Section 8

```
library(coda)
library(MASS)
library(tidyverse)
```

Metropolis Algorithm

To generate sample s + 1 of a Metropolis MCMC sampler given (possibly) unnormalized density $p(\theta)$:

1. Propose a new sample θ_* given the old sample θ_s from a symmetric distribution
2. If $p(\theta_*) > p(\theta_s)$, then set θ_{s+1} equal to θ_* and go to the next iteration
3. If $p(\theta_*) < p(\theta_s)$, then
    a. Generate a random number r from uniform(0, 1)
    b. If $r < \frac{p(\theta_*)}{p(\theta_s)}$, set θ_{s+1} equal to θ_* and go to the next iteration
    c. Otherwise set θ_{s+1} equal to θ_s

Little wheels

Assume you are given the following (incredibly inefficient) code and told to use it to sample from a normal distribution

```
# pdf we want to sample
p = function(theta) {
  dnorm(theta, 1.0, 2.0)
}
metropolis = function(theta_s) {
  # Function should return the next state
  #   in the Markov chain given the current state, theta_s!

  theta_p = rnorm(1, theta_s, 1.0)

  if(p(theta_p) > p(theta_s)) {
    return(theta_p)
  } else {
    r = runif(1)
    if(r < p(theta_p) / p(theta_s)) {
      return(theta_p)
    } else {
      return(theta_s)
    }
  }
}
N = 10000
samples = rep(0, N)
samples[1] = 10.0
for(i in 1:(length(samples) - 1)) {
```

```
  Samples[i + 1] = metropolis(samples[i])
}
```

We can look at our traceplots and effective sample size estimates with the coda package:

```
plot(as.mcmc(samples))
effectiveSize(samples)
```

Medium wheels

Using the un-logged densities is numerically unstable. As an example of what can happen, compare the outputs of:

```
print(1.0e-100 * 1e-100, format = "e", digits = 20)
```

```
## [1] 9.99999999999999821e-201
```

```
print(1.0e-200 * 1e-200, format = "e", digits = 20)
```

```
## [1] 0
```

It is really common to need to evaluate numbers this small in a probabilistic model. For instance, a term like 0.36^{300} might come up when evaluating a binomial pmf that models a basketball player's yearly shooting percentage. If we extend that to maybe three years worth of shots-made, (0.36^{900}), we'll see that evaluates to zero. The trick to avoid this is working on the log scale.

We want our metropolis algorithm to work on a log scale too. Because log is a monotonic increasing function, we can just take the log of the conditions in steps 2 and 3 above and get our new algorithm:

2. If $\log p(\theta_*) > \log p(\theta_s)$, then set θ_{s+1} equal to θ_* and go to the next iteration
3. If $\log p(\theta_*) < \log p(\theta_s)$, then
    a. Generate a random number r from uniform(0, 1)
    b. If $\log(r) < \log p(\theta_*) - \log p(\theta_s)$, set θ_{s+1} equal to θ_* and go to the next iteration
    c. Otherwise set θ_{s+1} equal to θ_s

Rewrite the code above to work on the log scale and convince yourself it is working.

```
logp = function(theta) {
  dnorm(theta, 1.0, 2.0, log = TRUE)
}
metropolis = function(theta_s) {
  # Use logp, *not* log(p(...))
  theta_p = rnorm(1, theta_s, 1.0)

  if(logp(theta_p) > logp(theta_s)) {
    return(theta_p)
  } else {
    r = runif(1)
    if(log(r) < logp(theta_p) - logp(theta_s)) {
      return(theta_p)
    } else {
      return(theta_s)
    }
  }
}
N = 10000
samples = rep(0, N)
```

```r
samples[1] = 100.0
for(i in 1:(length(samples) - 1)) {
  samples[i + 1] = metropolis(samples[i])
}
```

Big wheels

Because you are enterprising young statisticians, you want to sample a multidimensional distribution. You can use the funciton mvrnorm to sample from a multivariate proposal distribution like so:

```r
library(MASS)
mvrnorm(1, c(1.0, 2.0), matrix(c(1.0, 0.5, 0.5, 2.0), nrow = 2))
```

```
## [1] 1.817168 2.727315
```

This is a sample from:

$$N\left(\begin{bmatrix}1\\2\end{bmatrix}, \begin{bmatrix}1 & 0.5\\0.5 & 2\end{bmatrix}\right)$$

Because you are enterprising young software engineers, you want to write a function that does the Metropolis sampling for you, handles the burnin, and makes it easy to try different proposal covariances.

Because this is PSTAT115, you get to do that now:

```r
logp = function(theta) {
  sum(dnorm(theta, c(1.0, 2.0), c(2.0, 3.0), log = TRUE))
}
metropolis = function(theta_s, cov) {
  # Note: theta_s is a vector now!
  theta_p = mvrnorm(1, theta_s, cov)

  if(logp(theta_p) > logp(theta_s)) {
    return(theta_p)
  } else {
    r = runif(1)
    if(log(r) < logp(theta_p) - logp(theta_s)) {
      return(theta_p)
    } else {
      return(theta_s)
    }
  }
}
# theta_0 is your initial parameter guess
# burnin is number of burnin samples
# maxit are the number of samples you generate (post burnin)
# cov is the proposal covariance matrix
rw_metrop_multi = function(theta_0, burnin, maxit, cov) {
  samples = matrix(0, ncol = length(theta_0), nrow = burnin + maxit)

  samples[1,] = theta_0
  for(i in 1:(nrow(samples) - 1)) {
    samples[i + 1,] = metropolis(samples[i,], cov)
  }
  samples[burnin:(burnin + maxit),]
```

```
}
samples = rw_metrop_multi(c(10.0, 10.0), 1000, 10000, matrix(c(1.0, 0.0, 0.0, 2.0), nrow = 2))
```

Check that the sampler is producing results you trust, because we are about to work on a non-trivial problem!

