MCMC Algorithms

Patrick Ford, FCAS CSPA

January 2018

Introduction

Applying Bayesian models became practical with the development of Markov Chain Monte Carlo (MCMC) methods combined with the advent of increased computing power beginning in the 1990's. There are different ways to implement MCMC. This study note will cover three popular MCMC algorithms that estimate the posterior distributions of parameters in a Bayesian MCMC model:

- Metropolis
- Gibbs
- Hamiltonian

The study note aims to provide simplified algorithm implementations that illustrate what happens in the background when one of them in invoked. In doing so, the study note intends to illuminate the algorithmic machinery in hopes of providing a level of insight that could not be attained by studying theory alone. In fact, we eschew almost all of the theory in favor of application in this study note. We employ `R` software as a medium to explore the algorithms and diagnostics common to MCMC implementations. All of the programs can be executed with base `R` functionality which means that there is no set-up overhead required to run the code blocks other than a working version of `R`. `R` is freely available online for all common operating systems.

This study note supplements the syllabus readings found in the book *Statistical Rethinking*[1] by Richard McElreath. However, this study note can be read and understood even if the reader is not familiar with *Statistical Rethinking*. It is only in the next two sections where we highlight the shortcomings of alternatives to MCMC and extend the example of King Markov that the connection to the book is made explicit.

Beyond Grid and Quadratic Approximation

Recall that the main shortcoming of grid approximation as a technique to compute the posterior distributions of a vector of parameters is that it scales very poorly in high dimensions. Suppose, for example, that our model has 7 continuous parameters and in order to feel confident that we represent their posterior distributions precisely enough, we create a finite grid of 1,000 equally-spaced values to represent potential realizations at those values for each of the 7 parameters. This means we would need to comb over $1e^{21}$ possible parameter combinations in order to approximate the posterior. Obviously that is too many computations for most computers to finish in a reasonable amount of time, if at all. We see that grid approximation does not scale well with the number of parameters and thus is not tractable for most problems.

We might also remember that the method of quadratic approximation computationally scales better than grid approximation, but it too can struggle in the face of complex, hierarchical models. It also does not fare well in the presence of posterior distributions that cannot be well approximated by a Gaussian distribution. Scale parameters, especially those with a lot of posterior density near a boundary, such as a small variance parameter that is close to zero, tend to not be served well by quadratic approximation.

Given the shortcomings of grid and quadratic approximation, we turn to MCMC sampling algorithms. The overarching idea of MCMC is that if we design a carefully-considered sampling strategy, we can feel confident that the sample distribution is representative enough of the true target posterior distribution to make statistical inferences. This idea is not too conceptually different than a political or social scientist sending out a survey or poll to a representative sample of respondents to estimate a population response. MCMC sampling algorithms don't draw from the posterior distribution directly similar to how a survey might be designed to not ask the question(s) of most immediate interest but rather ask related questions that are more

[1] McElreath, R., *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*, CRC Press, 2016

easily answered. Instead MCMC algorithms request simpler responses of the data - through the model - such that when all the simpler responses are combined, we often arrive at the answer of true interest - the posterior distribution. We will try to keep this idea in mind as we work through the MCMC sampling strategies in the remainder of the study note.

Revisiting King Markov's Islands

Statistical Rethinking uses the analogy of the benevolent King Markov to introduce the first MCMC algorithm - the **Metropolis algorithm**. Recall the main tenets of this example:

- The islands represent candidate parameters.
- The islands' relative populations represent the true posterior probabilities of the islands, which as we remember, is not known fully known by King Markov at any one time.
- The time spent on the islands represents samples taken from the posterior of the islands.

To further generalize this heuristic, conceptualize a string of tiny islands that are linked together such that King Markov can move to any island from any other island. To decide which tiny island to visit next, which we denote as θ_{prop}, King Markov will need a more general **proposal distribution** than only being able to move to adjacent islands. Such a proposal distribution should give him the potential to visit any island from any other island. The Metropolis algorithm calls for this proposal distribution to be a normal distribution centered at the current parameter value, θ_{curr}, and a user-specified variance, σ^2. In mathematical notation:

$$\theta_{prop} \sim Normal(\theta_{curr}, \sigma^2)$$

The proposal distribution is a separate and distinct distribution from either the prior or posterior distribution for the parameter. The proposal distribution's sole purpose is to give candidate parameter values to *try* and potentially accept as a valid sample from the posterior distribution of θ.

By *try* in the preceding paragraph we technically mean "insert the proposed parameter value, θ_{prop}, into the numerator of Bayes' formula and compare the value of the numerator in that formula to the value of the numerator for the current parameter value, θ_{curr}." After this computational step of calculating numerators, we put some simple logic around whether to accept or reject θ_{prop}. Instead of comparing seashells and stones like King Markov, our logic will be to automatically accept if the numerator evaluated at θ_{prop} is greater than the numerator evaluated at θ_{curr} and *probabilistically* accept if the numerator of Bayes' formula at θ_{prop} is less than the numerator of Bayes' formula at θ_{curr}. In less words:

$$p(accept\ \theta_{prop}) = min(\frac{Bayes'\ numerator\ at\ \theta_{prop}}{Bayes'\ numerator\ at\ \theta_{curr}},\ 1.0)$$

And Bayes' numerator is simply the product of the likelihood, $l(Data|\theta)$, and the prior distribution, $p(\theta)$. We will also call Bayes' numerator the *unnormalized posterior* since it is proportional to the true posterior density up to a constant. So the probability of accepting θ_{prop}, which is a realization from the normally distributed proposal distribution, is:

$$p(accept\ \theta_{prop}) = min(\frac{l(Data|\theta_{prop})p(\theta_{prop})}{l(Data|\theta_{curr})p(\theta_{curr})},\ 1.0)$$

Metropolis Algorithm

In practice, how do we actually figure out when to accept θ_{prop}? We always accept if $\frac{l(Data|\theta_{prop})p(\theta_{prop})}{l(Data|\theta_{curr})p(\theta_{curr})}$ is greater than or equal to 1.0, and, if not, we sample from a uniform random variable, $r \sim U(0,1)$, and accept when r is less than $p(accept\ \theta_{prop})$. Notice a couple of things about this algorithm:

- By taking the ratio of the numerator of Bayes' formula at the proposed and current parameter values, we have circumvented having to compute the difficult denominator in Bayes' formula (typically referred to as the evidence or average likelihood).
- We always accept proposed parameter values that have a higher posterior density than the current parameter value, and we even accept some proposed values that have lower posterior density just like King Markov would sometimes leave a more populated island for a less populated island.
- If θ_{prop} could not have possibly generated the data we're analyzing, then the likelihood, $l(Data|\theta_{prop})$, would equal zero and so would $p(accept\ \theta_{prop})$.
- More interestingly, if the prior distribution selected has no density at θ_{prop} then $p(\theta_{prop})$ would equal zero and so would $p(accept\ \theta_{prop})$. Thus it would behoove us to always remember that if it's not possible to realize a parameter value from the prior distribution, it will not be possible to realize it from the posterior distribution.

To recap, here's the three things we must be able to do to perform MCMC sampling with the Metropolis algorithm for a single parameter.

1. We must be able to generate a random value from the proposal distribution, $Normal(\theta_{curr}, \sigma^2)$.
2. We must be able to calculate the unnormalized posterior densities (the numerators in Bayes' formula) at both the proposed and current parameter values.
3. We must be able to generate a uniform random value from 0 to 1 to accept or reject the proposed parameter value.

Let's build our own Metropolis algorithm to sample from the posterior of a single parameter. Here's the motivation:

- We flip a coin a number of times to find out if it's fair. Let's call the parameter θ the *fairness factor* as it estimates the true probability that the coin will land with its head-side up. A fair coin would have $\theta = 0.5$.
- From 41 independent flips, we observe 13 heads. This is our $Data$.
- Flipping a coin successive times represents the sum of independent Bernoulli trials, which means our likelihood function is binomial:

$$Data \sim Binom(n = 41, p = \theta)$$

- The only missing component to Bayes' formula is the *prior* distribution. We'll do the most natural thing and select a beta distribution which is conjugate prior to the binomial likelihood. We do this so we can compare our sampled posterior via Metropolis MCMC against the true posterior distribution which we can compute analytically. The prior we'll employ for the job is:

$$p(\theta) \sim \beta(a = 2, b = 2)$$

- The prior distribution we choose is non-informative relative to the 41 observed flips. Selecting that prior is tantamount to claiming "in the past I've observed 4 coin flips for this coin, and 2 of the four came up heads". But we now have 41 real-life flips and only 13 were heads so the observed data should overwhelm our weak prior.
- Our proposal distribution for the Metropolis algorithm will be $Normal(\theta_{curr}, 0.05^2)$.
- θ represents a probability and so its values are bounded between 0 and 1. Therefore, we must reject any samples from the proposal distribution that fall outside of this range.This can occur because the proposal distribution is normal and thus defined for all values on the real number line.

Now let's codify these statements in `R` to see how it works. First we encode our data and `a` and `b` parameters of the prior beta distribution.

```
flips <- 41; heads <- 13
a <- 2; b <- 2
```

Then we create a function called `metropolis_algorithm` that takes three arguments. `samples` is the number of samples we want to draw from the posterior distribution and determines the length of the resulting MCMC

chain. `theta_seed` gives us a θ to start the algorithm, and `sd` is the standard deviation of the proposal distribution. Within the function we construct a `for` loop that repeatedly draws θ_{prop} from the standard normal proposal distribution (`rnorm`) and then computes the ratio of Bayes' numerators and finally carries out the accept/reject logic. We store the results in an `R` vector called `posterior_thetas` which we initialize to be `NAs` before we start the loop. To aid in comprehension, many of the code statements in this study note are annotated with comments. Comments begin with `#` and are either placed above or, if space allows, next to the statements they describe.

```
metropolis_algorithm <- function(samples, theta_seed, sd){

  theta_curr <- theta_seed
  posterior_thetas <- rep(NA, samples) # Create vector to store sampled parameters

  for (i in 1:samples){

    theta_prop <- rnorm(n = 1, mean = theta_curr, sd = sd) # Proposal distribution

    # If the proposed parameter is outside its range then set it equal to its current
    # value. Otherwise keep the proposed value
    theta_prop <- ifelse((theta_prop < 0 | theta_prop > 1), theta_curr, theta_prop)

    # Bayes' numerators
    posterior_prop <- dbeta(theta_prop, a, b) * dbinom(heads, flips, theta_prop)
    posterior_curr <- dbeta(theta_curr, a, b) * dbinom(heads, flips, theta_curr)

    # Calculate probability of accepting
    p_accept_theta_prop <- min(posterior_prop/posterior_curr, 1.0)

    rand_unif <- runif(n = 1)

    # Probabilistically accept proposed theta
    theta_select <- ifelse(p_accept_theta_prop > rand_unif, theta_prop, theta_curr)

    posterior_thetas[i] <- theta_select

    # Reset theta_curr for the next iteration of the loop
    theta_curr <- theta_select
  }
  return(posterior_thetas)
}
```

Once we've read the function into memory, we call it by providing the arguments to the function. Here we'll simulate `10000` samples with a starting value for θ of `0.9` and a standard deviation for our normal proposal distribution of `0.05`.

```
set.seed(555)
posterior_thetas <- metropolis_algorithm(samples = 10000, theta_seed = 0.9, sd = 0.05)
```

Now that we have 10,000 draws from the posterior distribution for the fairness factor θ stored in the `posterior_thetas` vector let's look at a kernel density estimate of the posterior. We will provide the code for plotting Figure 1 below, but the rest of the study note will omit plotting code to allow the reader to focus on the most important aspects of the algorithms without getting hung up on the plotting syntax.

```
opar <- par()
par(mar=c(2.5,3.5,3,2.1), mgp = c(1.7, 1, 0))
```

```r
d <- density(posterior_thetas)
plot( d
     ,main = expression(paste('Kernel Density Plot for ', theta))
     ,xlab = expression(theta)
     ,ylab = 'density'
     ,yaxt = 'n'
     ,cex.lab = 1.3
     )
polygon(d, col='dodgerblue1', border='dark blue')
```

