Lesson 4: Metropolis-Hastings

Lesson 4.1

Metropolis-Hastings is an algorithm that allows us to sample from a generic probability distribution (which we will call the target distribution), even if we do not know the normalizing constant. To do this, we construct and sample from a Markov chain whose stationary distribution is the target distribution. It consists of picking an arbitrary starting value, and iteratively accepting or rejecting candidate samples drawn from another distribution, one that is easy to sample.

The algorithm

Let's say we wish to produce samples from a target distribution $p(\theta) \propto g(\theta)$, where we don't know the normalizing constant (since $\int g(\theta)d\theta$ is hard or impossible to compute), so we only have $g(\theta)$ to work with. The Metropolis-Hastings algorithm proceeds as follows.

1. Select an initial value θ_0.
2. For $i = 1, \dots, m$, repeat the following steps:
    a. Draw a candidate sample θ^* from a proposal distribution $q(\theta^* \mid \theta_{i-1})$ (more on this later).
    b. Compute the ratio

$$\alpha = \frac{g(\theta^*)/q(\theta^* \mid \theta_{i-1})}{g(\theta_{i-1})/q(\theta_{i-1} \mid \theta^*)} = \frac{g(\theta^*)q(\theta_{i-1} \mid \theta^*)}{g(\theta_{i-1})q(\theta^* \mid \theta_{i-1})}.$$

    c. If $\alpha \geq 1$, then set $\theta_i = \theta^*$. If $\alpha < 1$, then set $\theta_i = \theta^*$ with probability α, or $\theta_i = \theta_{i-1}$ with probability $1 - \alpha$.

Steps 2b and 2c act as a correction since the proposal distribution is not the target distribution. At each step in the chain, we draw a candidate and decide whether to "move" the chain there or remain where we are. If the proposed move to the candidate is "advantageous," ($\alpha \geq 1$) we "move" there and if it is not "advantageous," we still might move there, but only with probability α. Since our decision to "move" to the candidate only depends on where the chain currently is, this is a Markov chain.

Proposal distribution

One careful choice we must make is the candidate generating distribution $q(\theta^* \mid \theta_{i-1})$. It may or may not depend on the previous iteration's value of θ. One example where it doesn't depend on the previous value would be if $q(\theta^*)$ is always the same distribution. If we use this option, $q(\theta)$ should be as similar as possible to $p(\theta)$.

Another popular option, one that does depend on the previous iteration, is Random-Walk Metropolis-Hastings. Here, the proposal distribution is centered on θ_{i-1}. For instance, it might be a normal distribution with mean θ_{i-1}. Because the normal distribution is symmetric, this example comes with another advantage: $q(\theta^* \mid \theta_{i-1}) = q(\theta_{i-1} \mid \theta^*)$, causing it to cancel out when we calculate α. Thus, in Random-Walk Metropolis-Hastings where the candidate is drawn from a normal with mean θ_{i-1} and constant variance, the acceptance ratio is $\alpha = g(\theta^*)/g(\theta_{i-1})$.

Acceptance rate

Clearly, not all candidate draws are accepted, so our Markov chain sometimes "stays" where it is, possibly for many iterations. How often you want the chain to accept candidates depends on the type of algorithm you use. If you approximate $p(\theta)$ with $q(\theta^*)$ and always draw candidates from that, accepting candidates often is

good; it means $q(\theta^*)$ is approximating $p(\theta)$ well. However, you still may want q to have a larger variance than p and see some rejection of candidates as an assurance that q is covering the space well.

As we will see in coming examples, a high acceptance rate for the Random-Walk Metropolis-Hastings sampler is not a good thing. If the random walk is taking too small of steps, it will accept often, but will take a very long time to fully explore the posterior. If the random walk is taking too large of steps, many of its proposals will have low probability and the acceptance rate will be low, wasting many draws. Ideally, a random walk sampler should accept somewhere between 23% and 50% of the candidates proposed.

In the next segment, we will see a demonstration of this algorithm used in a discrete case, where we can show mathematically that the Markov chain converges to the target distribution. In the following segment, we will demonstrate coding a Random-Walk Metropolis-Hastings algorithm in R to solve one of the problems from the end of Lesson 2.

