Print the samples to the screen
        print(rep,mu)
    }
} # end MCMC
```
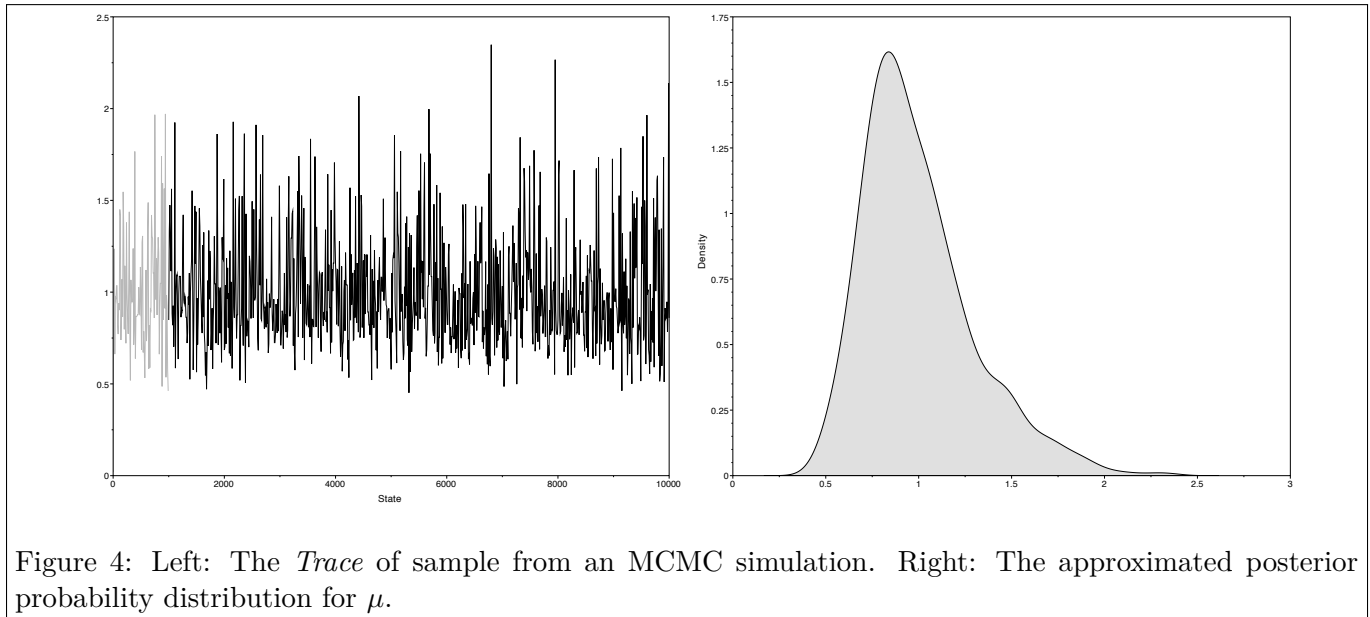
3.6 Exercise 1

Step 1) Write and execute this script, which you can give any name you like (there is also an example file called **archery_MH.Rev**).

Step 2) The `.log` file will contain samples from the posterior distribution of the model! Open the file in `Tracer` to learn about various features of the posterior distribution, for example: the posterior mean or the 95% credible interval.

Pretty awesome, right?

Below we show an example of the obtained output in `Tracer`. Specifically, Figure 4 shows the sample trace (left) and the estimated posterior distribution of μ (right). There are other parameters, such as the posterior mean and the 95% HPD (highest posterior density) interval, that you can obtain from `Tracer`.

Figure 4: Left: The *Trace* of sample from an MCMC simulation. Right: The approximated posterior probability distribution for μ.

4 More on Moves: Tuning and Weights

In the previous example we hard coded a single move updating the variable μ by drawing a new value from the prior. There are actually many other ways how to propose new values; some of which are more efficient than others.

First, let us rewrite the MCMC loop so that instead we call a function, which we name **move_prior** for simplicity, that performs the move:

```
for (rep in 1:reps){

    # call uniform move
    move_prior(1)

    if ( (rep % printgen) == 0 ) {
        # Write the samples to a file
        write(rep,p,"\n",file="archery_MH.log",append=TRUE)
        # Print the samples to the screen
        print(rep,mu)
    }

} # end MCMC
```

4.1 Prior move

Now we need to actually write the **move_prior** function. We mostly just copy the code we had before into a dedicated function

```
function move_prior(weight) {

    for (i in 1:weight) {
        # Propose a new value of mu
        mu_prime <- rexp(n=1, alpha)[1]

        # Compute the acceptance probability
        R = ( likelihood(mu_prime)/likelihood(mu) ) * ( prior(mu_prime)/prior(mu) )

        # Accept or reject the proposal
        u = runif(1,0,1)[1]
        if(u < R){
            # Accept the proposal
            mu = mu_prime
        }
    }
}
```

There are a few things to consider in the function **move_prior**. First, we do not have a return value because the move simply changes the variable μ if the move is accepted. Second, we expect an argument called **weight** which will tell us how often we want to use this move. Otherwise, this function does exactly the same what was inside the for loop previously.

(Note that you need to define this function before the for loop in your script).

4.2 Sliding-window move

As a second move we will write a sliding-window move. The sliding-window moves propose an update by drawing a random number from a uniform distribution and then adding this random number to the current value (*i.e.,* centered on the previous value).