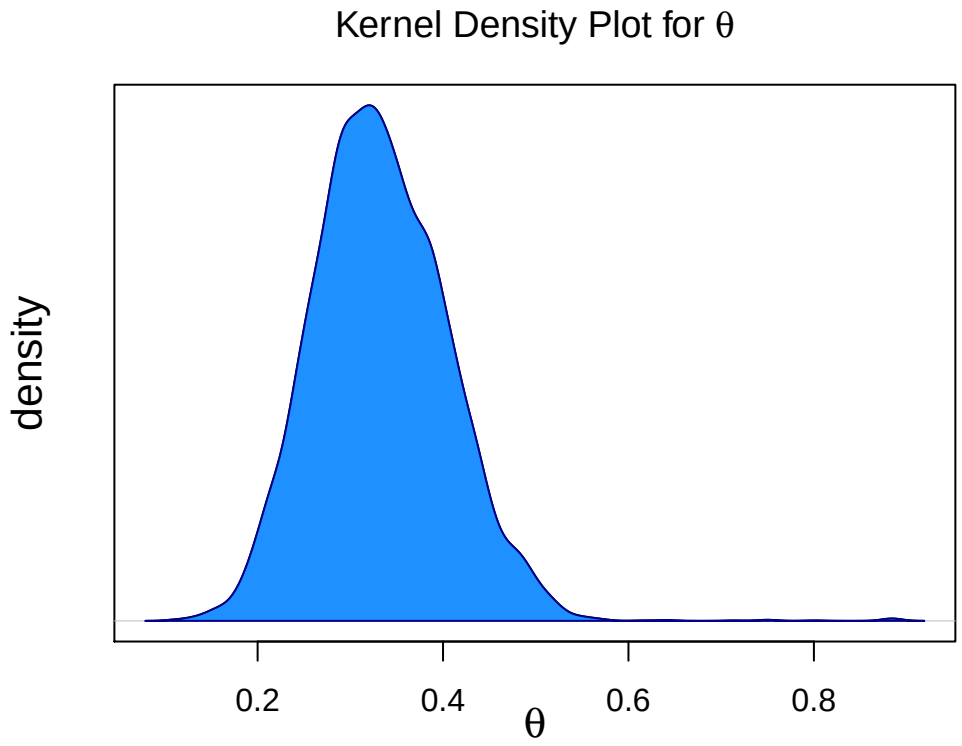

Figure 1: MCMC kernel density estimate plot

Notice the following points about Figure 1:

- We started the algorithm at $\theta = 0.9$ which was not close to where most of the posterior density exists, so our algorithm quickly moved us towards lower values of θ as seen by the lack of mass on the right-hand-side of the plot.
- The mode of the sampled posterior distribution is around 0.3. This also happens to be close to the maximum likelihood estimate for θ of $\frac{13}{41}$. This is no mistake. Recall that our prior was weak relative to the information contained in the 41 samples.
- Since the beta distribution is conjugate to the binomial distribution, we can see how close our sampled posterior distribution approximates the exact posterior for θ. Recall that with a $\beta(2, 2)$ prior and with 13 heads in 41 flips, the posterior distribution is also beta distributed with $a = 2 + 13$ and $b = 2 + (41 - 13)$. So, let's overlay the exact posterior distribution on our previous graph:

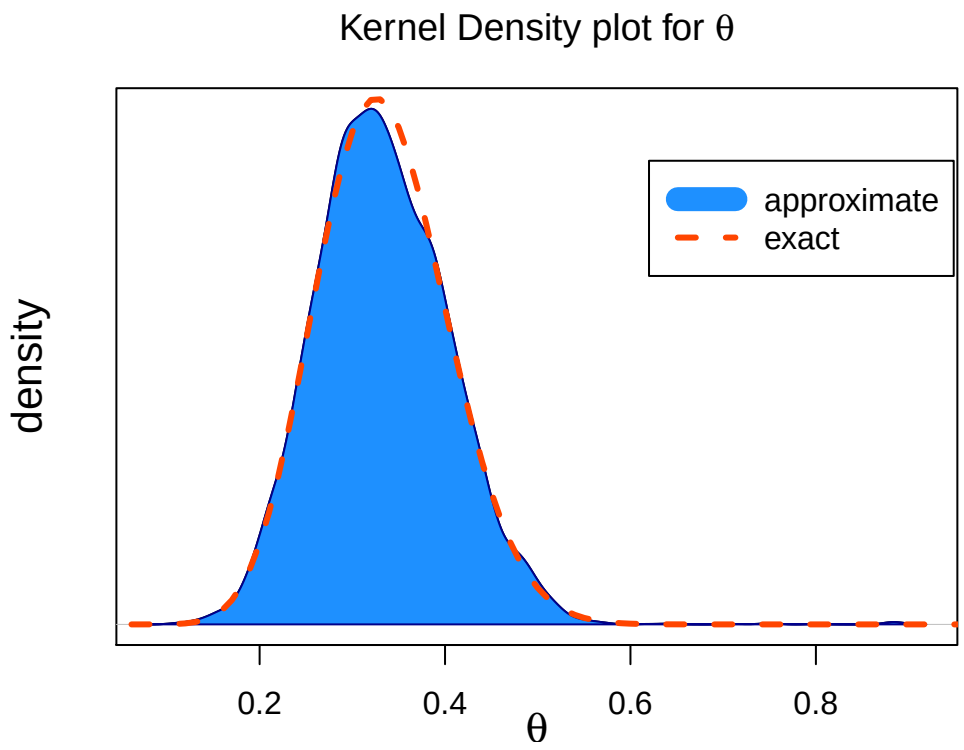

Figure 2: MCMC kernel density vs exact density

The prior distribution that we chose for this example is a *proper* distribution, meaning it integrates to 1.0 (i.e. $\int p(\theta)d\theta = 1.0$). We could've also chosen a prior distribution that did not integrate to 1.0 by, say, multiplying all its densities by a constant c such that $\int g(\theta)d\theta = c$. Such a distribution would be labeled as an *unnormalized density* and could easily be re-normalized by dividing by c. Once we have a proper prior distribution (perhaps by re-normalizing) we are guaranteed to have a proper posterior distribution. However, what if the prior density is *improper* and integrates to ∞? It turns out that even improper priors can produce proper posterior distributions. For example, the $\beta(0,0)$ is an improper prior distribution, but it turns out that so long as our data contains at least one coin flip with a head and one flip with a tail, the posterior $p(\theta|y)$ is proper. In fact, using the power of conjugacy we know the posterior with a $\beta(0,0)$ prior is $\beta(heads, flips - heads)$, which is certainly proper so long as we we have one head and one tail. In general, if $\int p(\theta|y)d\theta$ is finite for all y then the posterior is (unnormalized) proper.

Why would we ever choose an improper prior such that we have to worry about the posterior being proper? Sometimes improper priors are convenient and thus are employed in a Bayesian analysis. For instance, a prior for a variance parameter, v, might be defined as $p(v) \sim \frac{1}{v}$ for $v > 0$ which implies that v becomes less likely in inverse proportion to its value. If we do the calculus we will see that this is not a proper prior, but the convenience of assigning v's prior in this manner is evident. In general, improper priors can be safely employed when the likelihood dominates the prior, but we should always check to ensure that the resulting posterior is (unnormalized) proper.

Going back to our coin-flipping example, it's instructive to see the relative influence between the data (through the likelihood) and the prior distribution. To most easily observe, suppose that we changed our prior distribution on θ to reflect the fact that we've observed a lot of coin flips in the past, perhaps from other coins, and have generally found that all those coins were fair. So even though we've observed only 13 heads in 41 flips for this coin, we have a strong prior belief that this coin, just like all the others we've observed, is fair. Accordingly, we're going to put a regularizing prior on the fairness factor. To effect this, all we need to do is scale up our selections of a and b. Let's select a $\beta(a = 20, b = 20)$ prior distribution. Again,

we know the analytical solution to the posterior distribution is:

$$\theta^{post} \sim \beta(20 + 13, 20 + (41 - 13))$$

So let's compute the three components of Bayes' equation in R.

```
a <- 20; b <- 20 # Prior parameters

# Create a sequence of thetas ranging from 0.1 to 0.9 since this is where the density is
min <- 0.1; max <- 0.9
thetas <- seq(from = min, to = max, length.out = 1000)

# Pass the sequence of thetas to calculate the likelihood, prior, and  posterior
likelihood <- dbinom(heads, flips, thetas)
prior <- dbeta(thetas, a, b)
posterior <- dbeta(thetas, (a + heads), (b + flips - heads))
```

Graphing these densities consecutively yields the following facet plot:

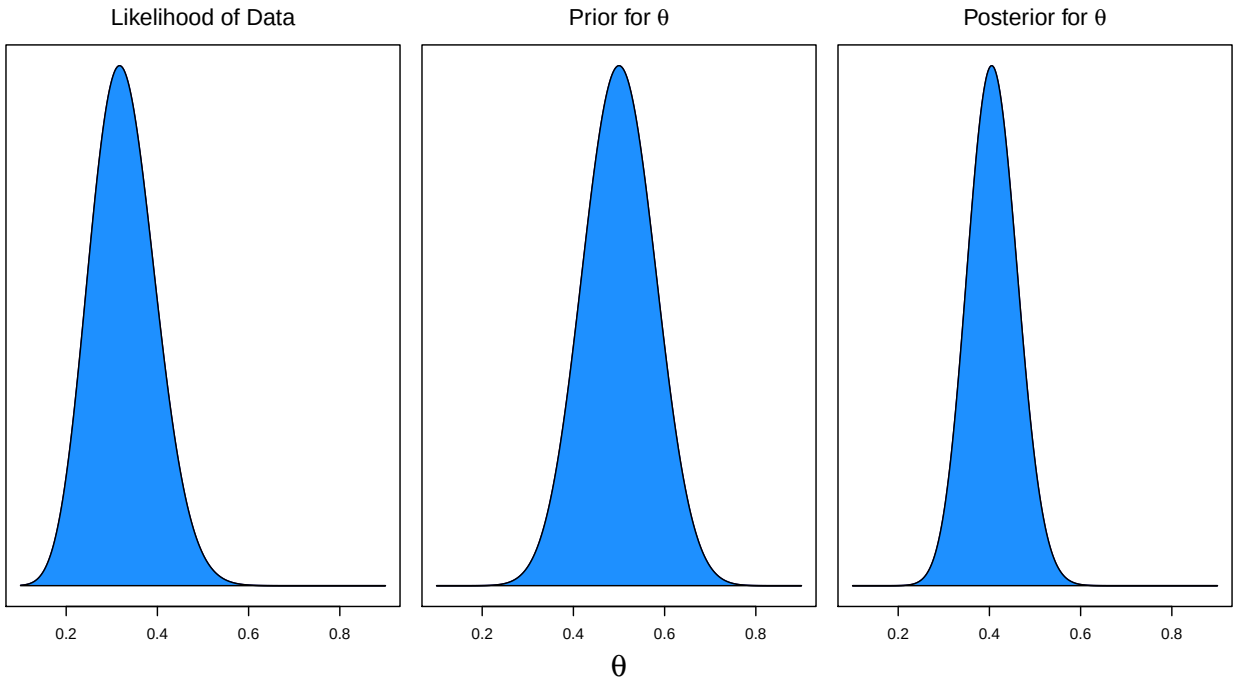

Figure 3: Facet Plot of Likelihood, Prior, and Posterior

In Figure 3, the posterior distribution's mode is between the prior distribution's mode of 0.5 and the likelihood's mode of $13/41 = 0.32$. So the effect of a regularizing prior was to exert more gravitational pull on the posterior towards itself. Higher values of a and b would pull the posterior even further towards the prior's concentration of mass.

