Basic MCMC for Bayesian Analysis

Heather Gaya

9/2/2021

Contents

Today I'm going to try to explain the idea behind MCMC algorithms. We use them a lot when doing Bayesian analysis of data, but I recently realized I never really understood HOW they worked. So I decided to dig into the basics a little bit and see what I could figure out! Turns out that basic MCMC isn't as scary as I thought and I hope you'll share the same sentiment when I'm done!

Getting Started

First, let's simulate some fake data from a binomial distribution. Maybe we're flipping a coin and want to know if it's fair. We flip it 20 times and these are the results we get:

```
set.seed(20)
x <- rbinom(20, size = 1, prob = 0.2)
x
##  [1] 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

Let's take a look at our data with a table:

```
table(x)
## x
##  0  1
## 16  4
```

Looks like we have 16 zeros ("failures") and 4 1's ("successes"). Sooo probably not a fair coin. But what if we want to estimate the probability of success?

Slogging Through Numbers by Hand

If we reallllllly wanted to, we could try to figure this out entire "by hand" by guessing a bunch of values and using the probability density function of a binomial distribution to tell us if we were close to the correct value. Here's how that might look:

We know that the equation for a binomial distribution with size 20 is: $\frac{20!}{(20-X!)X!}(p^X)(1-p)^{20-X}$

So how do we go about finding p? Well, to start, what's the probability density of 4 successes if p = .1?

```
dbinom(4, 20, 0.1)
## [1] 0.08977883
```

Okay, not too high.

How about if p = .4?

```
dbinom(4, 20, 0.4)
## [1] 0.03499079
```

Hmm, a little lower.

What about p = .7?

```
dbinom(4, 20, 0.7)
## [1] 5.007558e-06
```

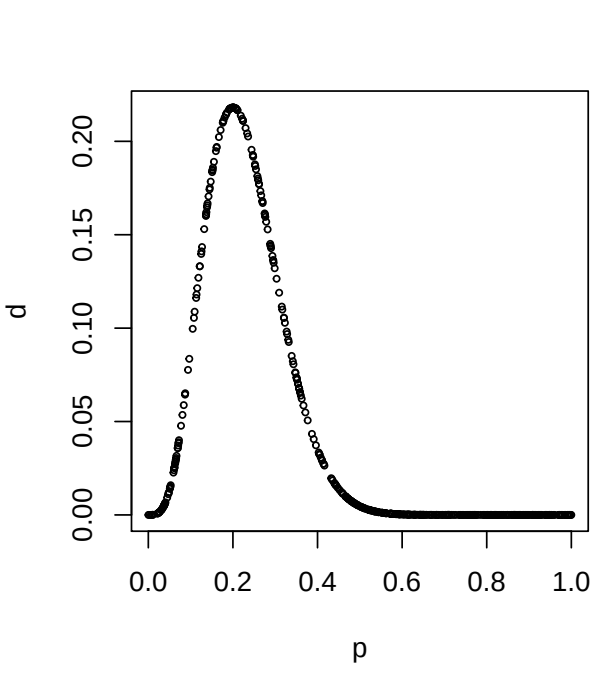
Oof, lower again.

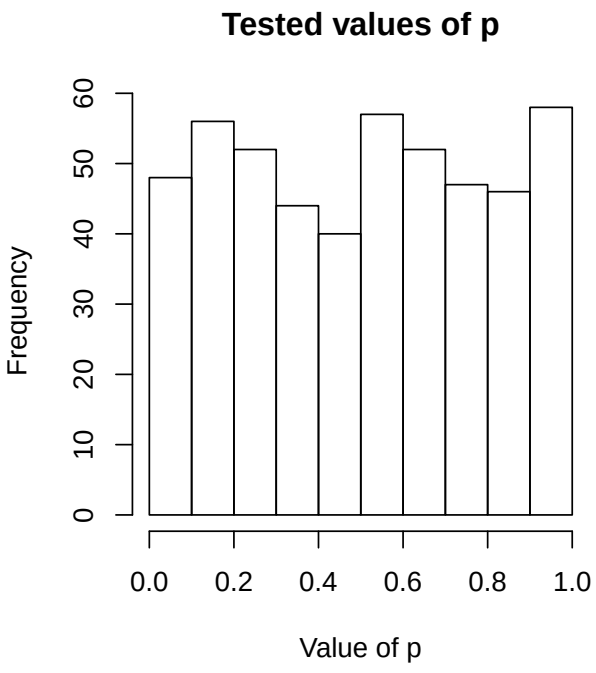
So we could do this over and over and over again and eventually we'd see that some numbers were larger (yay) and some were smaller (less good answers). If we guessed enough numbers, we'd probably find a few that were highest and we could say that p was pretty close to whatever that value was.

Making a Function Do This

Obviously picking random values of p by hand isn't very efficient. Let's have a function do it for us and graph what the result is:

```
pick.ps <- function(n) {
    p <- runif(n, min = 0, max = 1)
    d <- dbinom(4, 20, p)
    return(data.frame(p = p, d = d))
}
```





Awesome! We can see that the value of p is probably close to .3 because that's where the highest probability density was, but we can also see that this method tried a bunch of points (AKA wastes a lot of time) over around values that we basically know aren't going to be good answers. For instance, we're pretty sure this coin isn't going to have a success probability of .9, so we would kind of prefer that the model not spend so much time testing values over there.

Is there a way we could have the function spend more time in "good" areas, trying to estimate the exact value of p and less time in "bad" areas where the answer is very unlikely?

Metropolis Algorithm for a Binomial

Why yes! The simplest form of MCMC is a Metropolis-Hasting algorithm. First, we start on a value that's possible for p (so, something between 0 and 1).

```
start <- runif(1, min = 0, max = 1)
start
## [1] 0.7986704
```

Then we choose another value, called our proposal value.

```
proposed.p <- runif(1, min = 0, max = 1)
proposed.p
## [1] 0.3539545
```

Next we want to calculate the ratio of probability densities between our new value and our original value. If our ratio is greater than 1, we can just calculate it as 1.