Lesson 4.2

Guest lecture by Herbie, demonstration in discrete case

Lesson 4.3

Random walk with normal likelihood, t prior

Recall the model from the last segment of Lesson 2 where the data are the percent change in total personnel from last year to this year for $n = 10$ companies. We used a normal likelihood with known variance and t distribution for the prior on the unknown mean. Suppose the values are $y = (1.2, 1.4, -0.5, 0.3, 0.9, 2.3, 1.0, 0.1, 1.3, 1.9)$. Because this model is not conjugate, the posterior distribution is not in a standard form that we can easily sample. To obtain posterior samples, we will set up a Markov chain whose stationary distribution is this posterior distribution.

Recall that the posterior distribution is

$$p(\mu \mid y_1, \ldots, y_n) \propto \frac{\exp[n(\bar{y}\mu - \mu^2/2)]}{1 + \mu^2}$$

The posterior distribution on the left is our target distribution and the expression on the right is our $g(\mu)$.

The first thing we can do in R is write a function to evaluate $g(\mu)$. Because posterior distributions include likelihoods (the product of many numbers that are potentially small), $g(\mu)$ might evaluate to such a small number that to the computer, it effectively zero. This will cause a problem when we evaluate the acceptance ratio α. To avoid this problem, we can work on the log scale, which will be more numerically stable. Thus, we will write a function to evaluate

$$\log(g(\mu)) = n(\bar{y}\mu - \mu^2/2) - \log(1 + \mu^2)$$

This function will require three arguments, μ, $\bar{y}$, and n.

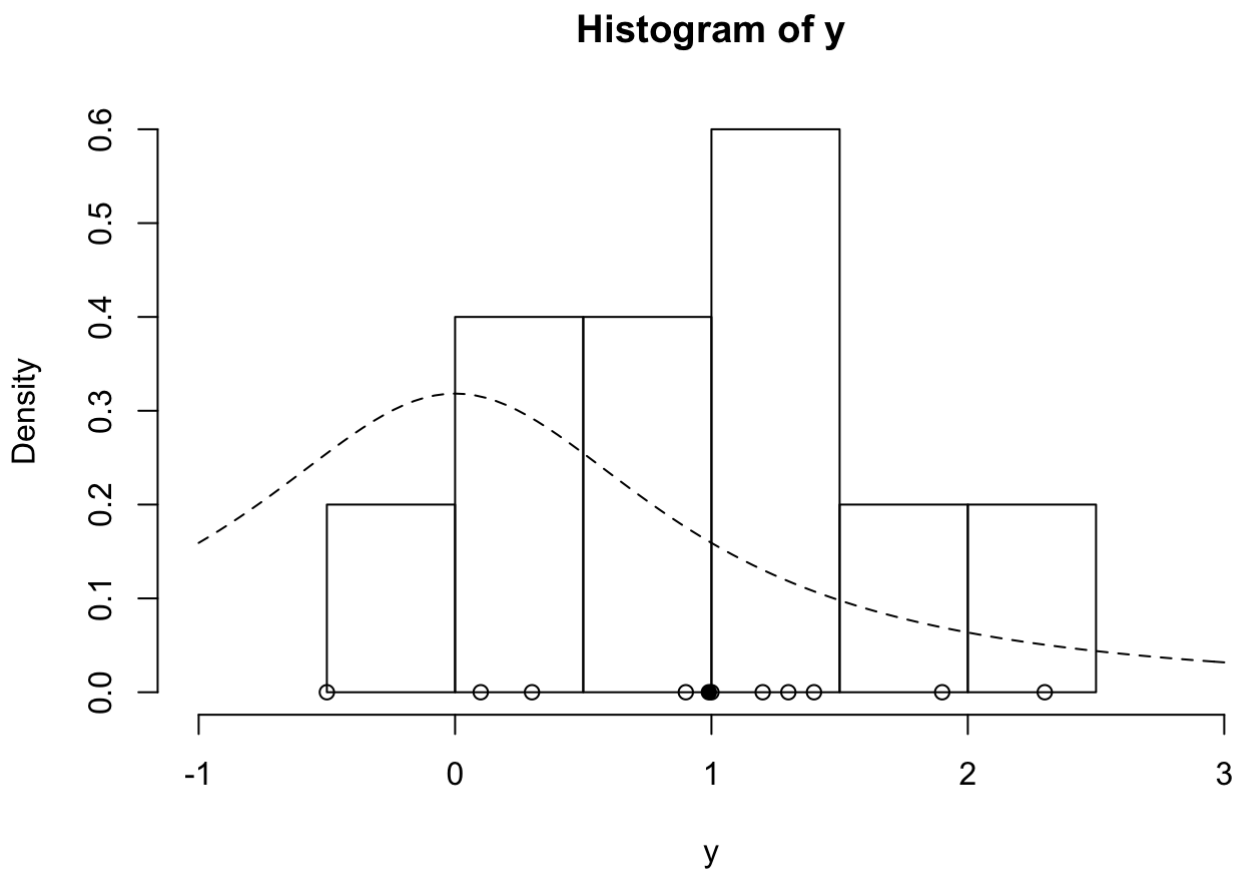
```
lg = function(mu, n, ybar) {
  mu2 = mu^2
  n * (ybar * mu - mu2 / 2.0) - log(1 + mu2)
}
```