Before moving on it's worth going back to our proposal distribution and discussing the decision to set $\sigma = 0.05$. This was a decision we casually made, but this parameter is important for the efficiency of our algorithm. Setting σ too low means that it will take many samples to map out the high and low density regions of the posterior. Setting it too high means that many samples are wasted as the θ_{prop}'s bounce around from high to low density areas of the posterior with most being rejected. To most easily see this, let's plot the first 100

samples from the posterior under three scenarios: (1) a small σ of 0.005, (2) our original σ of 0.05, and (3) a large σ of 0.5.

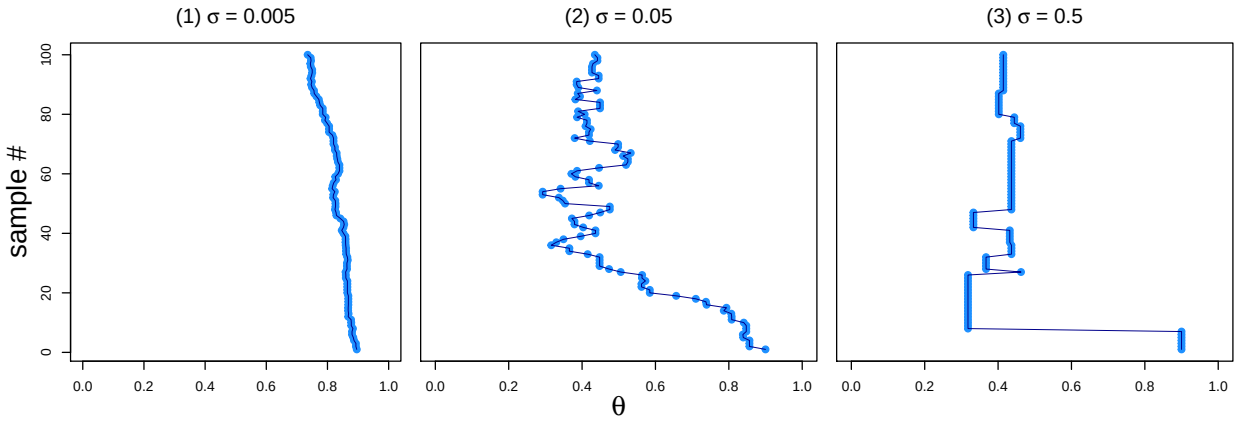

Figure 4: First 100 samples with three different proposal distributions. Inspired by Figure 7.4 in Kruschke, J.K. (2015). *Doing Bayesian Data Analysis in R: A Tutorial with R, JAGS, and Stan.* 2nd Edition. Academic Press/Elsiver.

Figure 4 above reveals that the random walk with $\sigma = 0.005$ has barely moved away from our seed for θ of 0.9 after 100 samples while the random walk with $\sigma = 0.5$ wastes a lot of samples. This waste occurs because when θ moves into the region of high posterior density after several samples, the dispersed proposal distribution makes it likely to draw a θ_{prop} far away from the current position which will often be rejected. It's important to recognize that with enough time and patience all three of these scenarios would converge to the same posterior distribution. It's just that some proposal distributions will be more efficient than others in sampling from the posterior.

Metropolis Algorithm in 2-D

Now that we have programmed and summarized the posterior distribution from the Metropolis algorithm from scratch in a single parameter dimension, it naturally begs the question of how we generalize to two or more dimensions. We will extend our coin-flip example by introducing another coin. In doing so we will need a joint prior distribution for the fairness factor of both coins. The joint prior distribution will be of the form $p(\theta_1, \theta_2)$ where θ_1 is the fairness factor for the first coin and θ_2 is the fairness factor for the second coin. To simplify matters, we will assume our prior beliefs about these coins are independent. What this implies is any prior belief for θ_1 is not affected by any prior belief for θ_2 and vice versa. Mathematically:

$$p(\theta_i|\theta_j) = p(\theta_i) \ for \ i, j = 1, 2 \ and \ p(\theta_1, \theta_2) = p(\theta_1)p(\theta_2)$$

As in the one-dimensional case, we will opt for a beta distribution as the prior. Since we have no reason to believe that these coins are not fair we will choose the same values of a and b for both priors.

We will also assume that the data generation process between the two coins are independent of each other and, as we did in one dimension above, we will assume that flips for a coin are independent. Armed with these assumptions, we aim to compute the joint posterior distribution of θ_1 and θ_2:

$$p(\theta_1, \theta_2|Data) = \frac{p(Data|\theta_1, \theta_2)p(\theta_1, \theta_2)}{\int\int p(Data|\theta_1, \theta_2)p(\theta_1, \theta_2)d\theta_1 d\theta_2}$$

Given our assumptions, it should come as no surprise that the joint posterior distribution can be factored into two independent beta distributions due to conjugacy.

$$p(\theta_1, \theta_2|Data) = \beta(a_1 = a + heads_1, b_1 = b + flips_1 - heads_1)\beta(a_2 = a + heads_2, b_2 = b + flips_2 - heads_2)$$

Just like in the one-dimensional case, it is helpful to visualize the joint prior, likelihood, and posterior. But before we do, we need to parameterize our prior distributions and we need some data. As mentioned above, we will assume the priors are independent and both θ_1 and θ_2 follow a $\beta(10, 10)$ distribution. Then suppose we flip the first coin `25` times and get `17` heads and since our thumb is tired, we flip the second coin only `9` times and get `1` head. Let's compute a grid of exact posterior results in `R` so that we can plot the three components: the joint prior, the joint likelihood, and the joint posterior.

```
flips_1 <- 25; heads_1 <- 17
flips_2 <- 9; heads_2 <- 1
a <- 10; b <- 10

thetas <- seq(from = 0, to = 1.0, length.out = 50)

prior_1 <- dbeta(thetas, a, b)
prior_2 <- dbeta(thetas, a, b)

likelihood_1 <- dbinom(heads_1, flips_1, thetas)
likelihood_2 <- dbinom(heads_2, flips_2, thetas)

posterior_1 <- dbeta(thetas, (a + heads_1), (b + flips_1 - heads_1))
posterior_2 <- dbeta(thetas, (a + heads_2), (b + flips_2 - heads_2))

# Matrix of densities. Multiply (*) because of independence.
joint_prior <- outer(prior_1, prior_2, FUN='*' )
joint_likelihood <- outer(likelihood_1, likelihood_2, FUN='*')
joint_posterior <- outer(posterior_1, posterior_2, FUN = '*')
```

And now we plot the `joint` densities:

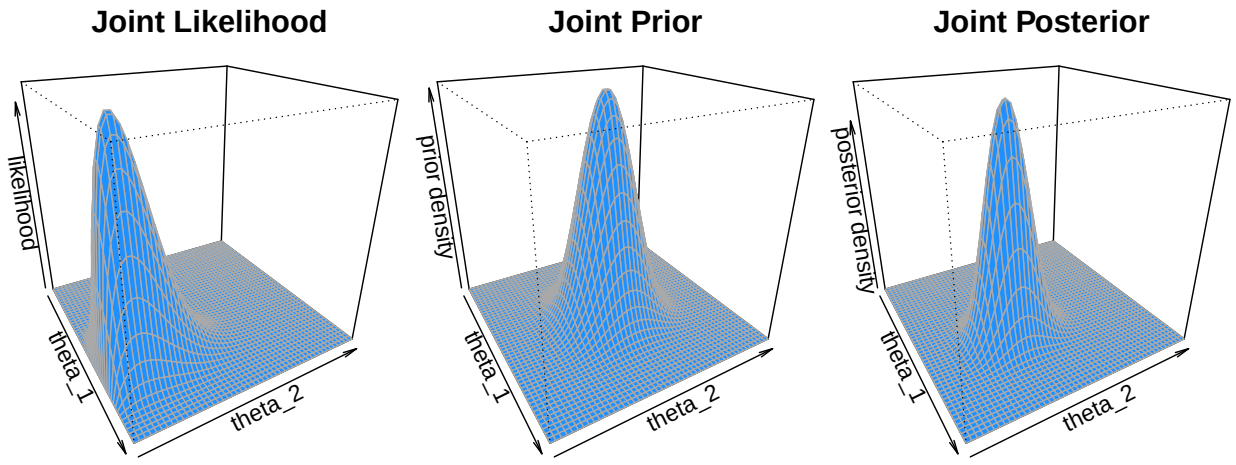

Figure 5: Facet Plot of Likelihood, Prior, and Posterior in 2-D

Now, suppose that we can't compute the posterior analytically. In that case, the Metropolis algorithm can be extended to the two-dimensional parameter space. The code below is very similar to the single parameter case

above, except we need to do things in 2-D, which means we employ vectors instead of scalars and matrices instead of vectors. Code is provided below of a naive implementation of the Metropolis algorithm in two dimensions. The main thing to be aware of is that we supply a covariance matrix for the proposal distribution instead of a scalar standard deviation. To keep things as straightforward as possible, we make our proposal distribution bivariate normal and again choose a variance of 0.05^2. We also assume that draws from the proposal distribution are independent by passing a covariance matrix, `cov_mat`, that has `0`s off the diagonal implying no covariance between θ_1^{prop} and θ_2^{prop}. Otherwise the code is a straightforward adaption of the single parameter case. We'll define our function and then call it in the same block of code. We request `20000` samples and we set a seed for reproducibility.

```
library(MASS) # Needed for mvrnorm() function

flips <- c(flips_1, flips_2) # Create a vector of flips
heads <- c(heads_1, heads_2) # Create a vector of heads

metropolis_algorithm_2d <- function(samples, thetas_seed, cov_mat){# 2D metropolis
                                                                   # algorithm function

  thetas_curr <- thetas_seed
  posterior_thetas <- matrix(NA, nrow = samples, ncol = 2) # Create a matrix to store
                                                           # results
  for (i in 1:samples){# Similar for loop as before in 1-D

    # 2-dimensional proposal distribution
    thetas_prop <- mvrnorm(n = 1, mu = thetas_curr, Sigma = cov_mat)

    # If any of the values in thetas_prop is not in [0, 1.0] then replace their value with
    # the same element in thetas_curr, otherwise keep their values
    thetas_prop <- mapply(function(x, y) ifelse((x > 1 | x < 0), y, x),
                          x = thetas_prop,
                          y = thetas_curr)

    # Calculate Bayes' numerators
    posterior_prop <- dbeta(thetas_prop, a, b) * dbinom(heads, flips, thetas_prop)
    posterior_curr <- dbeta(thetas_curr, a, b) * dbinom(heads, flips, thetas_curr)

    # Accept/reject logic
    p_accept_theta_prop = pmin(posterior_prop/posterior_curr, 1.0)

    # Draw 2 random uniform values
    rand_unif <- runif(n = 2)

    # If probability of accept is greater than the random uniform then accept
    # thetas_prop otherwise return thetas_curr
    thetas_select <- mapply(function(x, y, w, z) ifelse(x > y, w, z),
                            x = p_accept_theta_prop,
                            y = rand_unif,
                            w = thetas_prop,
                            z = thetas_curr)

    posterior_thetas[i, ] <- thetas_select

    # Reset theta_curr for the next iteration of the loop
    thetas_curr <- thetas_select
```

```
  }
  return(posterior_thetas)
}

# Call the function
set.seed(225)
posterior_thetas_2d <- metropolis_algorithm_2d(20000, thetas_seed = c(0.5, 0.5)
                                               ,cov_mat = matrix(c(0.05^2, 0, 0, 0.05^2)
                                                                 ,ncol=2))
```

Now let's plot our `posterior_thetas_2d` matrix:

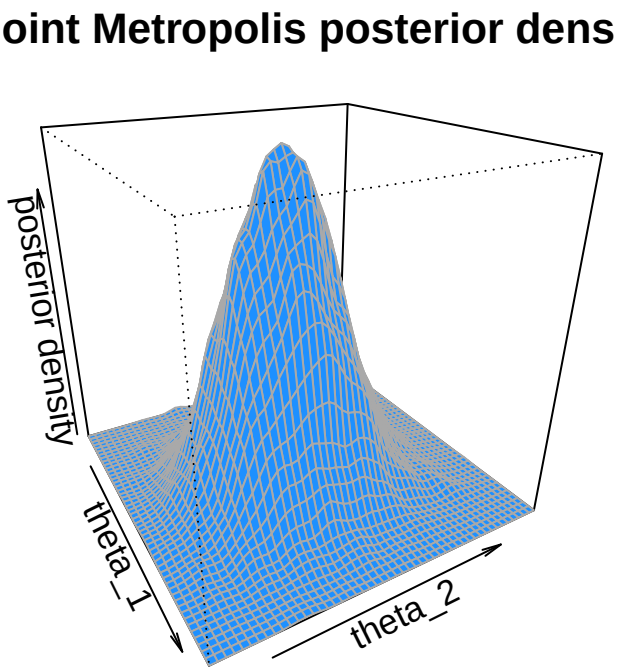

Figure 6: Joint MCMC posterior density

To convince ourselves that the Metropolis algorithm yields a chain of samples that approximate the exact posterior distribution, let's compare a couple summary statistics. We'll consider the first 5,000 samples from the `posterior_thetas_2d` matrix as the **warmup**[2] period of the chain of samples. We should remove samples from the warmup period when assessing whether the chain is representative of the target posterior distribution. Otherwise MCMC estimators based on early samples will be biased because the early samples of a chain are influenced by their starting positions which may be far removed from the posterior's area of high density. We'll discuss how to set an appropriate warmup period and how to assess convergence in the **MCMC Diagnostics** sections of the study note. To evaluate our MCMC sample, we'll make a large number of draws from the exact posterior distribution to compare to `posterior_thetas_2d`. Since the exact

[2]Consistent with recent trends in the Bayesian literature, we will refer to the early samples of a chain that get discarded as the **warmup** period and not **burn-in**. Burn-in carries the connotation in industrial engineering of throwing away stress-tested products that fail. We don't discard whole MCMC chains that fail since the failure of one chain implies a failure of the whole process.

joint posterior can be factored into two independent beta distributions, we can sample from the marginal beta distributions independently. Here's code that does this:

```
set.seed(481) # For reproducibility
warmup<- seq(5000) # 1,2,...,5000
posterior_thetas_2d_post_warmup <- posterior_thetas_2d[-warmup, ] # Remove warmup period

n <- nrow(posterior_thetas_2d_post_warmup) # Number of samples to draw from the exact
                                           # marginal posterior distributions. We set this
                                           # equal to the number of samples post warmup

exact_theta_1 <- rbeta(n, (a + heads_1), (b + flips_1 - heads_1)) # Random sample from the
                                                                  # first marginal
exact_theta_2 <- rbeta(n, (a + heads_2), (b + flips_2 - heads_2)) # Random sample from the
                                                                  # second marginal

apprx_theta_1 <- posterior_thetas_2d_post_warmup[ ,1]
apprx_theta_2 <- posterior_thetas_2d_post_warmup[ ,2]

# Combine into a matrix so that we can apply the quantile() function to each column of the
# matrix with one function call
matrx_thetas <- cbind(exact_theta_1, apprx_theta_1, exact_theta_2, apprx_theta_2)

# Let's look at the 2.5th, 5th, 25th, median, 75th, 95th, and 97.5th quantile
quant_thetas <- apply(matrx_thetas, MARGIN = 2, FUN = quantile, c(0.025,0.05, 0.25, 0.5
                                                                  ,0.75, 0.95, 0.975))
```

And here are the results:

Table 1: Quantiles of Posterior Distributions

	exact_theta_1	apprx_theta_1	exact_theta_2	apprx_theta_2
2.5%	0.457	0.456	0.220	0.218
5%	0.481	0.479	0.242	0.241
25%	0.551	0.555	0.319	0.317
50%	0.601	0.605	0.378	0.376
75%	0.651	0.655	0.439	0.437
95%	0.716	0.718	0.530	0.523
97.5%	0.738	0.738	0.557	0.550

The approximated quantiles from our sampled chains appear fairly consistent with the exact quantiles which gives us confidence in our implementation of the Metropolis algorithm. Note that implementations of the Metropolis algorithm that are available in popular software libraries will be more efficient than our functions due to vectorization, but the `for` loops in our code allows us to see what's happening at each step more easily.

It's important to return to our proposal distribution once again. Recall that in the two-dimensional space the proposal distribution was bivariate normal with zero covariance between the two parameters. It doesn't have to be this way. If the covariances were anything other than zero, some proposals would be more likely to be made in some diagonal direction than another direction. Doing so might lead to a more efficient sampling strategy, although not in this case since we know the posterior distributions of our fairness factors are independent. Or the proposal distribution could be non-normal. A non-normal proposal distribution that is also non-symmetric (i.e. $p(accept \theta_{prop} | \theta_{curr}) \neq p(accept\ \theta_{curr} | \theta_{prop})$) would be an implementation of the generalization of the Metropolis algorithm known as the *Metropolis-Hastings algorithm*. Configuring the details of our proposal distribution so as to optimize efficiency and accuracy of the proposal sampling

is called *tuning*. In more complex settings tuning the proposal distribution is not just a convenience but a requirement to achieve meaningful results. Because of this, we look to other MCMC algorithms to improve efficiency without requiring us to fiddle with too many tuning "knobs" of the algorithm.

Gibbs Algorithm

Due to sampling inefficiency and the need to tune the proposal distribution of the Metropolis algorithm, we explore other popular MCMC sampling strategies. Recall that the core problem is that we don't always know the analytical form of the joint posterior distribution of the parameters (except in simple cases like we walked through above) and, even if we did know it, our computers might not be programmed to easily draw samples from it. The **Gibbs** algorithm is another such MCMC sampler which allows us to circumvent this problem. As we will see, Gibbs is an efficient algorithm that does not require tuning, but it also has some limitations.

Like the Metropolis algorithm, the Gibbs algorithm can be conceptualized as a random walk through the parameter space where we start the walk at an arbitrary point and where the next step only depends on the current position (this is the Markov property of the MCMC). What makes the Gibbs algorithm distinct from the Metropolis algorithm is how each step is taken. At each point in the walk, one component parameter is selected from the n-dimensional parameter space we're walking through. The component parameters are selected in turn creating an ordered cycle, $\theta_1, \theta_2, ..., \theta_n$. When a component parameter is selected, say θ_i, we choose a new value for that parameter by drawing it from the *conditional* posterior distribution of that parameter given the values of the other parameters and the data, $p(\theta_i|\theta_{j\neq i}, Data)$. The selected value for θ_i, combined with the unchanged values of $\theta_{j\neq i}$ is the new position in the random walk. The conditional sampling process then continues through all the parameters $\theta_1, \theta_2, ..., \theta_n$ to form a complete cycle and then the cycle is repeated.

Samples from the conditional posterior distribution across all parameters come to represent the joint posterior distribution, which is the goal. Why does this happen? A mathematical proof is beyond the scope of this study note, but here's some intuition. The Gibbs algorithm can be thought of as a special case of the Metropolis algorithm in which the proposal distribution dynamically depends on both the component parameter selected and its current location in the parameter space. In fact, the proposal distribution *becomes* the conditional posterior distribution for that parameter as the algorithm converges. And because the proposal distribution aims to mirror the conditional posterior distribution exactly, the proposed move is always accepted. Of course, because we seed the Gibbs algorithm with random parameter values, samples at early iterations may not be representative of the target posterior distribution and thus early samples should be discarded as warmup.

It is trivial to derive the fact that in our two-coin example above the conditional posterior probability distribution for θ_1 and θ_2 are both distributed $\beta(a + heads_i, b + flips_i - heads_i)$ $i = 1, 2$, and since they are independent, the joint posterior distribution is the product of the two conditional posterior distributions. Unlike the Metropolis algorithm where the parameters are sampled simultaneously, the parameters in the Gibbs algorithm are sampled consecutively, and so a scatterplot of the samples will look different than that of the Metropolis algorithm. Figure 7 below shows such a scatterplot for both the Gibbs and Metropolis algorithms with 500 random samples. Notice that each step of the random walk for the Gibbs algorithm is parallel to either the θ_1 or θ_2 axis as seen by the thin lines that connect the samples. This occurs because the sampled parameters are drawn one at time holding the other one constant. In contrast, the parameters update simultaneously in the Metropolis algorithm as evinced by the diagonal lines connecting successive samples.

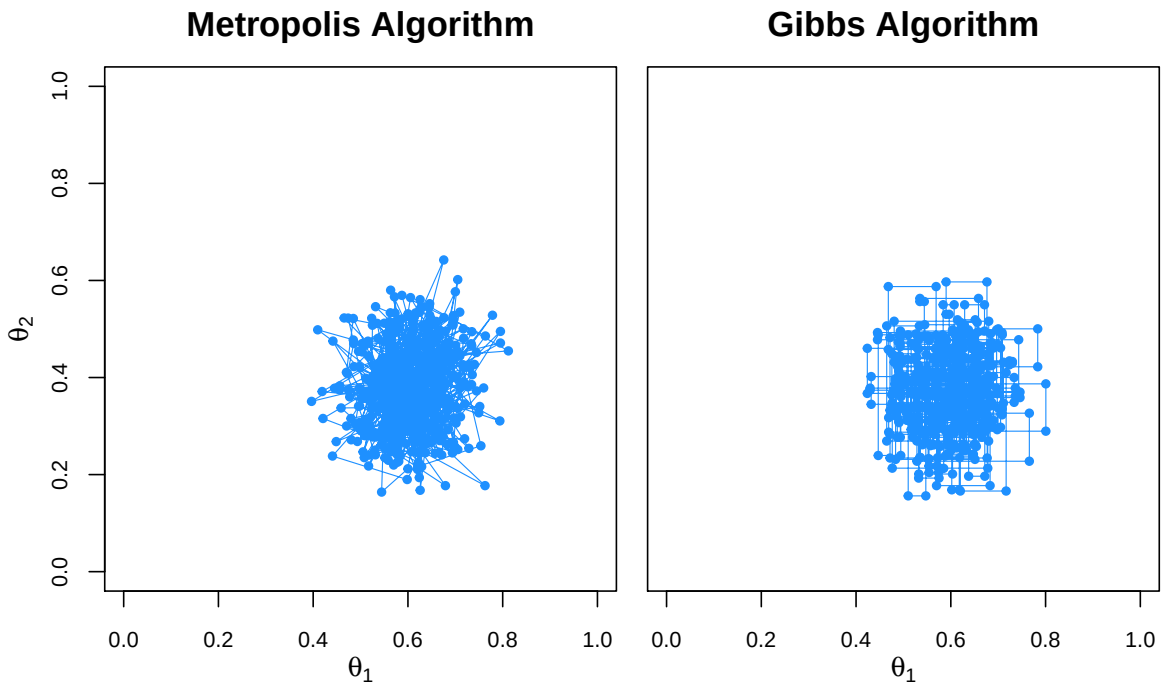

Figure 7: Scatterplot of samples comparing Metropolis and Gibbs algorithms. Inspired by Figure 7.6 in Kruschke, J.K. (2015). *Doing Bayesian Data Analysis in R: A Tutorial with R, JAGS, and Stan.* 2nd Edition. Academic Press/Elsiver.