```
alpha <- min(1, dbinom(4, 20, proposed.p)/dbinom(4, 20, start))
alpha
## [1] 1
```

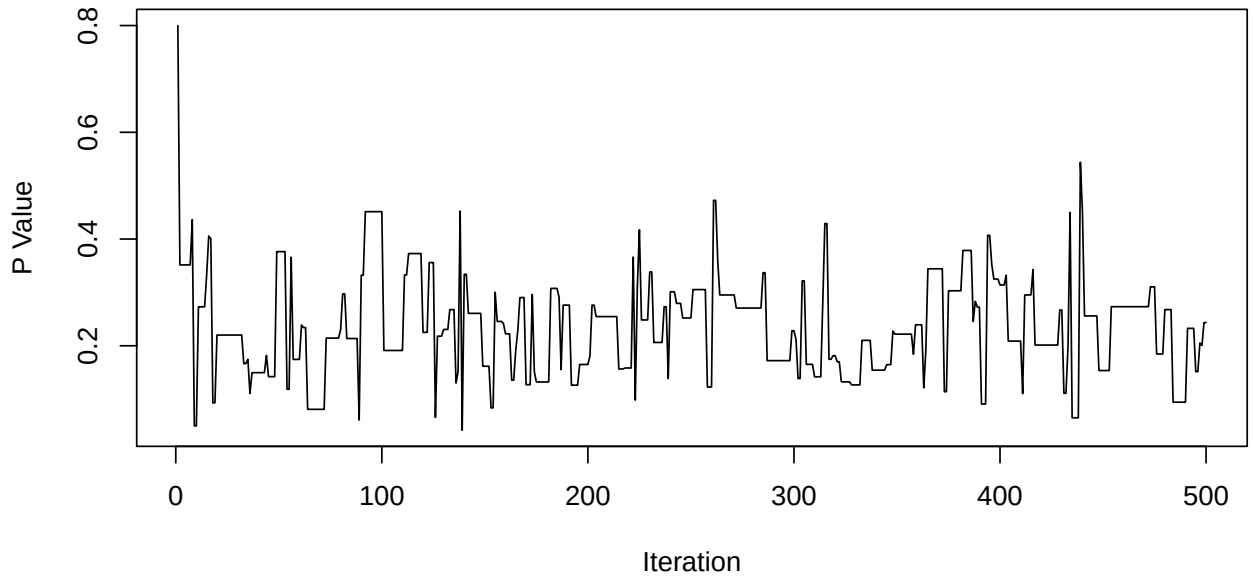
Then we use this value alpha to randomly "flip a coin" and tell us if we should accept the new value or keep the old value. So if the proposed p has a much higher probability density then we will tell the function "hey go sample over there, that looked good." Otherwise we'll stay where we are.

```
if (runif(1) < alpha) start <- proposed.p
```

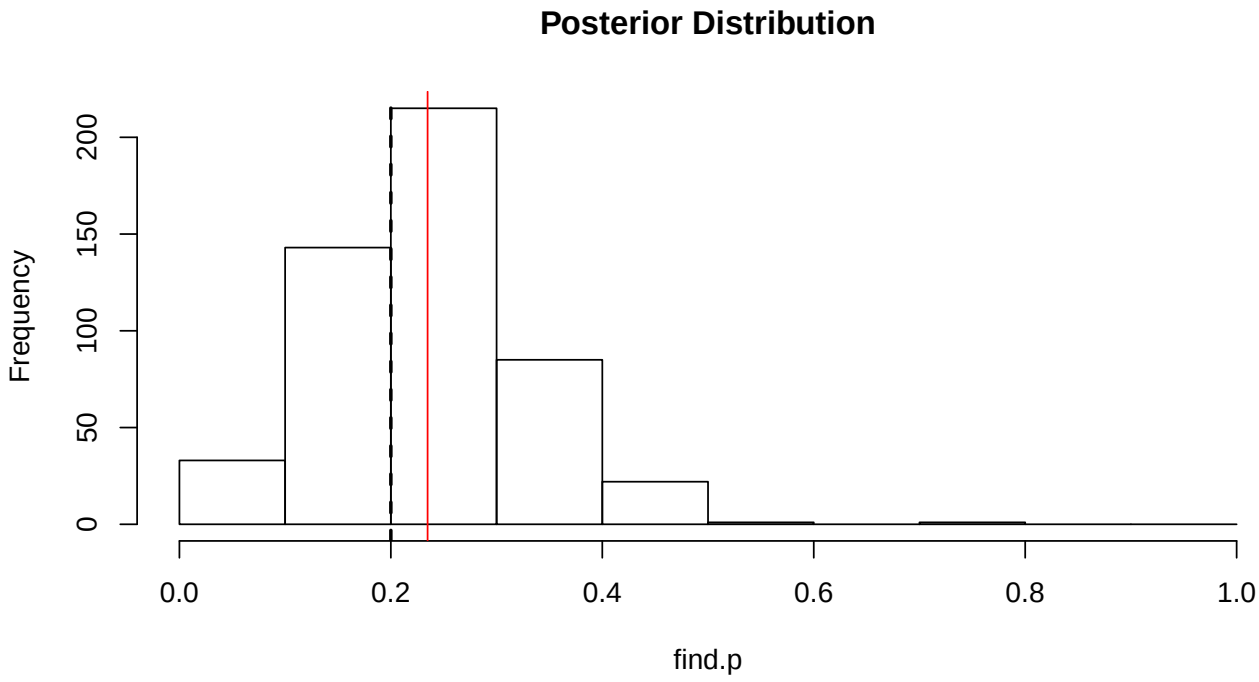
In this case, we like the new value better. Let's repeat this process a bunch of times and keep track of all the values we "accept".

```
binom.samp <- function(start, steps) {
    res <- array(NA, dim = steps)  #make an empty array to hold results
    p <- start
    res[1] <- p  #gotta start somewhere
    for (i in 2:steps) {
        new.p <- runif(1)  #propose a new point
        alpha <- min(1, dbinom(4, 20, new.p)/dbinom(4, 20, p))  #acceptance prob
        if (runif(1) < alpha)
            p <- new.p  #if accepted, proposed value is new p; otherwise it will keep the old p
        res[i] <- p  #record results
    }
    return(res)

}
```

```
find.p <- binom.samp(start = 0.8, steps = 500)  #run the function 500 times
plot(1:500, find.p, type = "l", xlab = "Iteration", ylab = "P Value")  #plot it
```



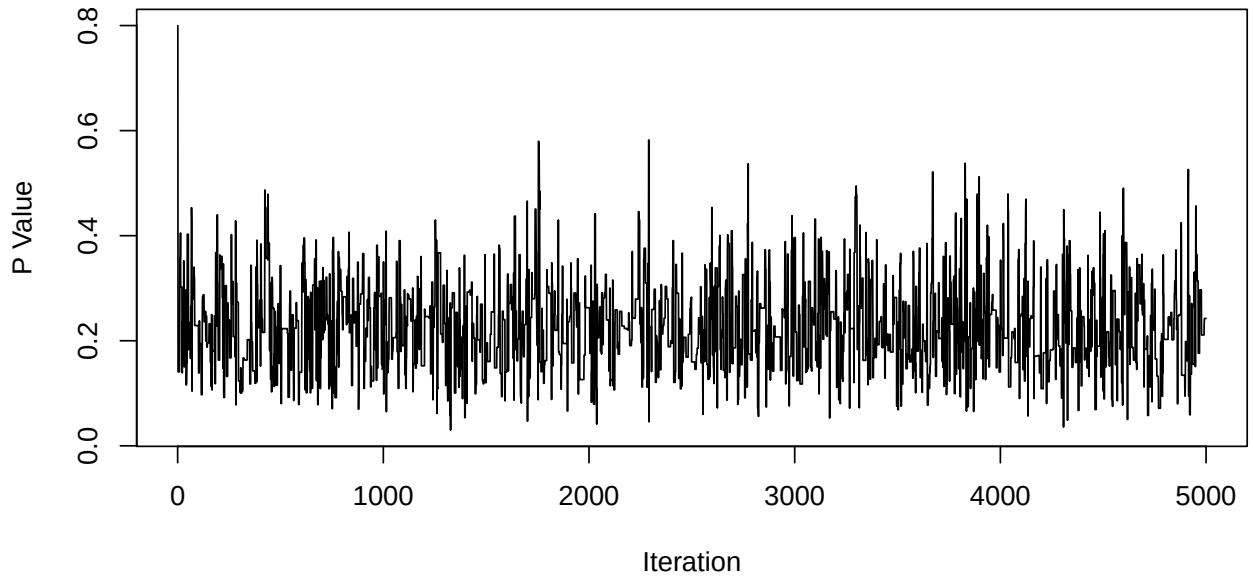