Next, let's write a function to execute the Random-Walk Metropolis-Hastings sampler with normal proposals.

```
mh = function(n, ybar, n_iter, mu_init, cand_sd) {
  ## Random-Walk Metropolis-Hastings algorithm

  ## step 1, initialize
  mu_out = numeric(n_iter)
  accpt = 0
  mu_now = mu_init
  lg_now = lg(mu=mu_now, n=n, ybar=ybar)

  ## step 2, iterate
  for (i in 1:n_iter) {
    ## step 2a
    mu_cand = rnorm(n=1, mean=mu_now, sd=cand_sd) # draw a candidate

    ## step 2b
    lg_cand = lg(mu=mu_cand, n=n, ybar=ybar) # evaluate log of g with the candidate
    lalpha = lg_cand - lg_now # log of acceptance ratio
    alpha = exp(lalpha)

    ## step 2c
    u = runif(1) # draw a uniform variable which will be less than alpha with probabi
lity min(1, alpha)
    if (u < alpha) { # then accept the candidate
      mu_now = mu_cand
      accpt = accpt + 1 # to keep track of acceptance
      lg_now = lg_cand
    }

    ## collect results
    mu_out[i] = mu_now # save this iteration's value of mu
  }

  ## return a list of output
  list(mu=mu_out, accpt=accpt/n_iter)
}
```

Now, let's set up the problem.

```
y = c(1.2, 1.4, -0.5, 0.3, 0.9, 2.3, 1.0, 0.1, 1.3, 1.9)
ybar = mean(y)
n = length(y)
hist(y, freq=FALSE, xlim=c(-1.0, 3.0)) # histogram of the data
curve(dt(x=x, df=1), lty=2, add=TRUE) # prior for mu
points(y, rep(0,n), pch=1) # individual data points
points(ybar, 0, pch=19) # sample mean
```