The Gibbs algorithm is useful for when we can't compute or sample from the joint posterior distribution, $p(\theta_i|Data)\ for\ i = 1, 2, ...n$, but we can compute and sample from all the conditional posterior distributions, $p(\theta_i|\theta_{j\neq i}, Data)$. However, unless we are working exclusively with conditionally conjugate distributions we cannot typically compute and/or sample from the conditional posterior distributions exactly. If approximations of the conditional distributions can be constructed then the principles of the Gibbs algorithm can be combined with the accept/reject rule of the Metropolis-Hastings algorithm. Also if some of the parameters' conditional posterior distributions can be sampled from and some cannot then the parameters can be updated one at a time with the Gibbs algorithm where possible and a one-dimensional Metropolis update elsewhere. Regardless, this places constraints on the Gibbs algorithm.

Another disadvantage of the Gibbs algorithm is in the case of correlated parameters (such as a slope and intercept parameter in a simple linear regression). Imagine two highly correlated parameters such that their joint posterior lies in a narrow tunnel of density that sits at a 45? angle from the origin. In the Gibbs algorithm, each step in our walk lies parallel to one of the two axes as we saw above, and so we can envision that when it looks to take a step it cannot go very far in the direction of high density since it is constrained to stay in the tunnel and will quickly bump into the boundaries of the tunnel when it moves in either direction. This will cause the algorithm to stall and explore the posterior very slowly if at all. Thus the advantages of the Gibbs algorithm - efficiency in sampling from the posterior and no tuning of the proposal distribution - can be offset by a couple major disadvantages - namely the ability to compute and sample from conditional posterior distributions and sampling efficiency in models with correlated parameters.

Hamiltonian Monte Carlo

Similar to the Gibbs algorithm, the Hamiltonian Monte Carlo algorithm (HMC) employs a proposal distribution

that changes depending on the current position in parameter space, but unlike the Gibbs algorithm, HMC does not rely on being able to compute and sample from conditional posterior distributions of the parameters. Instead HMC solves for the direction in which the posterior density increases - called the *gradient* - and molds the proposal distribution in that direction. Recall that for the Metropolis algorithm the proposal distribution's shape is the exact same no matter where it finds itself in the parameter space. This is what leads to a lot of wasted samples and inefficiency.

How does HMC adapt its proposal distribution depending on the parameter's current position? HMC takes inspiration from the physics of energy and first computes the unnormalized negative log posterior density called the *potential function* (also sometimes referred to as the *potential energy*). It then generates a proposal in a fashion not too unlike flicking a marble that rests on the surface of the frictionless potential function at the location of the last accepted proposed parameter, which we've been calling θ_{curr}. The marble then rolls along the surface of the potential function for a period of time and wherever it ends up at the end of the period is the proposed parameter, θ_{prop}. That whole process - conceptualized as flicking a marble in a random direction with a random initial momentum and letting it roll around for a defined period of time and then recording the position of the parameter as θ_{prop} - is how HMC arrives at new proposals. That doesn't seem like it would produce a *distribution*, does it? Notice we said that the flick happens in a *random* direction with a *random* initial momentum. If this flicking process were repeated many times from the same starting location, the randomness would lead to many different trajectories for the marble to take as it rolls around on the surface of the potential function and thus many different terminal locations for the marble. It is these different terminal locations that can be conceptualized as the HMC proposal distribution for the current position.

The HMC process is illustrated in the graphic below which shows two HMC proposal distributions that start from two different θ_{curr}'s marked by two large dots in each of the columns. The first row shows the unnormalized posterior density. The second row shows the potential function with a random initial momentum applied to the marble. The third row allows us to visualize the potential trajectories the marble could take under different initial conditions (direction and momentum), and the fourth row displays this as a histogram of proposals.

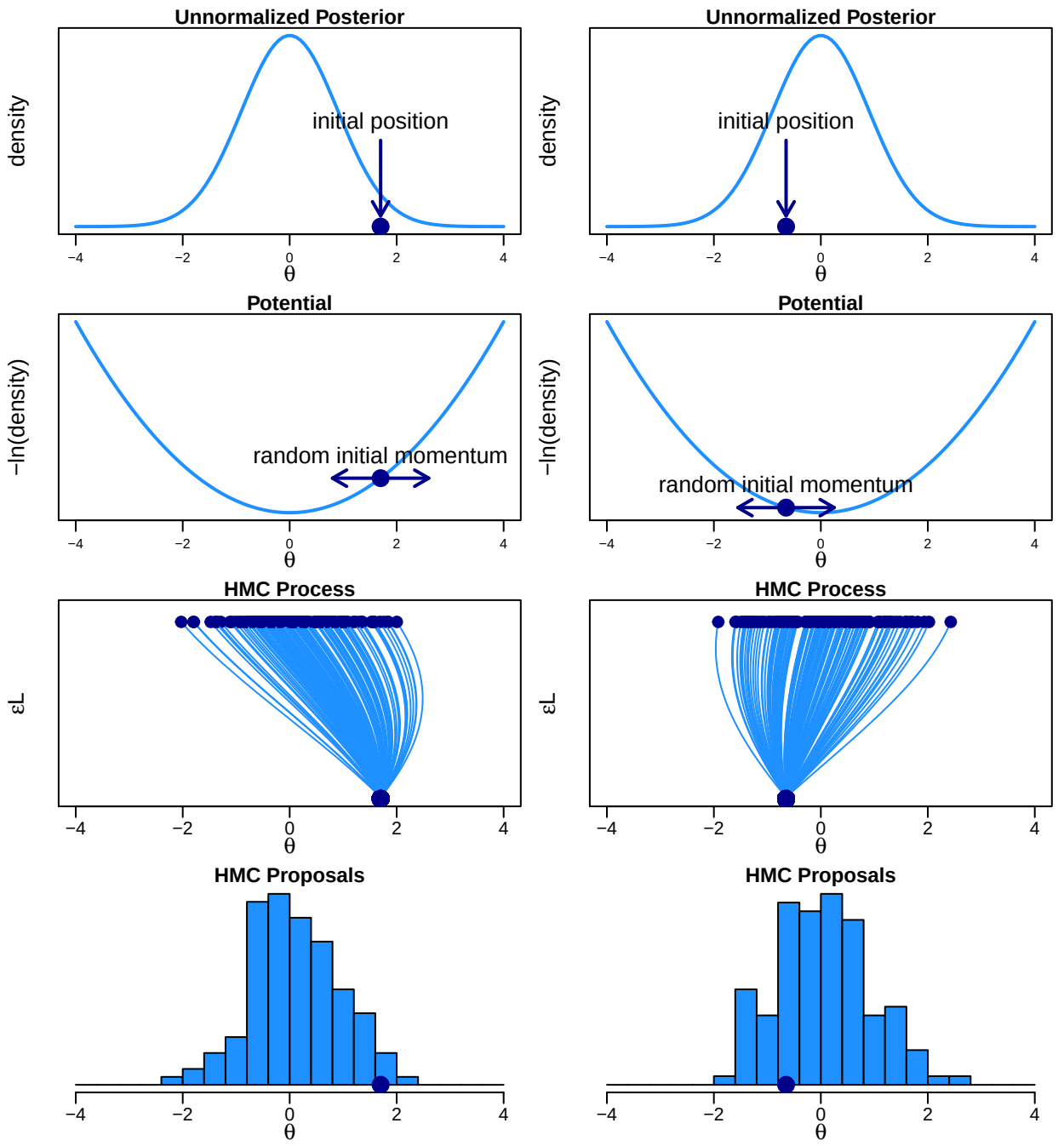

Figure 8: HMC process for two different starting positions. Inspired by Figure 14.1 in Kruschke, J.K. (2015). *Doing Bayesian Data Analysis in R: A Tutorial with R, JAGS, and Stan*. 2nd Edition. Academic Press/Elsiver.

In Figure 8 we notice that the third row's plots y-axes are labeled as ϵL. This can be conceptualized as the amount of time that the marble was allowed to travel before time elapsed. An HMC process involves L steps where each step involves (1) updating the momentum sequentially from its initial random value along the direction opposite of the gradient of the potential function (i.e. in the direction of the gradient of the posterior density) and then (2) updating the parameter, θ, based on the value of the momentum. ϵ controls how big any one update can be and L controls the number of updates. If we denote the value of the momentum as ϕ

and, as is customary for HMC, assume its distribution is normal with standard deviation s, and if we denote the potential function as $U(\theta|y)$, each step involves the following sub-steps:

1. Use the gradient of the potential function of θ to make a step of ϕ:

$$\phi \Leftarrow \phi - \epsilon \frac{\partial U(\theta|y)}{\partial \theta}$$

2. Use the momentum to update the position of θ:

$$\theta \Leftarrow \theta + \epsilon \frac{\phi}{s}$$

HMC performs these steps L number of times to produce the proposed parameter, θ_{prop}. Then it puts the proposal through the familiar acceptance logic from the Metropolis algorithm with one wrinkle: It augments the ratio of the Bayes' numerators at the proposed and the current parameter values with the ratio of the momentum densities at the proposed and current ϕ values.

$$p(accept\, \theta_{prop}) = min(\frac{Bayes'\ Numerator\ at\ \theta_{prop} p(\phi_{prop})}{Bayes'\ Numerator\ at\ \theta_{curr} p(\phi_{curr})},\ 1.0) \equiv min(\frac{l(Data|\theta_{prop})p(\theta_{prop})p(\phi_{prop})}{l(Data|\theta_{curr})p(\theta_{curr})p(\phi_{curr})},\ 1.0)$$

The acceptance formula is adorned in this way because, in theory, the sum of potential energy and kinetic energy is constant and therefore the ratio above will always be equal to 1.0 and a proposal will never be rejected. In practice, the continuous dynamics are discretized into small intervals (i.e. steps) and such calculations are only approximate such that not all proposals will be accepted. HMC can be tuned in three places: (1) the probability distribution of the momentum parameter ϕ which typically involves setting the standard deviation s (or the covariance matrix in multiple dimensions), (2) the step size, ϵ, and (3) the number of steps, L, per iteration. Modern implementations of HMC such as Stan help us by adapting these parameters during the warmup period. A couple of reasonable defaults is to set s based on some crude estimate of the scale of the potential function and then select ϵ and L, such that $\epsilon L = 1$. For instance, $\epsilon = 0.05$ and $L = 20$.

Figure 9 below shows the effect on the proposal distribution when s is set too low on the left panel and when s is set too high on the right panel. When s is set too low the proposal distribution is too concentrated and does not have sufficient time to move into the region of high posterior mass. Conversely when s is set too high the proposal distribution is too diffuse relative to the posterior distribution. With enough iterations both of these mis-tuned HMC implementations would likely produce a reasonable representation of the true posterior distribution but they wouldn't be as efficient as a well-tuned implementation.