```
hist(find.p, breaks = seq(0, 1, by = 0.1), main = "Posterior Distribution")
abline(v = mean(find.p), col = "red")
abline(v = 0.2, lty = 2, lwd = 2)  #true value
```



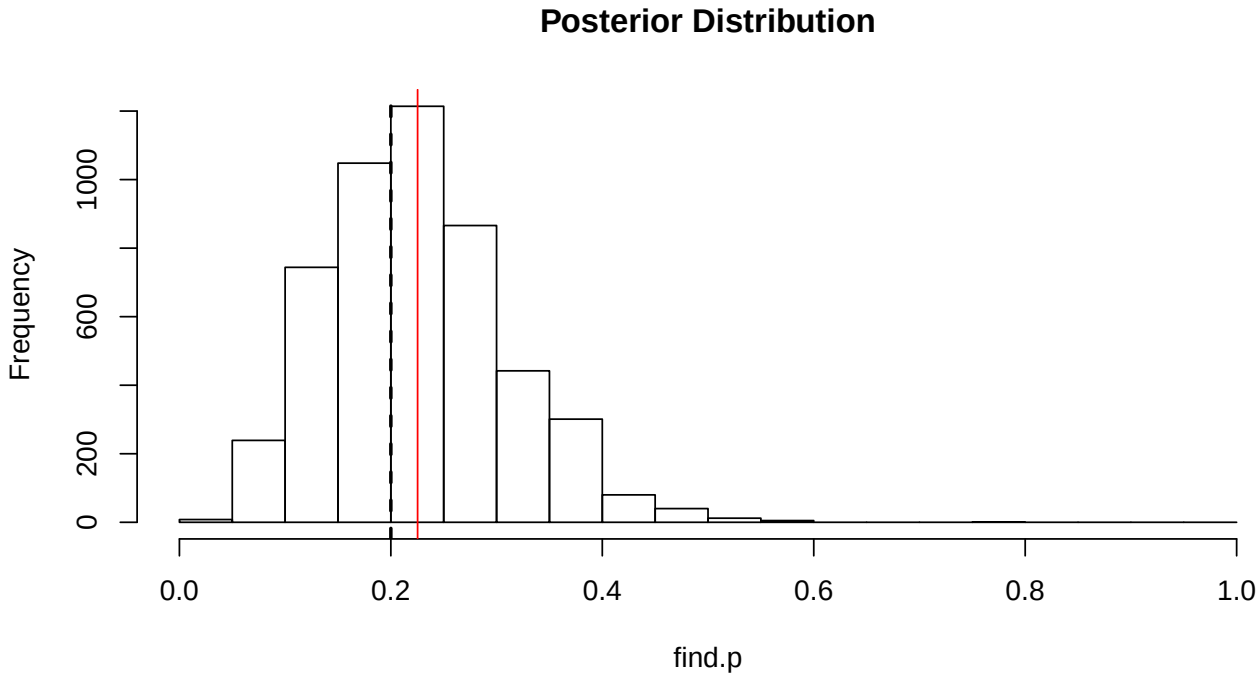

```
mean(find.p)
## [1] 0.2347525
```

Awesome, that's getting there! The true value can be seen in black and the red value is what our algorithm says is the mean accepted value of p. Let's see what happens if we run if for longer.

```
find.p <- binom.samp(start = 0.8, steps = 5000)
plot(1:5000, find.p, type = "l", xlab = "Iteration", ylab = "P Value")
```




```
hist(find.p, breaks = seq(0, 1, by = 0.05), main = "Posterior Distribution")
abline(v = mean(find.p), col = "red")
abline(v = 0.2, lty = 2, lwd = 2)  #true value
```




```
mean(find.p)
## [1] 0.2253029
```

Hmm not much of a change. When this happens, it tells us that the results we have now are the best we can get with this sampler, meaning - we have reached our answer! Now we can just calculate the mean and 95% credible interval for our data.

```
quantile(find.p, c(0.05, 0.95))
##        5%       95%
## 0.1005357 0.3763786
mean(find.p)
```