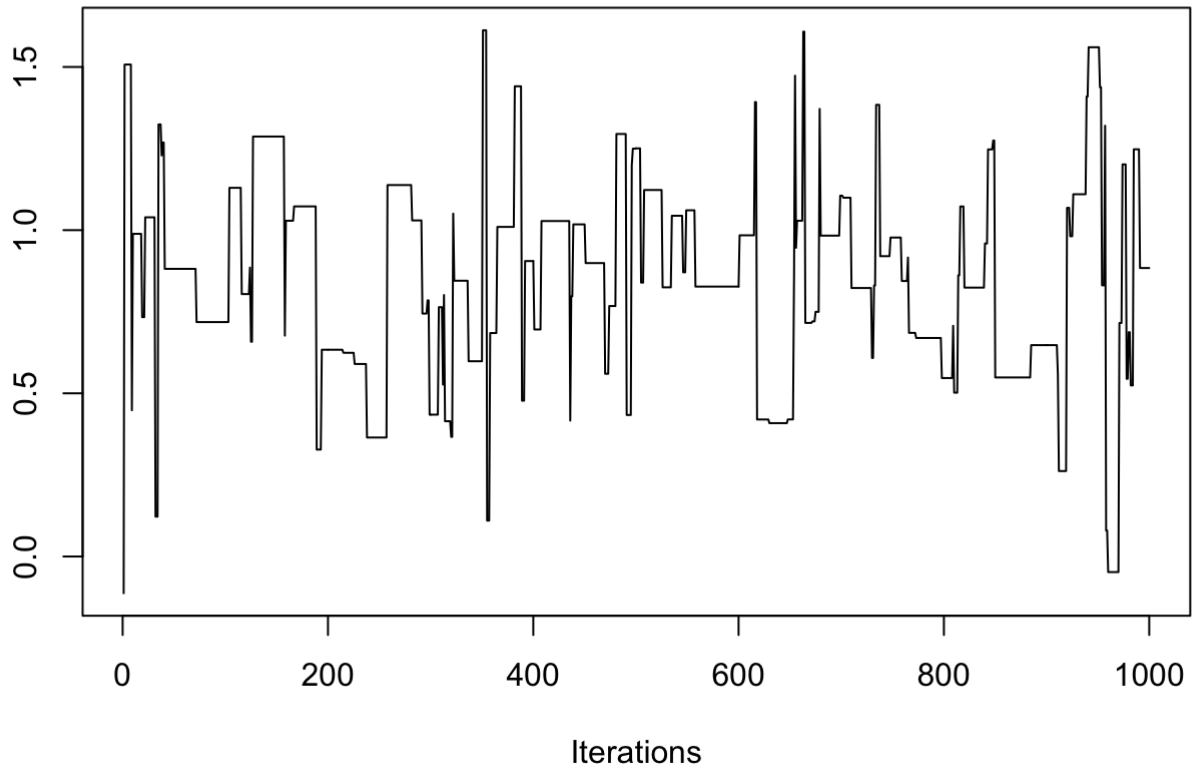


Finally, we're ready to run the sampler! Let's use $\mathrm{m} = 1000$ iterations and proposal standard deviation (which controls the proposal step size) 3.0, and initial value at the prior median 0.

```
set.seed(43) # set the random seed for reproducibility
post = mh(n=n, ybar=ybar, n_iter=1e3, mu_init=0.0, cand_sd=3.0)
str(post)
```

```
## List of 2
##  $ mu   : num [1:1000] -0.113 1.507 1.507 1.507 1.507 ...
##  $ accpt: num 0.122
```

```
library("coda")
traceplot(as.mcmc(post$mu))
```



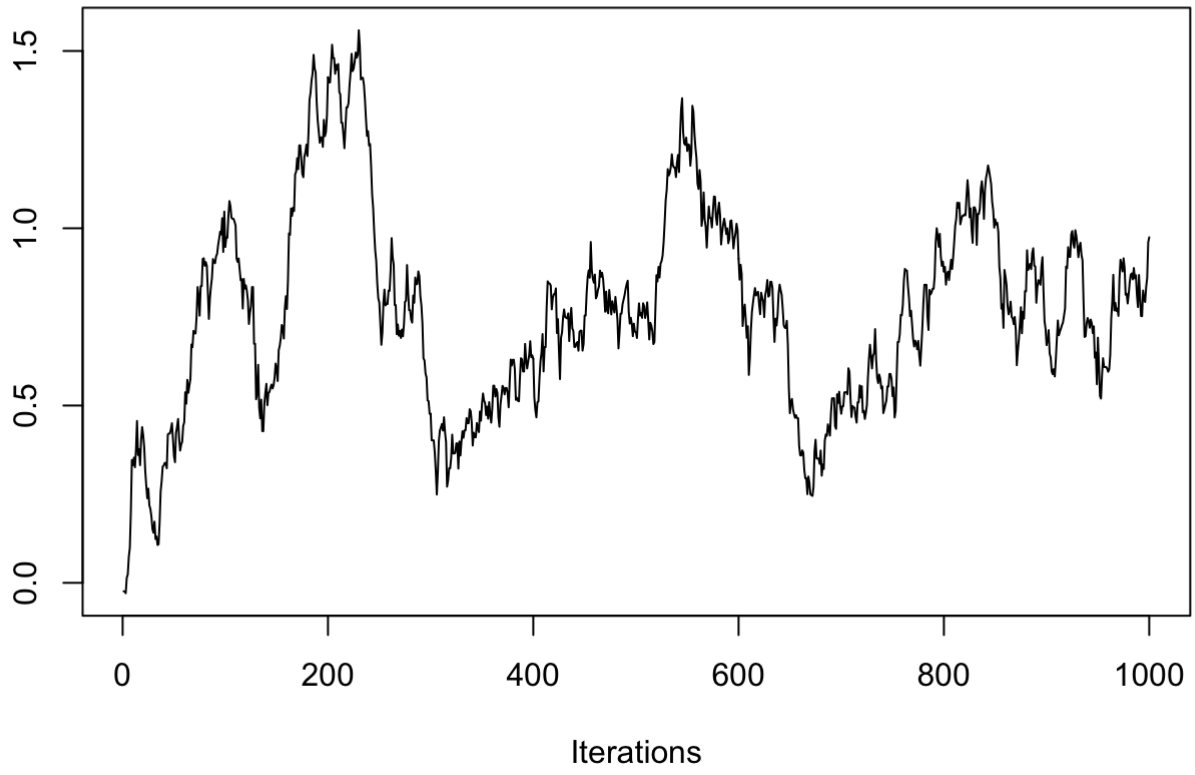
This last plot is called a trace plot. It shows the history of the chain and provides basic feedback about whether the chain has reached its stationary distribution.