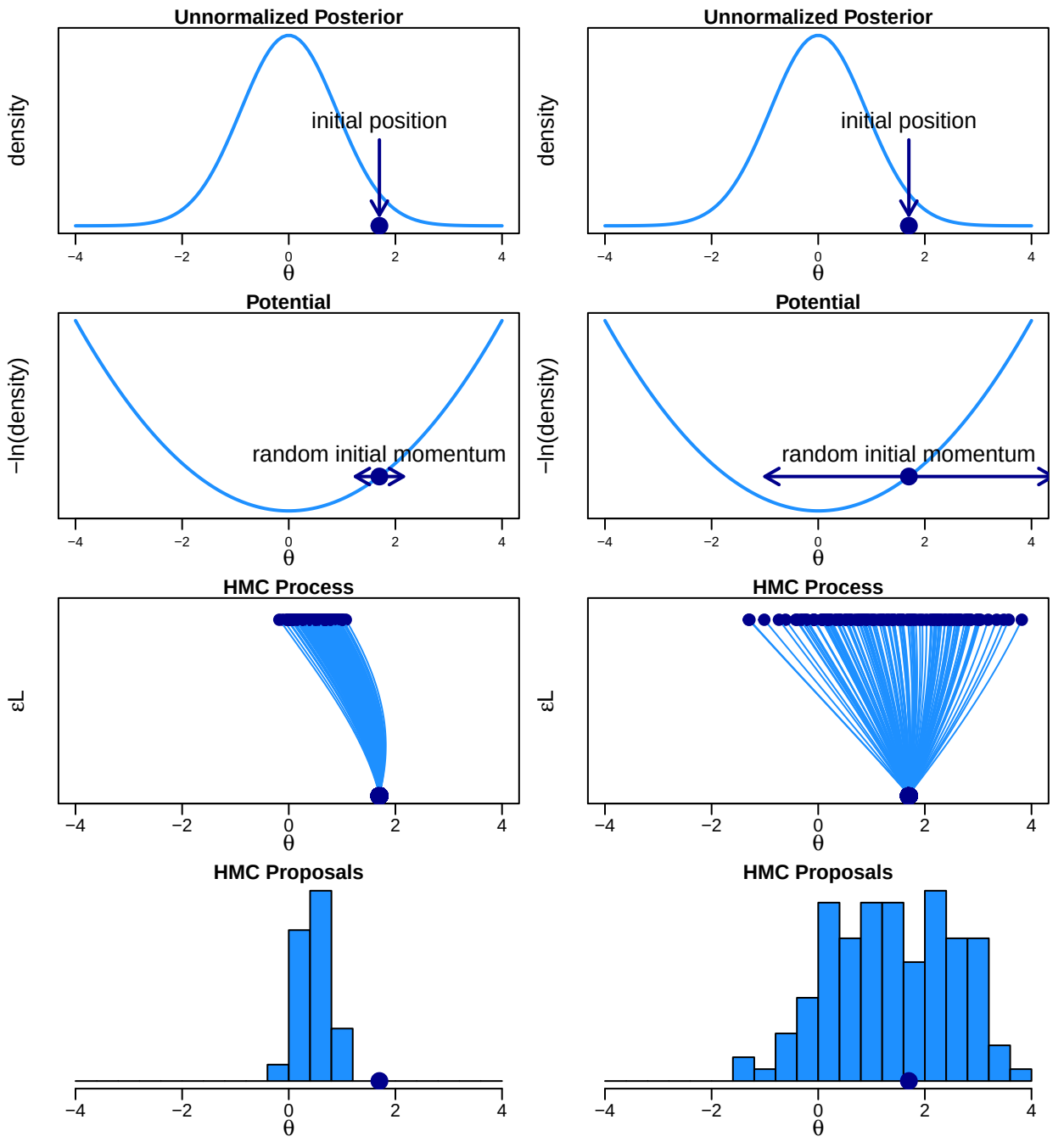

Figure 9: HMC process for two different proposal distributions. Inspired by Figure 14.2 in Kruschke, J.K. (2015). *Doing Bayesian Data Analysis in R: A Tutorial with R, JAGS, and Stan.* 2nd Edition. Academic Press/Elsiver.

Figure 10 reveals what happens with different ϵL trajectory lengths for two different starting positions. We can see that that the short trajectories on the left column of the figure don't give the "marble" enough time to "roll" around and thus don't produce a proposal distribution that represents the posterior well. The long trajectories on the right tend to overshoot the mode of the posterior and return towards the starting position making a U-turn. Implementations of HMC such as Stan provide guardrails to prevent such U-turns for these types of cases.

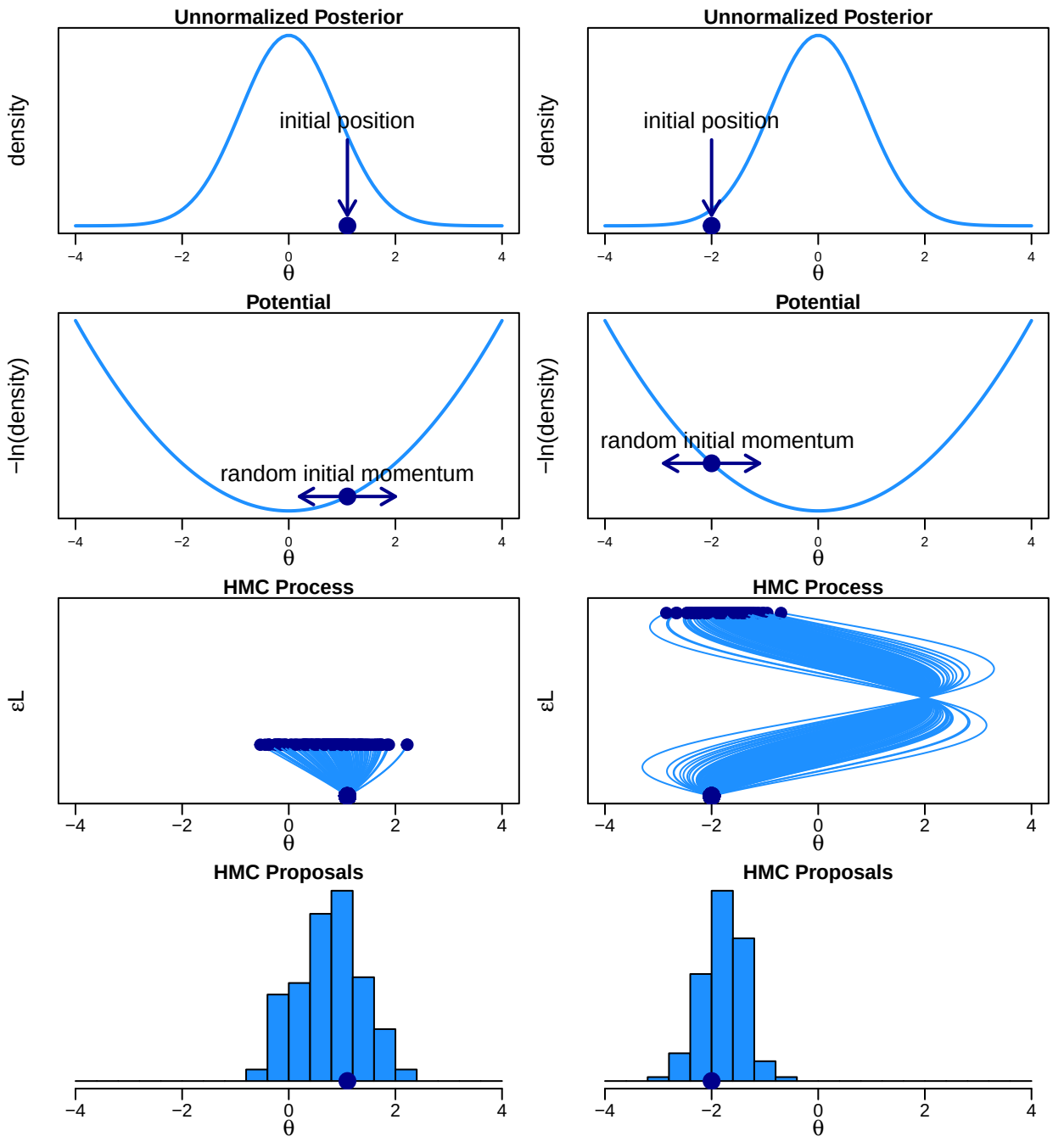

Figure 10: HMC process for two different trajectory lengths. Inspired by Figure 14.3 in Kruschke, J.K. (2015). *Doing Bayesian Data Analysis in R: A Tutorial with R, JAGS, and Stan*. 2nd Edition. Academic Press/Elsiver.

We may wonder why we would bother with HMC if it requires just as much tuning as the Metropolis algorithm and isn't as efficient as the Gibbs algorithm. As mentioned above, modern implementations of HMC are sophisticated enough such that HMC achieves a high degree of sampling efficiency and works well for a variety of situations that might be intractable for the Metropolis algorithm (e.g. parameter-heavy hierarchical models) and the Gibbs algorithm (e.g. highly correlated parameters). We can imagine fairly easily that HMC would be quite useful for posterior distributions that curve through the parameter space making something

like an S-shape. It is not a guarantee that HMC will always be more efficient than other MCMC algorithms, but it's a good default choice if other MCMC sampling algorithms are struggling with a particular model.

MCMC Diagnostics

As stated earlier, the goal of MCMC is to gather a representation of the true but incalculable posterior distribution via simulation. But simulations and the code we use to program them can go awry. Even if the simulation is executed flawlessly, we may not have given the simulation program reasonable defaults or let it run long enough to achieve good representation. Therefore, it would behoove us to utilize a set of diagnostics that indicate if the simulation process is behaving as expected. The diagnostics hopefully give us confidence that our samples are converging to a reasonable representation the target posterior. There are several methods which are necessary but not sufficient to ensure representativeness. We will explore some of them in turn below.

Trace plots - Trace plots involve simulating two or more chains of parameter values and plotting the values of each chain against the sample number of the sampling process, typically on the same set of axes. If the chains are all representative of the posterior distribution, they should overlap each other and be unrelated to their randomly set starting positions. They should also be stationary around the same modal value. We can create trace plots fairly easily with our one-dimensional `metropolis_algorithm` function used to sample θ. We will run three different chains with 10,000 samples each, and we will start with a diffuse set of `theta_seed`'s to ensure that the chains mix well despite their initial dispersion.

```
# Define our metropolis sampling process
samples <- 10000
theta_seeds <- c(0.05, 0.50, 0.95)
sd <- 0.075

flips <- 41; heads <- 13 # data

a <- 10; b <- 10 # prior beta distribution parameters

set.seed(124)
chains <- mapply(metropolis_algorithm, samples = samples, theta_seed = theta_seeds
                 ,sd = sd)
```

We plot the first 150 samples for each of the three chains below in Figure 11. Notice that these chains mix extremely fast. By about sample 25, they seem to have concentrated around the mode of the posterior distribution and are stationary. In this case we could safely set the warmup period to be about 100 iterations. In realistic applications, warmup periods are typically a couple hundred to a couple thousand iterations depending on the complexity of the model and the data that the model is trying to represent.

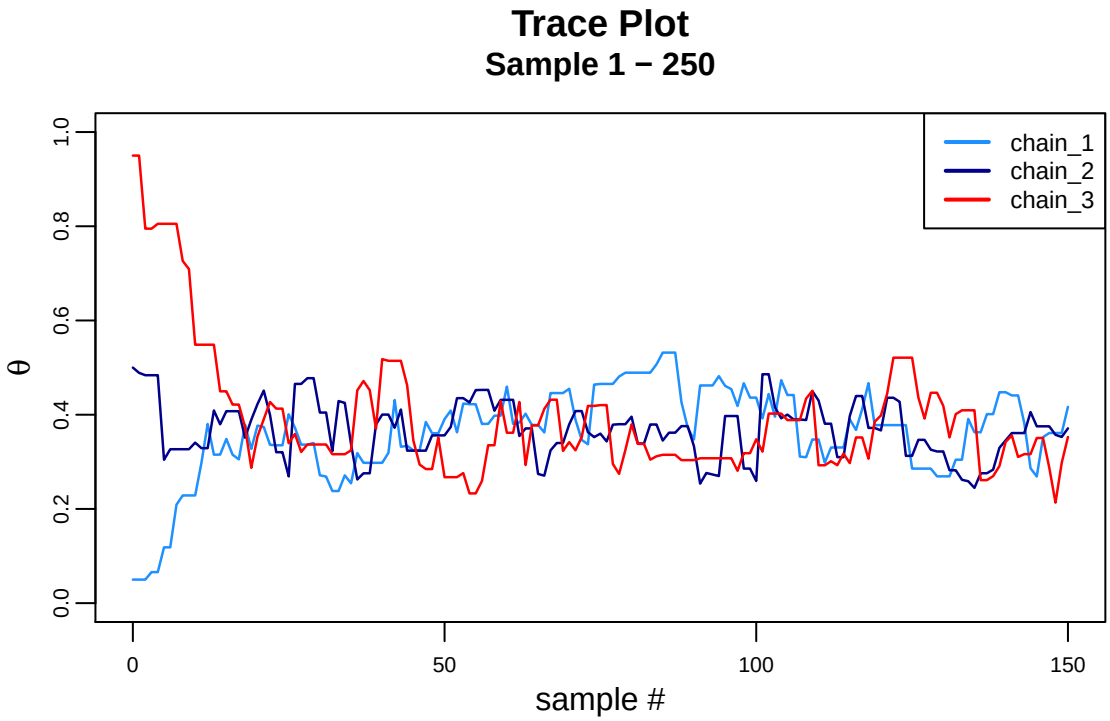

Figure 11: Trace plots: First 150 samples from three chains

And here are the last 5000 samples of the three chains. We can barely make out the individual chains.