```
## [1] 0.2253029
```

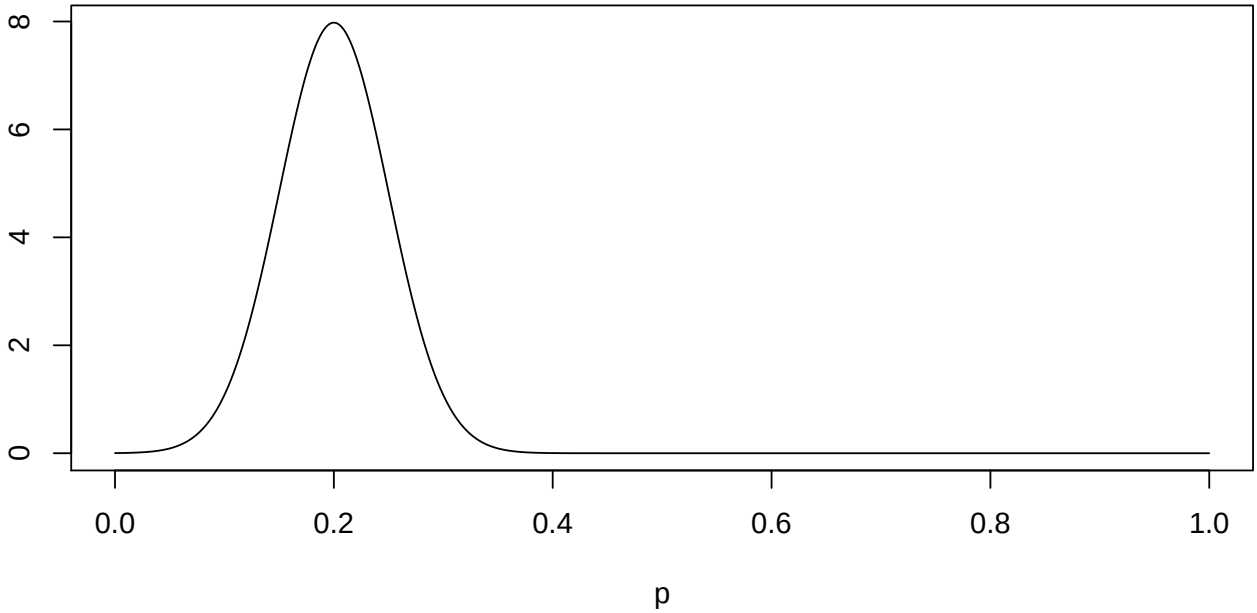
So we now know that the probability of a "success" with this coin is about 0.23 (0.1, 0.38). That's not too far off from the value we simulated (.20)

Can we make this more efficient?

Adding in a Prior

Currently we've been suggesting new values by randomly drawing numbers between 1 and 0 from a uniform distribution. But maybe we think we know a little bit about the coin we're flipping. Maybe our friend Bob has flipped this coin before and he tells you he's pretty sure it lands on heads 20% of the time. He's like really really really sure. How can we incorporate this into our algorithm?

Instead of drawing new values from runif(1) let's draw them from a normal distribution with a mean at .2 and a standard deviation of .05. Our prior looks like this:



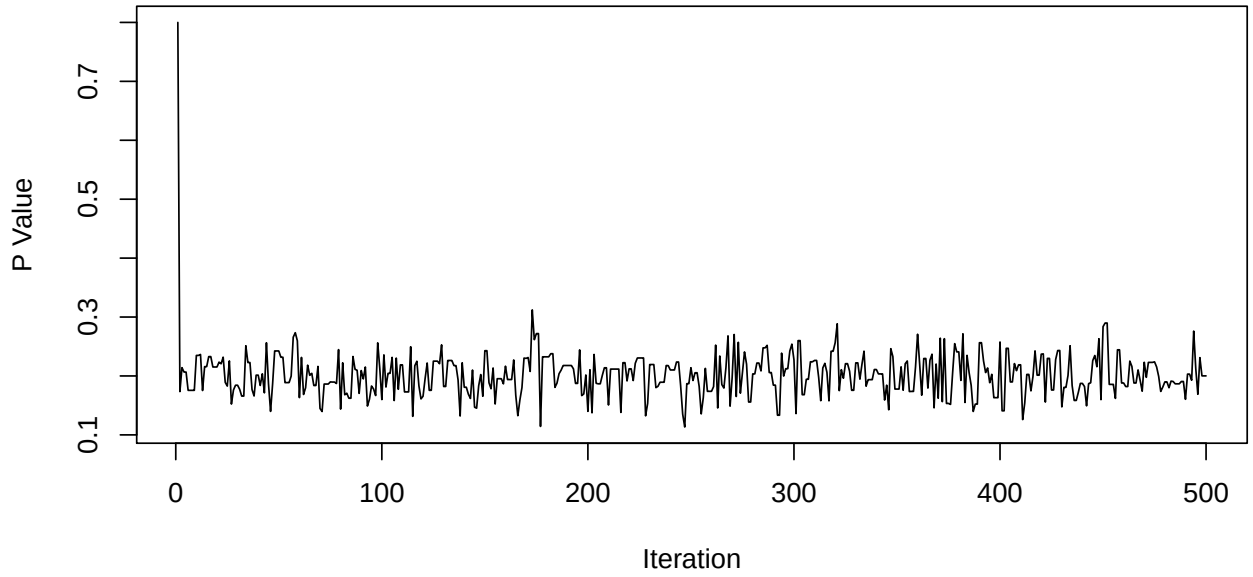
Since we have a prior there's another likelihood we need to consider here when we go about accepting or rejecting points. Not only do we want to maximize the likelihood that our data comes from the distribution with the parameters were suggesting, we also want to maximize the likelihood that the parameters themselves came from the prior distributions we're suggesting. So this means we want to know if our data is a good fit with the model we're suggesting (e.g. is .2 a good value of to match the data?) AND we want to know if the parameter we drew that make that model is a good fit with the prior (e.g. is .2 a likely value from our normal prior with mean =.2, sd = .05?).

Side note - this is easiest to do on the log scale! So we'll switch over to the log scale. Note that new/old is now exp(new-old) because of how logs work.

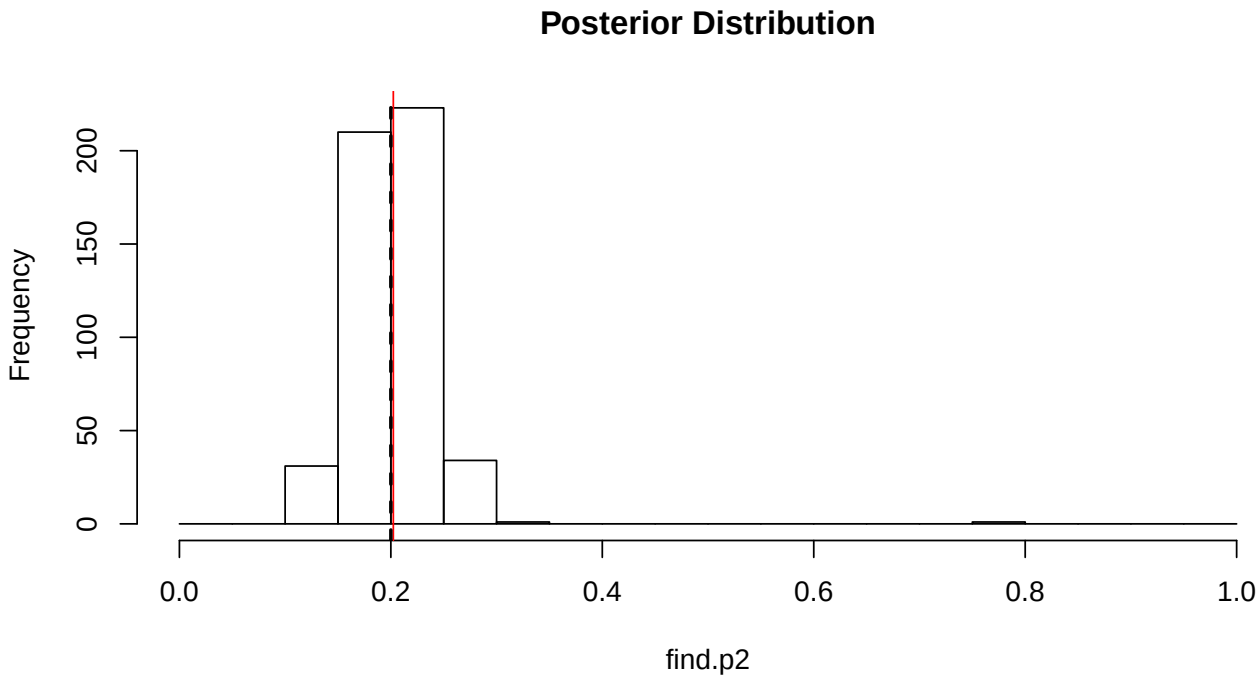
```
binom.samp2 <- function(start, steps) {
    res <- array(NA, dim = steps)  #make an empty array to hold results
    p <- start
    res[1] <- p
    for (i in 2:steps) {
        new.p <- rnorm(1, mean = 0.2, sd = 0.05)  #propose a new point
        old.like <- dbinom(4, 20, p, log = T) + dnorm(p, mean = 0.2, sd = 0.05,
            log = T)  #likelihood + prior likelihood
```