```
function move_slide(delta, weight) {

    for (i in 1:weight) {
        # Propose a new value of p
        mu_prime <- mu + runif(n=1,-delta,delta)[1]

        # Compute the acceptance probability
        R = ( likelihood(mu_prime)/likelihood(mu) ) * ( prior(mu_prime)/prior(mu) )

        # Accept or reject the proposal
        u = runif(1,0,1)[1]
        if(u < R){
            # Accept the proposal
            mu = mu_prime
        }
    }
}
```

In addition to the weight of the move, this move has another argument, **delta**. The argument **delta** defines the width of the uniform window from which we draw new values. Thus, if **delta** is large, then the proposed values are more likely to be very different from the current value of μ. Conversely, if **delta** is small, then the proposed values are more likely to be very close to the current value of μ.

> Experiment with different values for **delta** and check how the effective sample size (ESS) changes.

There is, *a priori*, no good method for knowing what values of **delta** are most efficient. However, there are some algorithms implemented in RevBayes, called *auto-tuning*, that will estimate good values for **delta**.

4.3 Scaling move

As a third and final move we will write a scaling move. The scaling move proposes an update by drawing a random number from a uniform(-0.5,0.5) distribution, exponentiating the random number, and then

multiplying this scaling factor by the current value. An interesting feature of this move is that it is not symmetrical and thus needs a Hastings ratio. The Hastings ratio is rather trivial in this case, and one only needs to multiply the acceptance rate by the scaling factor.

```
function move_scale(lambda, weight) {

    for (i in 1:weight) {
        # Propose a new value of p
        sf <- exp( lambda * ( runif(n=1,0,1)[1] - 0.5 ) )
        mu_prime <- mu * sf

        # Compute the acceptance probability
        R = ( likelihood(mu_prime)/likelihood(mu) ) * ( prior(mu_prime)/prior(mu) )

        # Accept or reject the proposal
        u = runif(1,0,1)[1]
        if(u < R){
            # Accept the proposal
            mu = mu_prime
        }
    }
}
```

As before, this move has a tuning parameter called `lambda`.

> ! The sliding-window and scaling moves are very common and popular moves in RevBayes. The code examples here are actually showing the exact same equation as implemented internally. It will be very useful for you to understand these moves.

4.4 Exercise 2

Step 1) Rewrite your previous script to include these three different moves now.

Step 2) Then, run the script to estimate the posterior distribution of μ again.

Step 3) Look at the output in `Tracer`.

Step 4) Are the distributions, mean and credible interval the same?

Step 5) Use only a single move and set `printgen=1`. Which move has the best ESS?

Step 6) How does the ESS change if you use a `delta=10` for the sliding-window move?

Step 7) Have a look at how the acceptance rate changes for different values of the tuning parameters.

However, this MCMC algorithm is *very* specific to our archery model and thus hard to extend (also it's pretty inefficient!).

5 The Metropolis-Hastings Algorithm with the *Real* RevBayes

We'll now specify the exact same model in `Rev` using the built-in modeling functionality and moves. It turns out that the `Rev` code to specify the above model is extremely simple and similar to the one we used before. Again, we start by "reading in" (*i.e.*, making up) our data.

```
# Simulate some data (i.e. shoot some arrows)
# First we need the number of arrows to shoot
n = 10
# Then we need some true mean distance
mu_true = 1
# Simulate the observed mean distance of the arrows we shot
arrow_mean = rgamma(1, n, n/mu_true)[1]
```

Now we specify our prior model.

```
# Specify the prior distribution
alpha <- 1.0
mu ~ dnExponential(alpha)
```

One difference between `RevBayes` and the MH algorithm that we wrote above is that many MCMC proposals are already built-in, but we have to specify them *before* we run the MCMC. We usually define (at least) one move per parameter immediately after we specify the prior distribution for that parameter.

```
# Define a move for our parameter, mu
moves[1] = mvSlide(mu, delta=1, weight=1.0)
```

Next, our likelihood model.

```
# Specify the likelihood model
d_bar ~ dnGamma(n, n/mu)
d_bar.clamp(arrow_mean)
```

We wrap our full Bayesian model into one model object (this is a convenience to keep the entire model in a single object, and is more useful when we have very large models):

```
# Construct the full model
my_model = model(mu)
```

We use "monitors" to keep track of parameters throughout the MCMC. The two kinds of monitors we use here are the `mnModel`, which writes parameters to a specified file, and the `mnScreen`, which simply outputs some parts of the model to screen (as a sort of progress bar).