It appears our proposal step size was too large (acceptance rate below 23%). Let's try another.

```
post = mh(n=n, ybar=ybar, n_iter=1e3, mu_init=0.0, cand_sd=0.05)
post$accpt
```

```
## [1] 0.946
```

```
traceplot(as.mcmc(post$mu))
```

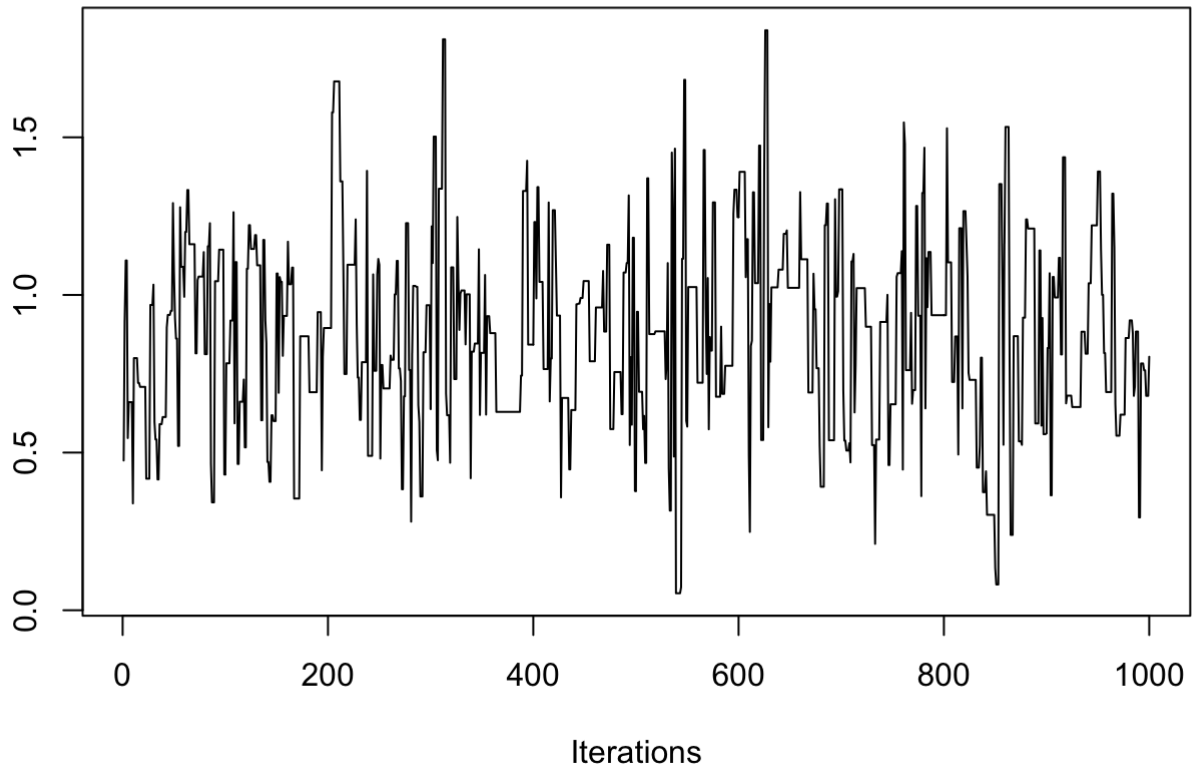


Oops, the acceptance rate is too high (above 50%). Let's try something in between.

```