```
        new.like <- dbinom(4, 20, new.p, log = T) + dnorm(new.p, mean = 0.2,
            sd = 0.05, log = T)

        alpha <- min(1, exp(new.like - old.like))  #acceptance prob
        if (runif(1) < alpha)
            p <- new.p  #if accepted, proposed value is new p
        res[i] <- p  #record results
    }
    return(res)
}
```

```
find.p2 <- binom.samp2(start = 0.8, steps = 500)
plot(1:500, find.p2, type = "l", xlab = "Iteration", ylab = "P Value")
```




```
hist(find.p2, breaks = seq(0, 1, by = 0.05), main = "Posterior Distribution")
abline(v = mean(find.p2), col = "red")
abline(v = 0.2, lty = 2, lwd = 2)  #true value
```



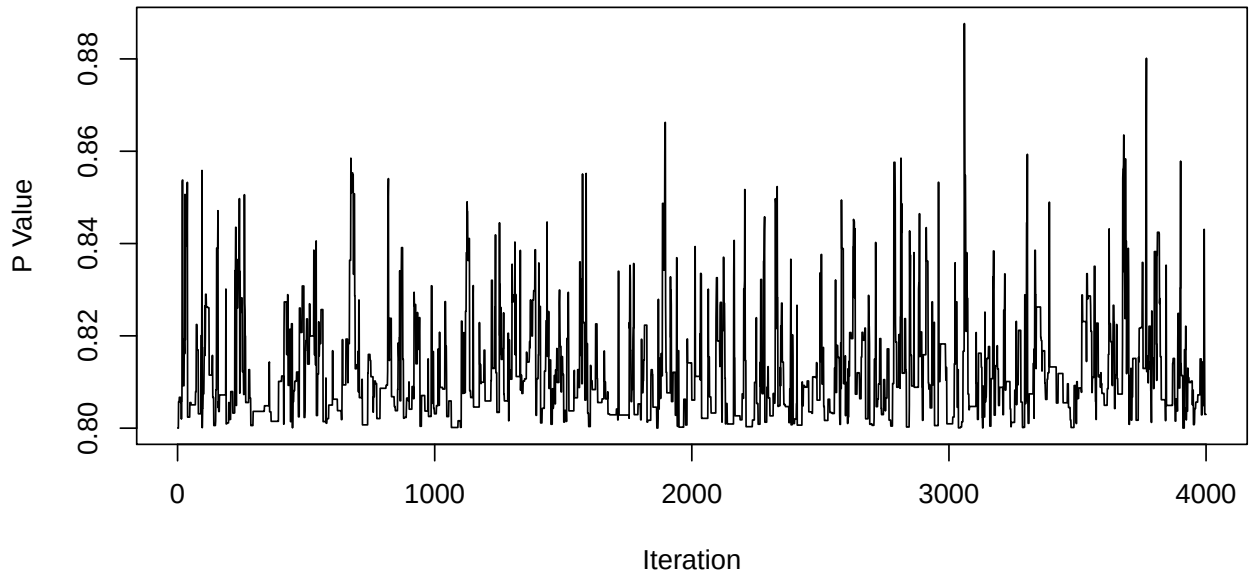

```
mean(find.p2)
## [1] 0.2023246
```

With this method we can see that our estimates are a lot more precise - about 0.2 (0.15, 0.26).

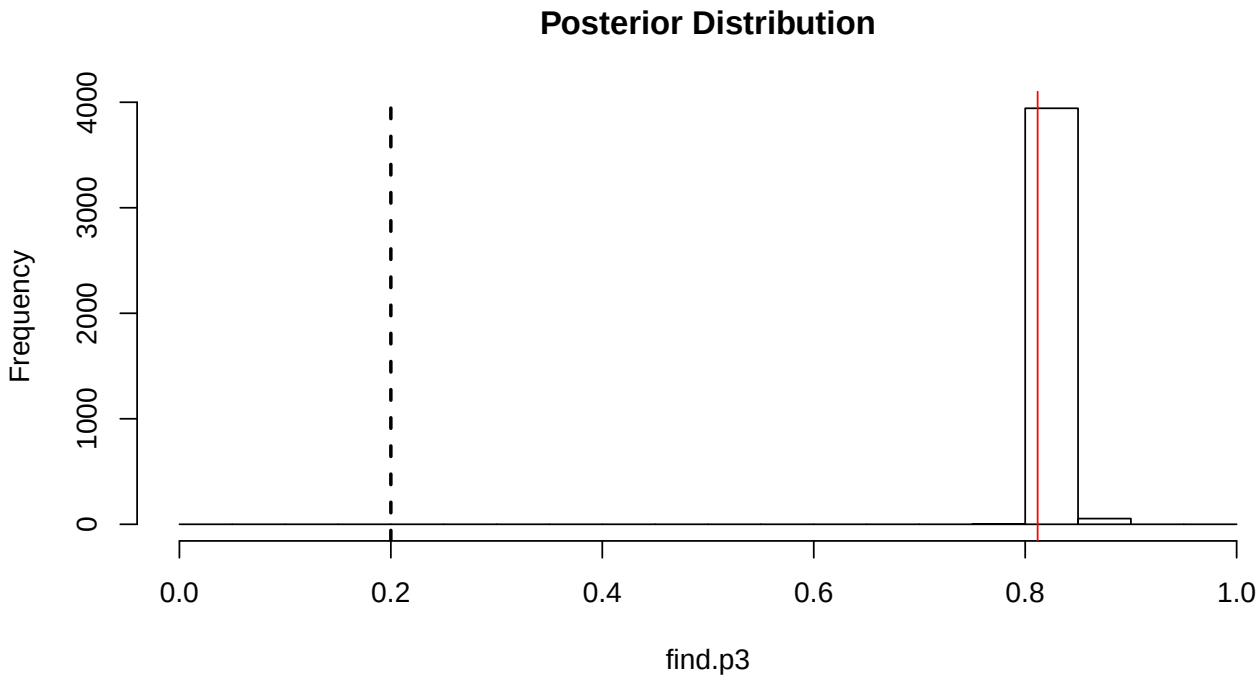
Notice however that we could seriously run into errors here if we suggest a really bad strong prior. Let's say for whatever reason we ask our cousin Ed about the probability of success of this coin and he tells you it has a 80%-90% chance of success. Here's what happens if we use a restrictive uniform prior of $p \sim U(.8, .9)$

```
binom.samp.bad <- function(start, steps) {
    res <- array(NA, dim = steps)  #make an empty array to hold results
    p <- start
    res[1] <- p
    for (i in 2:steps) {
        new.p <- runif(1, 0.8, 0.9)  #propose a new point
        old.like <- dbinom(4, 20, p, log = T) + dunif(p, 0.8, 0.9, log = T)  #likelihood + prior likeli
        new.like <- dbinom(4, 20, new.p, log = T) + dunif(new.p, 0.8, 0.9,
            log = T)

        alpha <- min(1, exp(new.like - old.like))  #acceptance prob
        if (runif(1) < alpha)
            p <- new.p  #if accepted, proposed value is new p
        res[i] <- p  #record results
    }
    return(res)
}
```