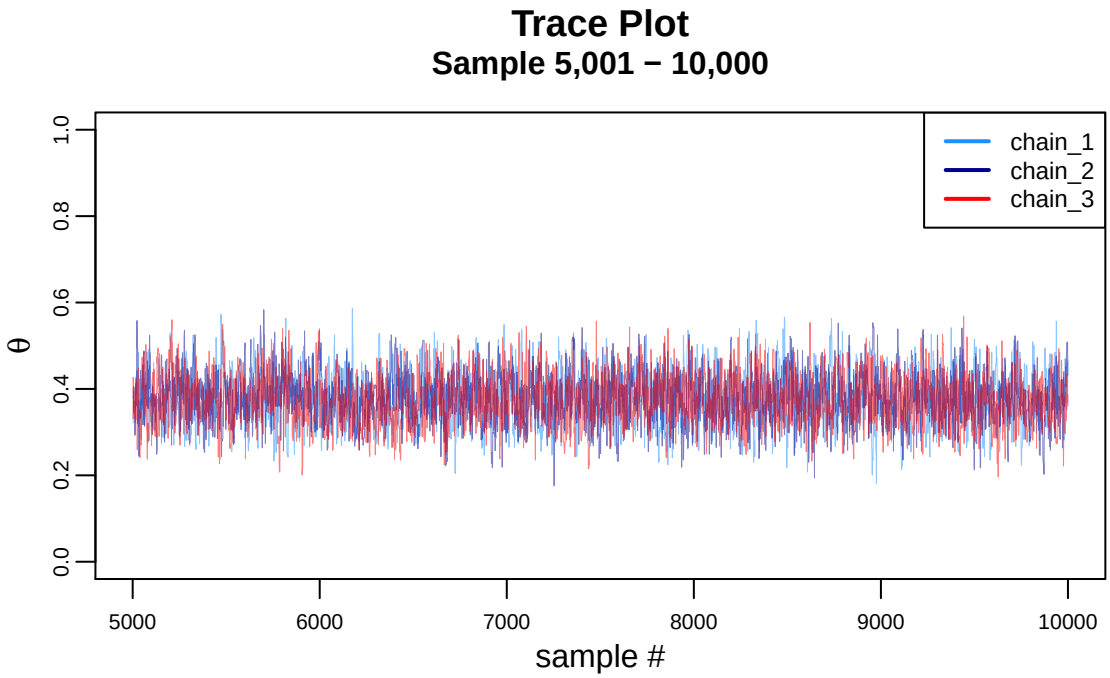

Figure 12: Trace plots: Last 5,000 samples from three chains

If any of the chains appeared to be isolated from the others then we would be nervous that the chains are not mixing well and have not converged on the posterior distribution. In that case, we could try to run more iterations or we might check our model definition, implementation method, or input data for potential issues.

Autocorrelation Plots - We know our within-chain samples are correlated since not all the θ_{prop}'s are accepted, and even if they are accepted, their value is a function of the last parameter value because θ_{curr} is the mean of the proposal distribution in the Metropolis algorithm. Even in HMC where the proposal distribution molds itself to the posterior distribution, the current parameter value will influence the proposal distribution for the next parameter. To get a sense of how correlated the proposed parameters are through the sampling iterations, we can plot the *autocorrelation* of our chains. This diagnostic won't necessarily give us an indication of if the chain is representative of the posterior distribution, but it will give us a sense of the efficiency of our MCMC algorithm. It will also be useful to check to see if all of our chains are behaving similarly. If any chain's autocorrelation plot is significantly different than the others this is a hint that something has gone wrong. The correlogram computed via the `acf` function in `R` provides this diagnostic. Let's run it on our three chains and store the results.

```
warmup <- seq(100) # Conservatively set the warmup period. We discard these samples from
                   # our correlogram

# Remove warmup samples
chains_post_warmup <- chains[-warmup, ]

# Apply the acf() across each column (margin 2) of the chains_post_warmup matrix
acfs <- apply(chains_post_warmup, MARGIN = 2, FUN = acf, plot=FALSE)
```

Plotting the correlograms in Figure 13 below, we observe that all three chains' autocorrelation is quite similar at each lag and the autocorrelation drops off fairly precipitously by lag 10. If our proposal distribution had a much smaller standard deviation, then the within-chain samples would be more highly correlated at all lags.

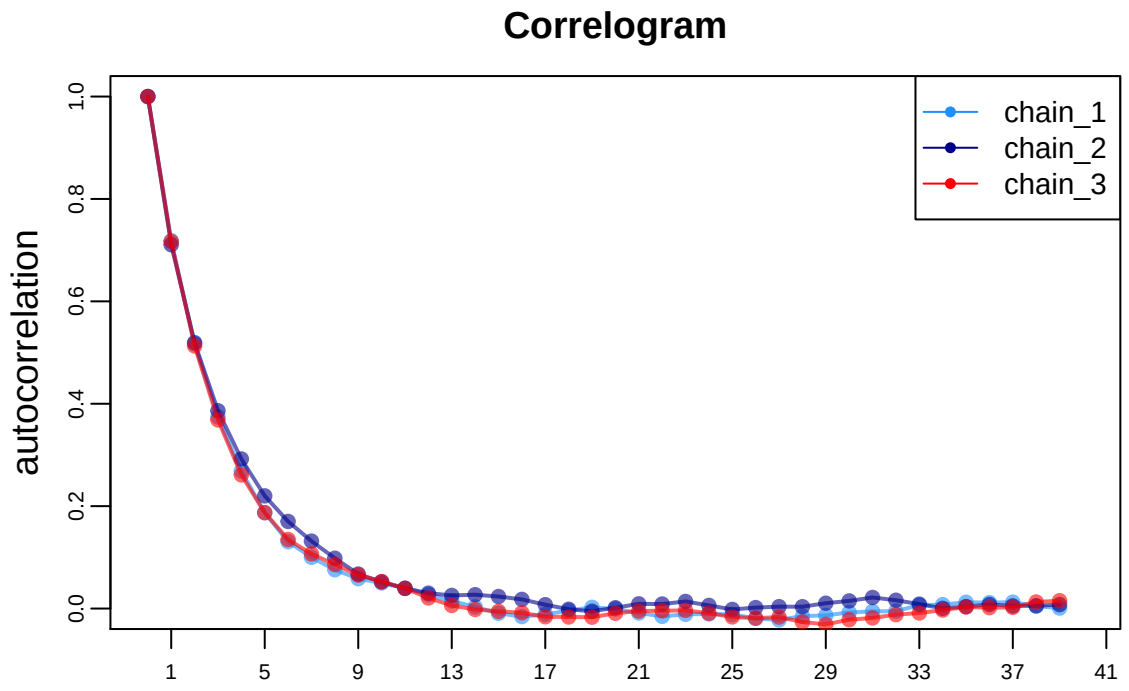

Figure 13: Autocorrelation for three chains

Effective Sample Size - Since MCMC chains are typically positively autocorrelated there is less information in an MCMC chain of samples as there would be if we were to estimate the posterior distribution outright by drawing independent samples from it. So it's natural to wonder what is the number of independent samples that would contain the same information as the total number of samples from our correlated MCMC chains. This is the *effective sample size*, or `n_eff` that you see referenced in *Statistical Rethinking*. The formula for `n_eff` follows directly from our correlogram seen above

$$n_{eff} = \frac{n}{(1 + 2\sum_{k=1}^{\infty} \rho_k)}$$

where ρ_k is the autocorrelation at lag k. Notice that the sum goes from $k = 1$ to $k = \infty$. Clearly, that is not possible to calculate in practice since our chains are finite. It's often standard to select a terminal point, t, such that the sum of the autocorrelation at two successive lags $\rho_{t-1} + \rho_t$ is negative. We can choose any of our three chains to compute `n_eff`. Let's select the first chain and show the calculations in `R`.

```
# Index into the first chain
chain1_post_warmup <- chains_post_warmup[, 1]
n <- length(chain1_post_warmup)

acf_1 <- acf(chain1_post_warmup, plot=FALSE, lag.max = 1000)[[1]] # Grab the
                                                                  # autocorrelations at
                                                                  # each lag

# Loop through each consecutive sum of autocorrelations while the sums are non-negative.
# The resulting t is our terminal point
acf_sums <- 999 # Initialize to be a positive number
t <- 1
while (acf_sums >= 0){
  t <- t + 1
  acf_sums <- acf_1[t - 1] + acf_1[t]
}

# Compute the denominator. The sum of the acf_1 vector is from the 2nd element to element
# t because the second element is the first lag, k = 1
denominator <- 1 + 2*sum(acf_1[2:t])

# Compute an approximation of the effective number of samples, n_eff
n_eff <- n / denominator
```

Most implementations of MCMC algorithms allow for *thinning* the sampled chains. This technique involves discarding every (h-1) of every h samples. The purpose is to reduce the autocorrelation of the sampled chains. While this technique can be useful from a memory or a post-sampling-processing time perspective, it does not improve the precision of the estimates from the posterior. To give us some intuition on why this is, let's thin one of the post warmup chains and then compute the effective sample size using the same logic as above.

```
h = 5 # Thin our chain by keeping only 1 out of every 5 samples
n <- length(chain1_post_warmup)
keep_seq <- seq(1, n, h) # Sequence of integers that represent the indices of the chain
                         # we will keep
chain1_post_warmup_thin <- chain1_post_warmup[keep_seq]

thin_n <- length(chain1_post_warmup_thin)

# Extract autocorrelations at each lag
acf_1_thin <- acf(chain1_post_warmup_thin, plot=FALSE, lag.max = thin_n/5)[[1]]
```

```
acf_sums <- 999
t <- 1
while (acf_sums >= 0){
  t <- t + 1
  acf_sums <- acf_1_thin[t - 1] + acf_1_thin[t]
}

thin_denominator <- 1 + 2*sum(acf_1_thin[2:t])
thin_n_eff <- thin_n / thin_denominator
```

Depending on the seed value, our results may vary but for this particular implementation we got `n_eff` = 1620 and `thin_n_eff` = 1239. We can see that thinning the chain, which reduces the autocorrelation, did not lead to a larger effective sample size and thus no information was added. Note that thinning the chain after already allocating the memory to store the samples of the chain like we did here is not best practice. We should rather adapt our `metropolis_algorithm` function to thin at the time of sampling such that we only store every h^{th} sample.