post = mh(n=n, ybar=ybar, n_iter=1e3, mu_init=0.0, cand_sd=0.9)
post$accpt
```

```
## [1] 0.38
```

```
traceplot(as.mcmc(post$mu))
```

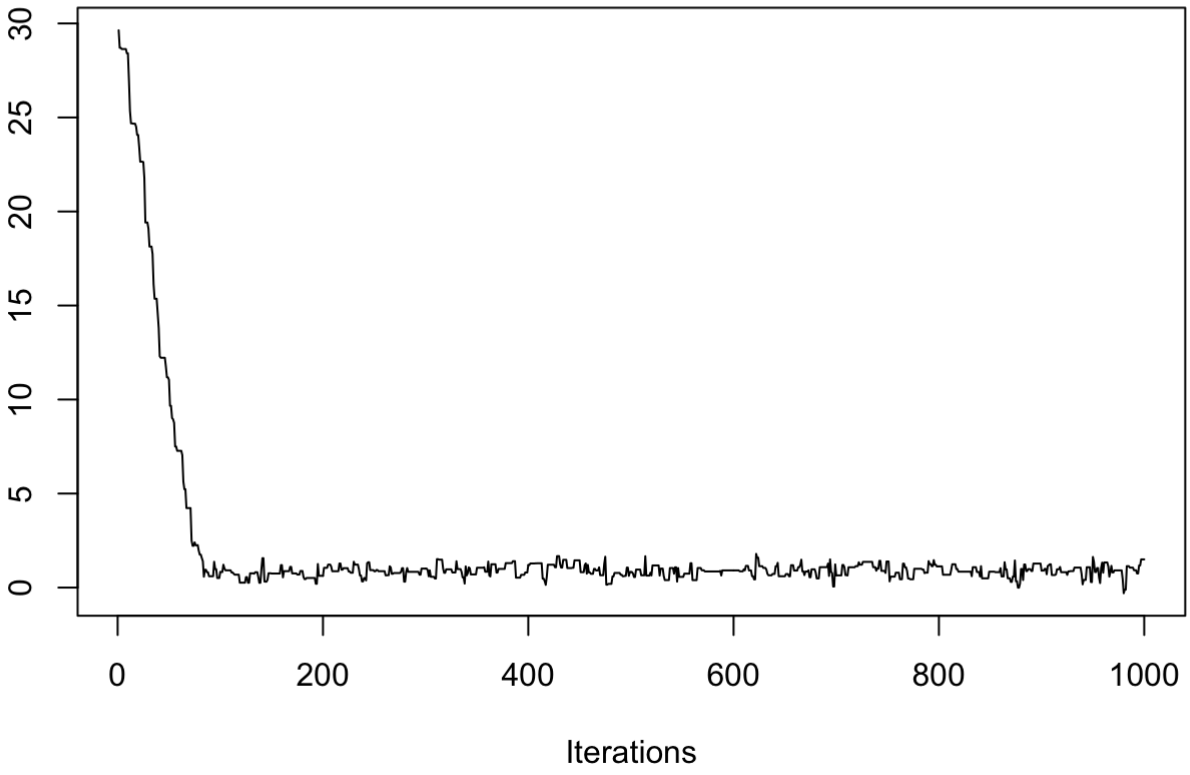


Hey, that looks pretty good. Just for fun, let's see what happens if we initialize the chain at some far-off value.

```
post = mh(n=n, ybar=ybar, n_iter=1e3, mu_init=30.0, cand_sd=0.9)
post$accpt
```