```
find.p3 <- binom.samp.bad(start = 0.8, steps = 4000)
plot(1:4000, find.p3, type = "l", xlab = "Iteration", ylab = "P Value")
```




```
hist(find.p3, breaks = seq(0, 1, by = 0.05), main = "Posterior Distribution")
abline(v = mean(find.p3), col = "red")
abline(v = 0.2, lty = 2, lwd = 2)  #true value
```




```
mean(find.p3)
## [1] 0.8117603
```

Even if we run this for a very long time, we won't get the answer we're looking for. We can tell something is wrong because the values are bumping up against p = 0.8. They "want" to be lower, but we told the model it could only use values between .8 and .9, so it's forced to stay between those bounds.

Let's move on to a more complicated scenario.

Linear Regression and MCMC

Let's say we have some data on baby turtles that we're rearing in captivity. Maybe we want to know how much we can expect our turtles to weigh based on how many hours they spend basking under a heat-lamp each week.

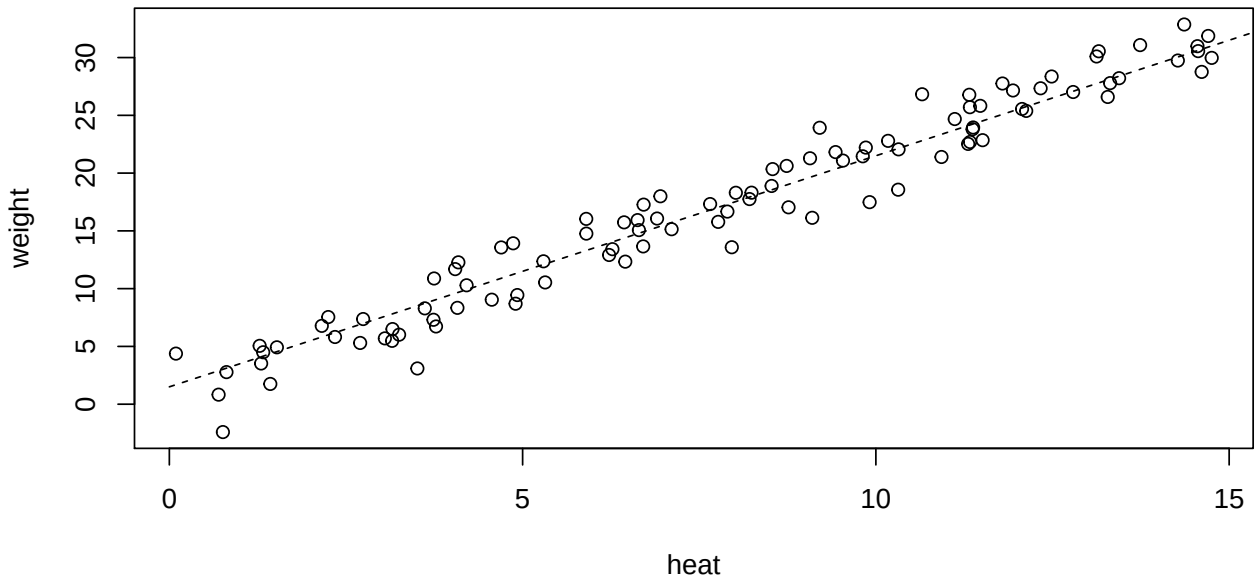
So our model is $weight = \beta_0 + \beta_1 * heat + \epsilon$

First let's make up some data and add a bit of error:

```
heat <- runif(100, min = 0, max = 15)
weight <- 1.5 + 2 * heat + rnorm(50, mean = 0, sd = 2)
```

Here's our data:

```
plot(heat, weight)
lines(c(0, 20), c(1.5, 41.5), lty = 2)
```



Let's try to estimate those two beta values and the sd of our error. This sounds complicated but it's not so bad.

First, we need to have some priors for these values. We have no idea but we can always change them if it seems like we chose restrictive priors. For now let's make them uniforms ranging from -5 to 5. In standard linear regression the underlying assumption is that the errors are normally distributed around the mean (the line) but we'll want to also estimate what sd might be. We'll just choose sd from a uniform distribution. This isn't an awesome model because there's a lot of uncertainty but it's a good start.

So our model is: $weight = \beta_0 + \beta_1 * heat + \epsilon$ with $\epsilon \sim N(0, sd)$

$sd \sim U(0, 5)$

$\beta_0 \sim U(-5, 5)$

$\beta_1 \sim U(-5, 5)$

Like before, we can start with some values of our choosing - let's go with 0 and 1 for the two betas and 1 for the sd.

```
beta0 <- 0
beta1 <- 1
sd <- 1
```

Next we need to calculate the likelihood of these values producing the data we have (the weights data). It's going to be easiest if we do this on the log scale, which is just a simple matter of adding "log = T" to the built in R functions.

First, we need to see what weight values would be predicted if our starting values were correct:

```
pred.weight.start <- beta0 + beta1 * heat
```

Remember that our model says that the expected values are going to follow a linear pattern with the residuals following a normal pattern. AKA, the exact values might not be directly on this linear regression but they should be evenly far away from it on either side. So we can ask R what the likelihood is of each of these values and then sum them all together to get the likelihood for the entire data set. Notice that once again will do this as two parts - one for the likelihood that the data fits with the model params and one for the likelihood that the chosen values fit their priors.

```
start.like.data <- sum(dnorm(weight, mean = pred.weight.start, sd = sd,
    log = T))
beta0prior = dunif(beta0, min = -5, max = 5, log = T)
beta1prior = dunif(beta1, min = -5, max = 5, log = T)
sdprior = dunif(sd, min = 0, max = 5, log = T)
```

This might seem a little silly when it's all uniform distributions but if we change our priors (as we will later on), this concept will be important!

Now our full likelihood for these values that we've chosen for beta0, beta1 and sd is:

```
start.like <- start.like.data + beta0prior + beta1prior + sdprior
```

Then we choose our proposal values.

```
proposed.beta0 <- runif(1, -5, 5)
proposed.beta1 <- runif(1, -5, 5)
proposed.sd <- runif(1, 0, 5)
```

Calculate the expected values from these proposed values and get the likelihood.

```
pred.weight.prop <- proposed.beta0 + proposed.beta1 * heat
proposed.like.data <- sum(dnorm(weight, mean = pred.weight.prop, sd = proposed.sd,
    log = T))
proposed.beta0prior = dunif(proposed.beta0, min = -5, max = 5, log = T)
proposed.beta1prior = dunif(proposed.beta1, min = -5, max = 5, log = T)
proposed.sdprior = dunif(proposed.sd, min = 0, max = 5, log = T)
proposed.like <- proposed.like.data + proposed.beta0prior + proposed.beta1prior +
    proposed.sdprior
```

Now we can compare if the new values are better, just like we did with the simple binomial example. One small thing to note - since we're on the log scale we want to do exp(proposed-start) which is just proposed/start (but on the log scale).