Gelman-Rubin Convergence Diagnostic - This diagnostic statistic comes in many different names in the Bayesian literature: Gelman-Rubin Convergence Diagnostic, $\hat{R}$, Potential Scale Reduction Factor, Shrinkage Factor, etc. All of these measure the same thing - the relationship between the estimated variance of the chains and the within chain variances. In a multi-dimensional analysis, the statistic should be calculated for each parameter of interest independently. Here is how we do it:

- Sample c chains of posterior parameters. Discard the samples in the warmup period.
- Split the c chains into a first half and a second half such that we have m subchains where $m = 2c$. Each m chain is composed of n samples from the posterior
- Let θ_{ij} be the sampled parameter value $i \in 1, 2, ...n$ from chain $j \in 1, 2, ...m$
- Let $\bar{\theta_j} = \frac{1}{n}\sum_{i=1}^{n}\theta_{ij}$ be the average parameter for chain j
- Let $\bar{\theta} = \frac{1}{m}\sum_{j=1}^{m}\bar{\theta_j}$ be the grand average across all chains
- Compute the between-chain variance: $B = \frac{1}{m-1}\sum_{j=1}^{m}(\bar{\theta_j} - \bar{\theta})^2$
- Compute the within-chain variance: $W = \frac{1}{m}\sum_{j=1}^{m}\frac{1}{n-1}\sum_{i=1}^{n}(\theta_{ij} - \bar{\theta_j})^2$
- Calculate an estimate of the total chain variance: $V = \frac{n-1}{n}W + B$
- Finally, solve for the Gelman-Rubin Convergence Diagnostic: $\hat{R} = \sqrt{\frac{V}{W}}$

The intuition behind the formula for $\hat{R}$ is that V overestimates the total variance of the posterior assuming the starting distribution is overdispersed, but is unbiased under stationarity (if the starting distribution equals the posterior distribution) or in the limit as $n \to \infty$. Conversely W underestimates the variance of the posterior for a finite n because the individual chains have not had time explore the full posterior distribution, but W is also unbiased as $n \to \infty$. So if the chains are exploring the target posterior distribution, the numerator is an overestimate of the true variance and the denominator is an underestimate of the true variance but both converge in expectation to the true variance as n gets really large. We employ the Gelman-Rubin Convergence Diagnostic by monitoring its value as n increases. We hope to see that the value of the statistic decreases and stabilizes near 1.0 as the number of samples in our chain increases. Let's program our own Gelman-Rubin Diagnostic function so that we can call it on our chains.

A tricky part of our implementation below in that we need to split the chains in half so we provide a helper function cleverly named `split_chain` to pass to the `apply` function which applies `split_chain` to each of the three columns in the matrix (recall we have three sampled chains). Because `R`'s `split` function returns a list with two elements, if we don't use the `unlist` function with its `recursive` argument set to `FALSE` when we call it, we would get a nested list structure which would be difficult to operate upon.

```
# Helper function to split the chains in half
split_chain <- function(x){
  len <- length(x)
```

```
  split_x <- split(x, seq_along(x) <= len/2)
  return(split_x)
}

compute_Rhat <- function(chains){# Function to compute Gelman-Rubin Diagnostic

  chains_split <- unlist(apply(chains, MARGIN = 2, FUN = split_chain)
                         , recursive = FALSE) # Split chains in half

  n <- length(chains_split[[1]]) # Get number of samples in a split chain

  Ws <- sapply(chains_split, var) # Compute the variance of each chain. These are the
                                  # components of W
  W <- mean(Ws) # Take the mean of the components of W. This is W

  chain_means <- sapply(chains_split, mean) # Compute the mean of each chain

  B <- var(chain_means) # Compute the variance of the chain means to get B

  V <- (n-1)/n * W + B

  Rhat <- sqrt(V / W)

  return(Rhat)
}
```

And now we prepare our data to feed into the `compute_Rhat` function. For all chains simultaneously, we loop over a different number of terminal samples, `terminal_n`, truncate the chains at each `terminal_n` and then compute the Gelman-Rubin Statistic.

```
# Create a sequence of terminal number of samples. We will compute Gelman-Rubin
# after 10 samples, 60 samples, 110 samples,...,total_n samples
total_n <- nrow(chains_post_warmup)
terminal_n <- seq(from = 10, to = total_n, by = 50)

# Helper function that will be called in the subsequent loop to limit the chain size to
# `n` samples
truncate_chain <- function(chain, n){
  new_chain <- chain[1:n]
  return(new_chain)
}

Rhats <- rep(0, length(terminal_n)) # Create vector of 0's to store results

# Loop through each terminal_n by truncating the chains at the value of n and then calling
# the compute_Rhat function. Store the results in the Rhats vector that we initialized
# above
i <- 1
for (n in terminal_n){
  truncated_chains <- apply(chains_post_warmup, MARGIN = 2, FUN = truncate_chain, n)
  Rhats[i] <- compute_Rhat(truncated_chains)
  i <- i + 1
}
```

We then visualize our `Rhats` vector to check that the chains are mixing well. We are hopeful that they are

mixing about the true posterior distribution, but this diagnostic cannot guarantee that this is the case. It can only comment on whether the chains are mixing well. We would be concerned if as `n` increased the statistic was not approaching 1.0 and we would want to draw more samples if the statistic was generally moving towards 1.0 but was meaningfully above 1.0.

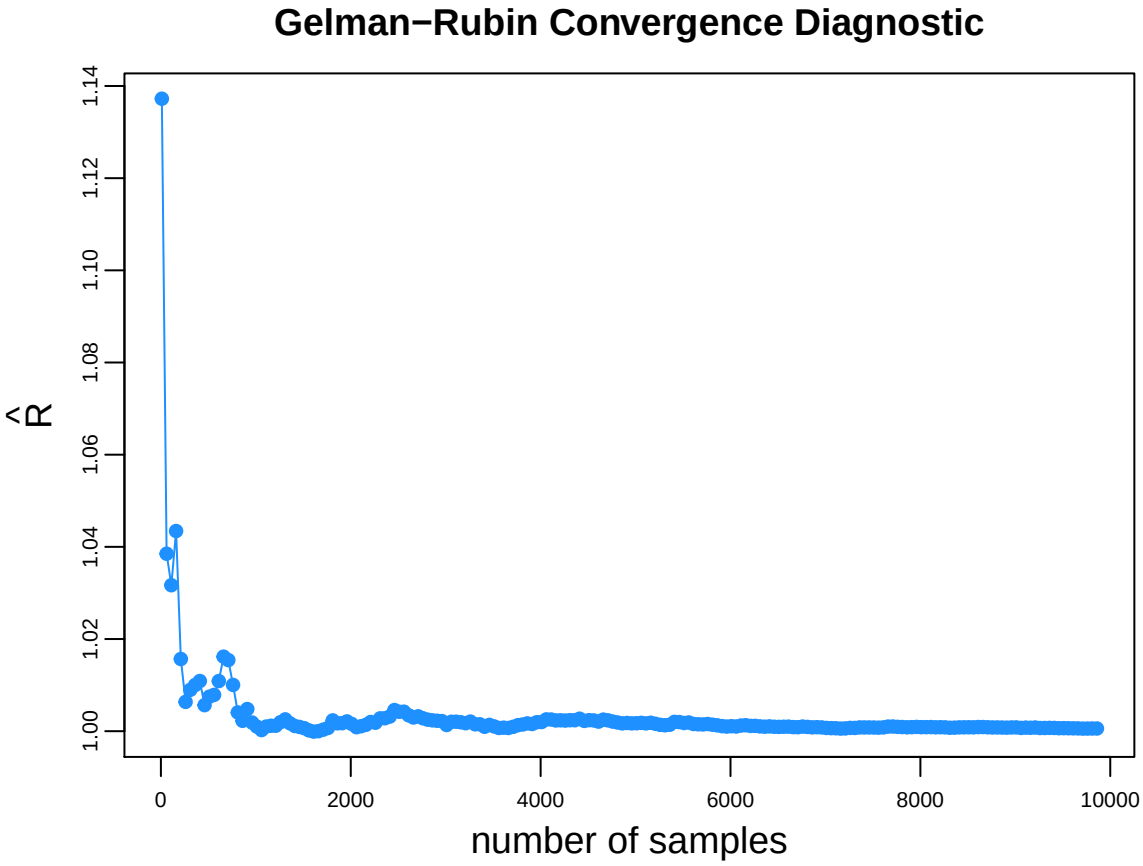

Figure 14: Gelman-Rubin Convergence Diagnostic as the sample size increases

Acceptance Rate - A chain's acceptance rate can be defined simply as the ratio of the number of unique values in the chain to the total number of samples in the chain. This is equivalent to the percentage of proposed parameter values, θ_{prop}, that are accepted in an MCMC sampler. We know that the Gibbs algorithm has an acceptance rate of 1.0 since all draws from the conditional posterior distributions are accepted. As mentioned earlier, sometimes it is impossible to sample from the full conditional posterior so Gibbs' approximations are employed which will likely not have an acceptance rate of 1.0. Metropolis, Metropolis-Hastings, and HMC algorithms will generally not have acceptance rates approaching 1.0. In most cases it would be counterproductive to have such high acceptance rates for these algorithms since the proposal distribution is only a crude approximation of the target posterior distribution. We can compute the acceptance rates of our three sampled chains for θ. Here's how we might do it in `R`.

```
# We'll provide two ways to do this. The first might be more intuitive to some readers as
# it better illustrates what we're actually doing. The second way is less code but perhaps
# a bit more opaque

# Intuitive way
```

```r
compute_accRate_intuitive <- function(chain){
  n <- length(chain) # Number of samples
  chain_front <- chain[-n] # Create a chain with element 1,2,3,..,n-1
  chain_back <- chain[-1] # Create a chain with element 2,3,4,...n
  accepts <- sum(chain_front != chain_back) # Count how many successive elements are
                                            # different from the next element
  acceptance_rate = accepts / (n - 1) # We subtract 1 because we could only make (n-1)
                                      # comparisons
  return(acceptance_rate)
}

# Opaque way. Employs R's unique() function which keeps only unique elements in a vector
compute_accRate_opaque <- function(chain){
  n <- length(chain)
  acceptance_rate = (length(unique(chain)) - 1) / (n - 1) # Subtract 1 from numerator
                                                          # because the first theta will
                                                          # always get counted as a unique
                                                          # value
  return (acceptance_rate)
}

# Apply both functions to our matrix of chains. Results should be equivalent
acceptance_rate_intuitive <- apply(chains_post_warmup, MARGIN = 2,
                                   FUN = compute_accRate_intuitive)

acceptance_rate_opaque <- apply(chains_post_warmup, MARGIN = 2,
                                FUN = compute_accRate_opaque)
```

Depending on the seed values employed, we get acceptance rates around 0.65. Notice that the opaque way of computing the acceptance rate using the `compute_accRate_opaque` function is only guaranteed to work if we have a high degree of precision in our sampled parameters. If any sort of rounding or truncation is applied then we might understate the acceptance rate because parameter values at different parts of the chain might be rounded or truncated such that they appear to be the same value and thus are not unique.

There is various guidance floating around in the Bayesian literature about what is an optimal acceptance rate for a particular MCMC algorithm. We'll not discuss those here as it depends on many things including but not limited to the type and implementation of the algorithm, number of dimensions in the parameter space, the degree of correlation in the parameter values, the aim of the study, etc.

Conclusion

By now we should have an understanding of why MCMC algorithms are needed to estimate the posterior distributions for parameters of interest and we should be comfortable with a couple popular ways to do so - Metropolis, Gibbs, and HMC. We should be familiar with these MCMC implementations including their basic machinery as well as their strengths and weaknesses. To help reinforce the simplest of these algorithms, the Metropolis algorithm, we have provided a basic implementation with the `R` software. We have also used the output of our Metropolis sampler to explore a handful of diagnostics which are useful in any Bayesian analysis to check samples for convergence, stationarity, healthy mixing across multiple chains, and efficiency with the ultimate goal of ensuring a suitable representation of the posterior distribution. We admire these powerful but simple algorithms which allow us to do something that would otherwise often be impossible, but we treat them with a healthy dose of skepticism too as they are just simple golems[3] that only follow the instructions we give them.

[3] A reference to the golems in chapter 1 of *Statistical Rethinking*