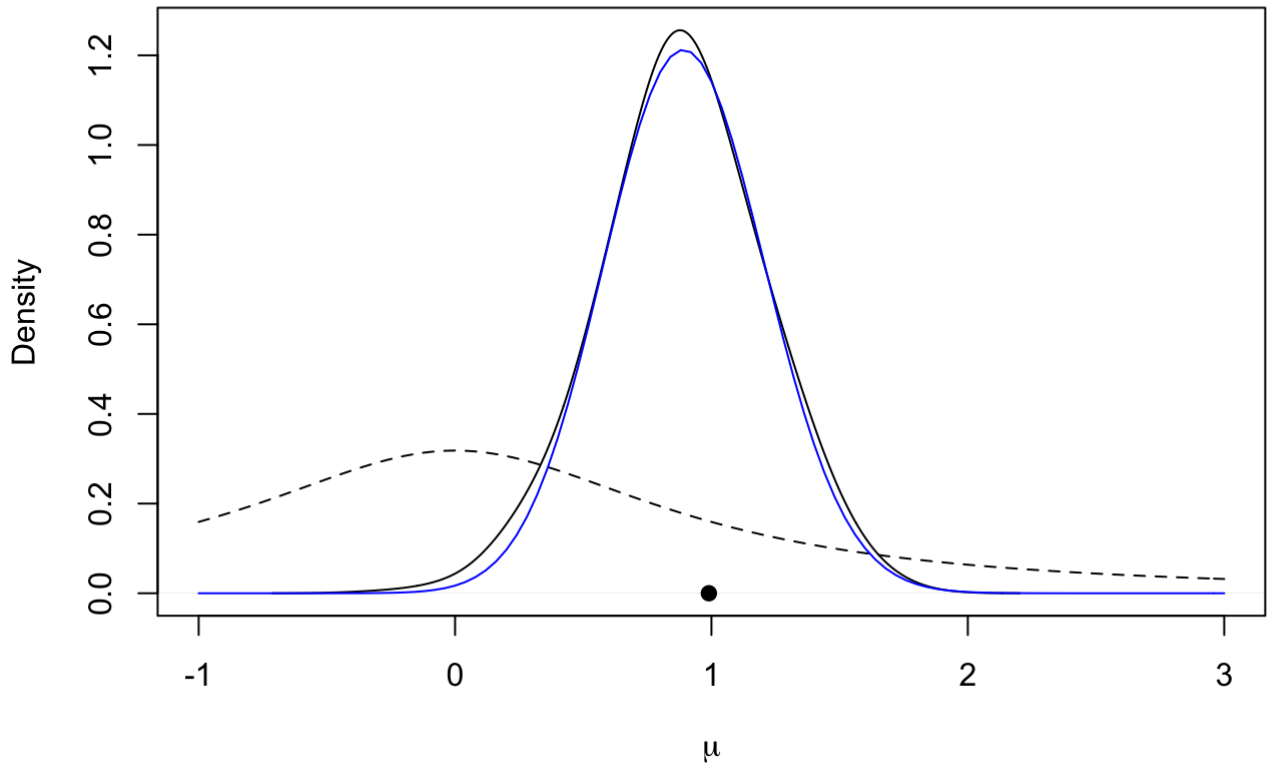
```
## [1] 0.387
```

```
traceplot(as.mcmc(post$mu))
```



It took awhile to find the stationary distribution, but it looks like we succeeded! If we discard the first 100 or so values, it appears like the rest of the samples come from the stationary distribution, our posterior distribution! Let's plot the posterior density against the prior to see how the data updated our belief about μ.

```
post$mu_keep = post$mu[-c(1:100)] # discard the first 200 samples
plot(density(post$mu_keep, adjust=2.0), main="", xlim=c(-1.0, 3.0), xlab=expression(m
u)) # plot density estimate of the posterior
curve(dt(x=x, df=1), lty=2, add=TRUE) # prior for mu
points(ybar, 0, pch=19) # sample mean

curve(0.017*exp(lg(mu=x, n=n, ybar=ybar)), from=-1.0, to=3.0, add=TRUE, col="blue") #
approximation to the true posterior in blue
```



These results are encouraging, but they are preliminary. We still need to investigate more formally whether our Markov chain has converged to the stationary distribution. We will explore this in a future lesson.

Obtaining posterior samples using the Metropolis-Hastings algorithm can be time-consuming and require some fine-tuning, as we've just seen. The good news is that we can rely on software to do most of the work for us. In the next couple of videos, we'll introduce a program that will make posterior sampling easy.