```
alpha <- min(1, exp(proposed.like - start.like))
```

If the proposal is accepted, these proposed values become our new values. Otherwise they stay the same as before. Then we repeat that process a bunch!

```
if (runif(1) < alpha) {
    beta0 <- proposed.beta0
    beta1 <- proposed.beta1
    sd <- proposed.beta1
}
```

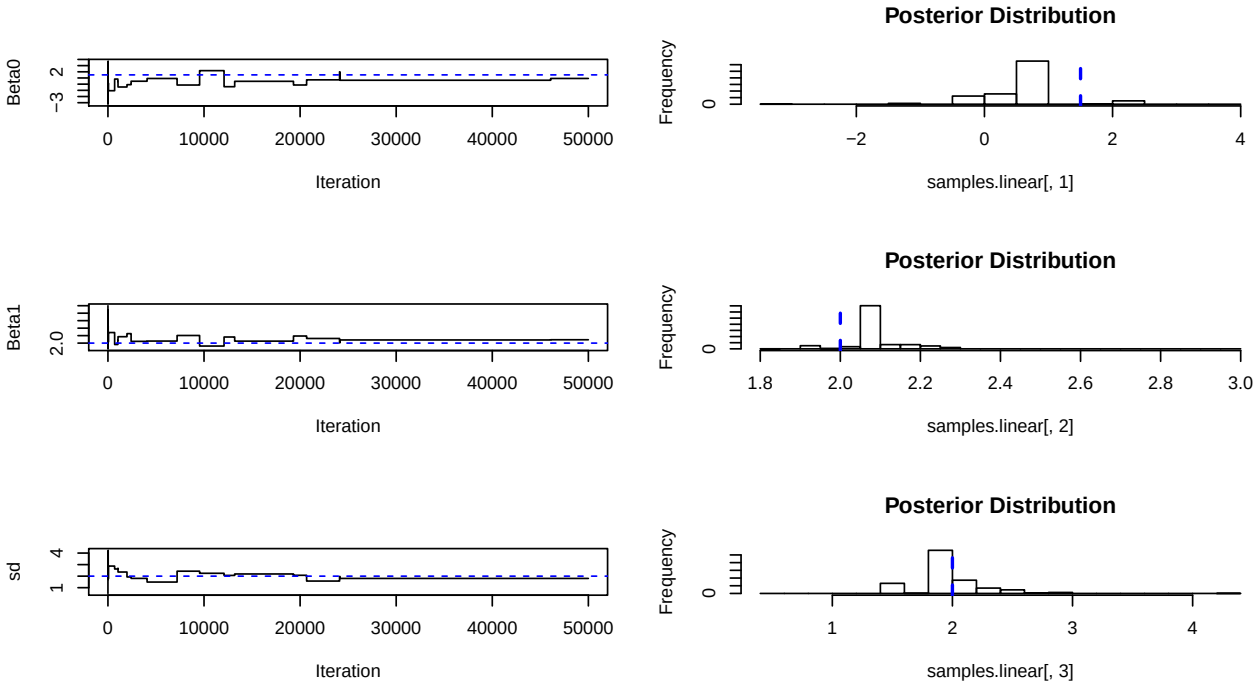
Let's see how this looks after many iterations. We can make a handy function to do this for us:

```
lin.samp <- function(beta0, beta1, sd, steps) {
    res <- array(NA, dim = c(steps, 3))  #make an empty array to hold results
    beta0 <- beta0
    beta1 <- beta1
    sd <- sd
    res[1, ] <- c(beta0, beta1, sd)
    for (i in 2:steps) {
        proposed.beta0 <- runif(1, -5, 5)
        proposed.beta1 <- runif(1, -5, 5)
        proposed.sd <- runif(1, 0, 5)

        # calculate for current values
        pred.weight.1 <- beta0 + beta1 * heat
        pred.like.data <- sum(dnorm(weight, mean = pred.weight.1, sd = sd,
            log = T))
        beta0prior <- dunif(beta0, min = -5, max = 5, log = T)
        beta1prior <- dunif(beta1, min = -5, max = 5, log = T)
        sdprior <- dunif(sd, min = 0, max = 5, log = T)

        pred.like <- pred.like.data + beta0prior + beta1prior + sdprior  #full current likelihood

        # calculate for proposed values
        pred.weight.prop <- proposed.beta0 + proposed.beta1 * heat
        proposed.like.data <- sum(dnorm(weight, mean = pred.weight.prop,
            sd = proposed.sd, log = T))
        proposed.beta0prior = dunif(proposed.beta0, min = -5, max = 5,
            log = T)
        proposed.beta1prior = dunif(proposed.beta1, min = -5, max = 5,
            log = T)
        proposed.sdprior = dunif(proposed.sd, min = 0, max = 5, log = T)

        proposed.like <- proposed.like.data + proposed.beta0prior + proposed.beta1prior +
            proposed.sdprior  #full proposed likelihood

        alpha <- min(1, exp(proposed.like - pred.like))  #acceptance prob

        if (runif(1) < alpha) {
            # change values if accepted then do it again
            beta0 <- proposed.beta0
            beta1 <- proposed.beta1
            sd <- proposed.sd
        }

        res[i, ] <- c(beta0, beta1, sd)  #record results
    }
    return(res)
}
```

Time to run!

```
samples.linear <- lin.samp(beta0 = 0, beta1 = 3, sd = 0.5, steps = 50000)
```