```
# Make the monitors to keep track of the MCMC
monitors[1] = mnModel(filename="archery_RB.log", printgen=10)
monitors[2] = mnScreen(printgen=1000, mu)
```

Finally, we assemble the analysis object (which contains the model, the monitors, and the moves) and execute the run using the `.run` command:

```
# Make the analysis object
analysis = mcmc(my_model, monitors, moves)

# Run the MCMC
analysis.run(100000)

# Show how the moves performed
analysis.operatorSummary()
```

Open the resulting **archery_RB.log** file in `Tracer`.

? Do the posterior distributions for the parameter *mu* look the same as the ones we got from our first analysis?

Hopefully, you'll note that this `Rev` model is substantially simpler and easier to read than the MH algorithm script we began with. Perhaps more importantly, this `Rev` analysis is *orders of magnitude* faster than our own script, because it makes use of extremely efficient probability calculations built-in to `RevBayes` (rather than the ones we hacked together in our own algorithm).

5.1 Exercise 3

Step 1) Run the built-in MCMC and compare the results to your own MCMC. Are the posterior estimates the same? Are the acceptance rates of the moves similar?

Step 2) Add a second move `moves[2] = mvScale(mu,lambda=0.1,weight=1.0)`

Step 3) Run the analysis again and compare the output.

6 Influence of the Prior

So far we have used a fairly simple exponential prior with $\alpha = 1$. However, we have not explored what impact this prior has on our estimate of μ, or whether it is an appropriate prior distribution. If the prior is very informative, then our posterior distribution will be relatively similar to our prior beliefs. In order

to explore the informativeness of the prior, we can change the true value of μ so that it is very different from our prior belief. If we are still able to recover the correct value of μ, then we can say that our prior is fairly uninformative.

If we find that our prior distribution is very informative, we have two options for minimizing sensitivity to the prior. First, we can use a less informative prior distribution. For example, since our data is exponentially distributed, a good choice for an uninformative prior is a Gamma(0,0) distribution (this is the Jeffreys prior). Unfortunately, this prior distribution is *improper* (it does not integrate to 1), and so we can't use it in RevBayes. However we can approximate this prior distribution by using very small parameter values, e.g. Gamma(0.001, 0.001). As you can see in Figure 5, compared to the exponential distribution, the Gamma(0.001, 0.001) distribution is much more "flat".
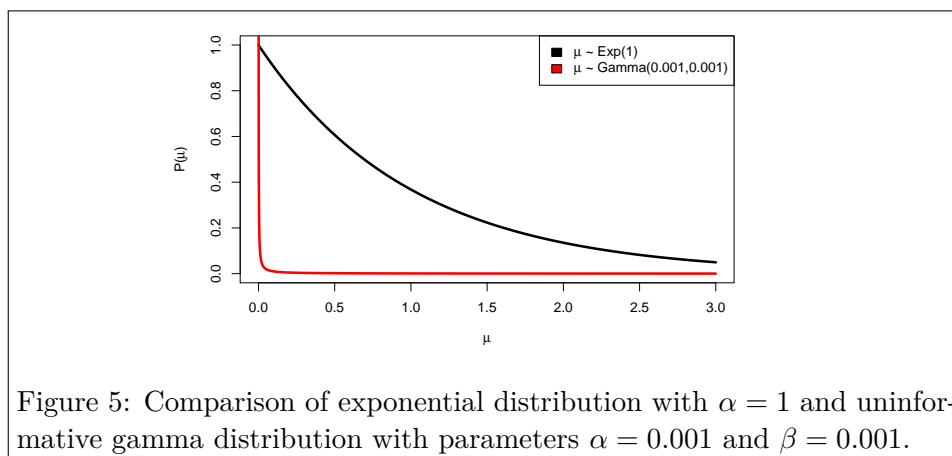

Figure 5: Comparison of exponential distribution with $\alpha = 1$ and uninformative gamma distribution with parameters $\alpha = 0.001$ and $\beta = 0.001$.

The second and simplest way we can overcome the informativeness of the prior is to increase the amount of data we collect. We can do that in our example by increasing the number of arrows we shoot.

6.1 Exercise 4

Step 1) Increase the true mean arrow distance so that it is significantly larger than α. How does this impact your estimate of μ?

Step 2) Now use an uninformative Gamma$(0.001, 0.001)$ prior for μ. Did your estimate of μ improve?

Step 3) Increase the number of arrows shot. How does this change the shape and scale of the posterior distribution?

References

Hastings, W. K. 1970. Monte Carlo sampling methods using Markov chains and their applications. Biometrika 57:97–109.

Höhna, S., M. J. Landis, and T. A. Heath. 2017. Phylogenetic inference using RevBayes. Current Protocols in Bioinformatics .

Metropolis, N., A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. 1953. Equation of state calculations by fast computing machines. Journal of Chemical Physics 21:1087–1092.

Version dated: September 20, 2017