```
par(mfrow = c(3, 2))
plot(1:50000, samples.linear[, 1], type = "l", xlab = "Iteration", ylab = "Beta0")
abline(h = 1.5, col = "blue", lty = 2)
hist(samples.linear[, 1], main = "Posterior Distribution")
abline(v = 1.5, lty = 2, lwd = 2, col = "blue")  #true value
plot(1:50000, samples.linear[, 2], type = "l", xlab = "Iteration", ylab = "Beta1")
abline(h = 2, col = "blue", lty = 2)
hist(samples.linear[, 2], main = "Posterior Distribution")
abline(v = 2, lty = 2, lwd = 2, col = "blue")  #true value
plot(1:50000, samples.linear[, 3], type = "l", xlab = "Iteration", ylab = "sd")
abline(h = 2, col = "blue", lty = 2)
hist(samples.linear[, 3], main = "Posterior Distribution")
abline(v = 2, lty = 2, lwd = 2, col = "blue")  #true value
```



Awesome! We got some okay estimates there, so we could leave it at that, but you might notice that the chains seem to stay at certain values for a long time and don't really explore the space very well. If this were a large dataset and we had more parameters, this would take a really long time for us to get a reasonable answer. Let's see what happens when we adjust our priors from uniform to a normal for the beta parameters.

We only have to adjust a few parts of the original function. First, when we draw proposed values we need to make sure we draw from the correct distributions. Next, we also want to calculate the likelihood of that value from the correct prior distribution. Let's change the priors on beta0 and beta1. I've marked the points in the function that changed with comments:

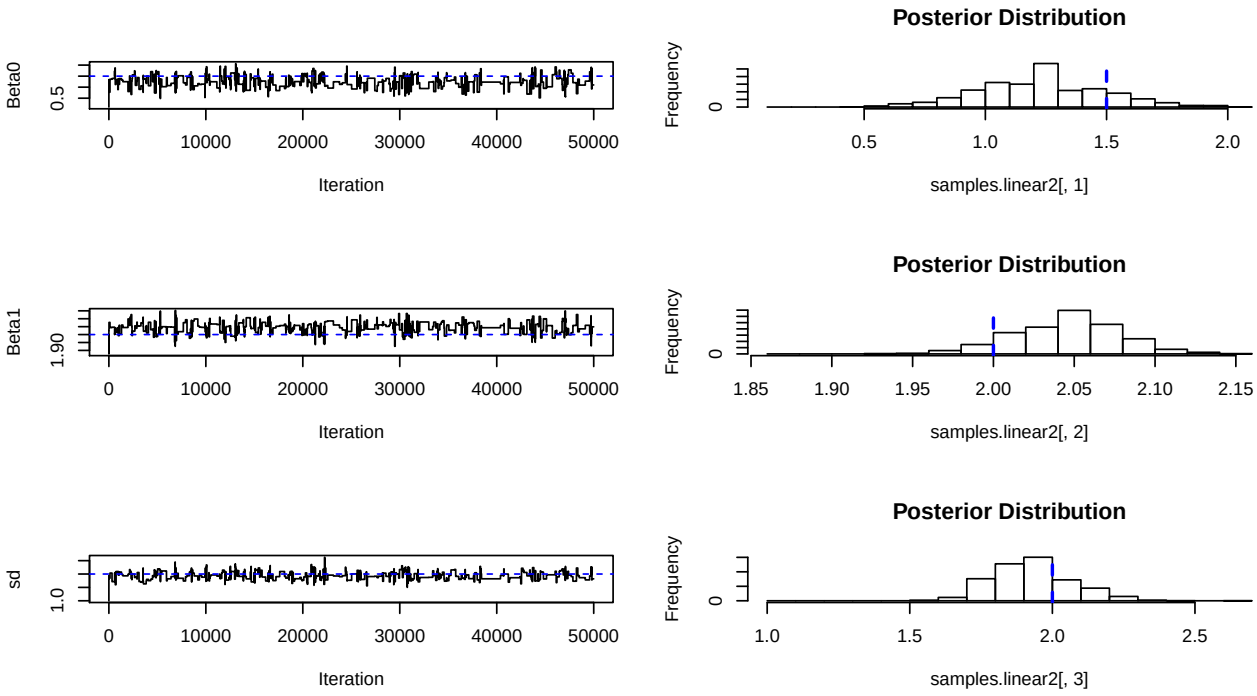
```
lin.samp.norms <- function(beta0, beta1, sd, steps) {
    res <- array(NA, dim = c(steps, 3))  #make an empty array to hold results
    beta0 <- beta0
    beta1 <- beta1
    sd <- sd
    res[1, ] <- c(beta0, beta1, sd)
    for (i in 2:steps) {
        proposed.beta0 <- rnorm(1, mean = 1.5, sd = 0.5)  #we changed this
        proposed.beta1 <- rnorm(1, mean = 2, sd = 0.25)  #we also changed this
```

```
        proposed.sd <- runif(1, 0, 5)

        pred.weight.1 <- beta0 + beta1 * heat
        pred.like.data <- sum(dnorm(weight, mean = pred.weight.1, sd = sd,
            log = T))
        beta0prior <- dnorm(beta0, mean = 1.5, sd = 0.5, log = T)  #so we have to change this
        beta1prior <- dnorm(beta1, mean = 2, sd = 0.25, log = T)  #and this
        sdprior <- dunif(sd, min = 0, max = 5, log = T)

        pred.like <- pred.like.data + beta0prior + beta1prior + sdprior  #full current likelihood

        pred.weight.prop <- proposed.beta0 + proposed.beta1 * heat
        proposed.like.data <- sum(dnorm(weight, mean = pred.weight.prop,
            sd = proposed.sd, log = T))
        proposed.beta0prior = dnorm(proposed.beta0, mean = 1.5, sd = 0.5,
            log = T)  #we need to change this
        proposed.beta1prior = dnorm(proposed.beta1, mean = 2, sd = 0.25,
            log = T)  #we also have to change this
        proposed.sdprior = dunif(proposed.sd, min = 0, max = 5, log = T)

        proposed.like <- proposed.like.data + proposed.beta0prior + proposed.beta1prior +
            proposed.sdprior  #full proposed likelihood

        alpha <- min(1, exp(proposed.like - pred.like))  #acceptance prob

        if (runif(1) < alpha) {
            beta0 <- proposed.beta0
            beta1 <- proposed.beta1
            sd <- proposed.sd
        }

        res[i, ] <- c(beta0, beta1, sd)  #record results
    }
    return(res)
}
```

Time to run it and graph the results.

```
samples.linear2 <- lin.samp.norms(beta0 = 0.1, beta1 = 2, sd = 1, steps = 50000)
```

```
par(mfrow = c(3, 2))
plot(1:50000, samples.linear2[, 1], type = "l", xlab = "Iteration", ylab = "Beta0")
abline(h = 1.5, col = "blue", lty = 2)
hist(samples.linear2[, 1], main = "Posterior Distribution")
abline(v = 1.5, lty = 2, lwd = 2, col = "blue")  #true value
plot(1:50000, samples.linear2[, 2], type = "l", xlab = "Iteration", ylab = "Beta1")
abline(h = 2, col = "blue", lty = 2)
hist(samples.linear2[, 2], main = "Posterior Distribution")
abline(v = 2, lty = 2, lwd = 2, col = "blue")  #true value
plot(1:50000, samples.linear2[, 3], type = "l", xlab = "Iteration", ylab = "sd")
abline(h = 2, col = "blue", lty = 2)
hist(samples.linear2[, 3], main = "Posterior Distribution")
abline(v = 2, lty = 2, lwd = 2, col = "blue")  #true value
```



Finally, we can compare the mean results of our beautiful MCMC efforts (both functions) with the true values we simulated. We can also quickly run an lm and see what that would give us.

```
lm(weight ~ heat)
##
## Call:
## lm(formula = weight ~ heat)
##
## Coefficients:
## (Intercept)         heat
##      0.7895       2.0913
```

```
##       True_Values Uniform_priors Normal_priors Lm_function
## Beta0         1.5      0.6016918      1.225948   0.7894898
## Beta1         2.0      2.0870534      2.046089   2.0913019
## sd            2.0      1.9108462      1.925022          NA
```

Sweet!

Hopefully this helped you understand the idea behind MCMC just a little bit more! I'm still learning myself, so lookout for future tutorials about this topic and other topics relevant to Bayesian Stats and Ecology! Feel free to shoot me an email at heather.e.gaya (at) gmail.com or find me on twitter (Doofgradstudent) if you have questions!