Model and Simulated data

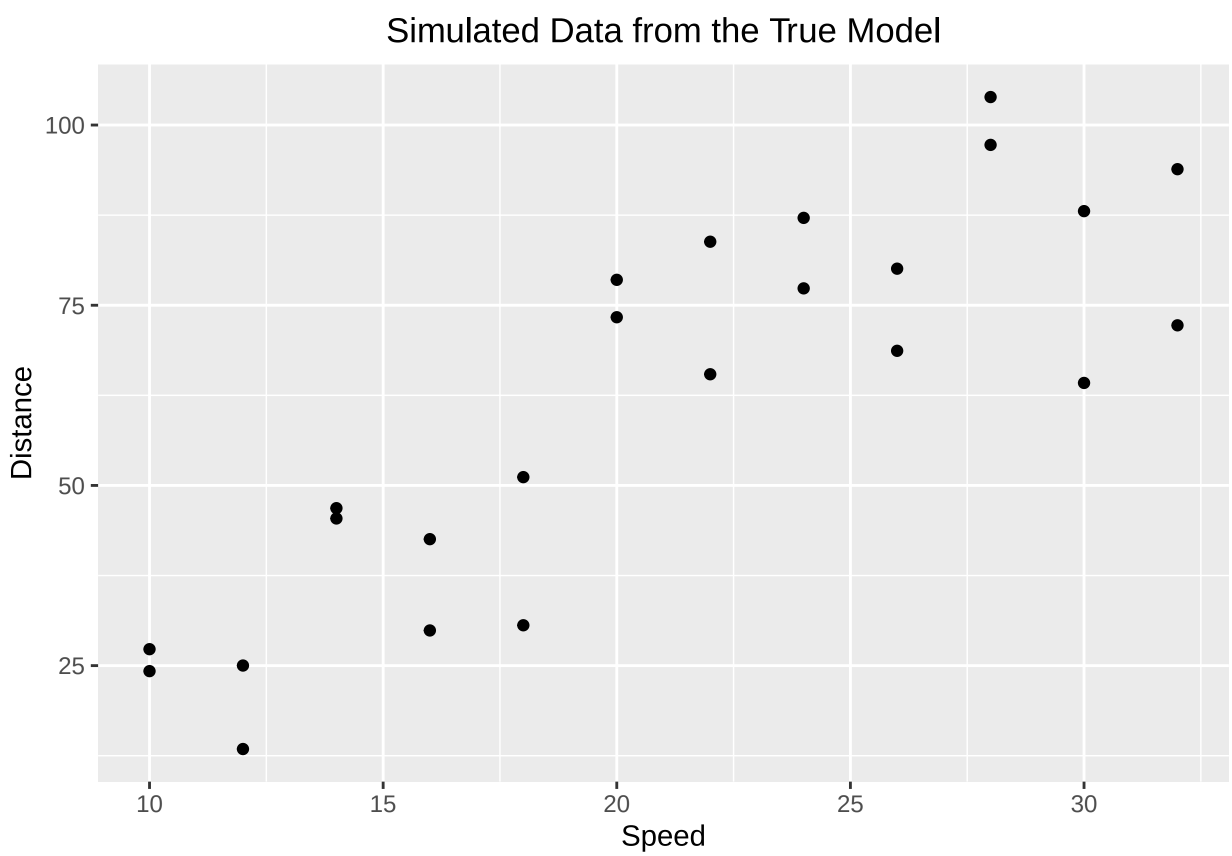
Let us consider a model of the speed of cars and the distances taken to stop. Assuming the true model is

$$dist = -8 + 3.5 * speed + \epsilon, \text{ where } \epsilon \sim N(0, 15^2)$$

And now we generate simulated data from the true model and try to estimate the parameters using the Bayesian approach.

```
true_beta0 <- -8
true_beta1 <- 3.5
x          <- rep(seq(10, 32, by = 2), each = 2)
set.seed(10)
y          <- true_beta0 + true_beta1 * x + rnorm(length(x), 0, 15)

data.frame(x = x, y = y) %>% ggplot() + geom_point(aes(x = x, y = y)) +
  xlab("Speed") + ylab("Distance") + ggtitle("Simulated Data from the True Model") +
  theme(plot.title = element_text(hjust = 0.5))
```



Now we try to propose a simple linear model and estimate the parameters using Metroplis-Hastings algorithm.

Our model is

$$dist = \beta_0 + \beta_1 * speed + \epsilon, \text{ where } \epsilon \sim N(0, 15^2).$$

Notice that here we assume we know the variance for the error term, so we can focus on the estimation of β_0 and β_1.

Function for log_posterior

From the lecture materials we know it is often more stable working with log-scale. So here we write a function for the log_posterior. You only need to consider the log for the essential parts in the posterior distribution.

```
logp <- function(beta){
  return(sum(dnorm(y - beta[1] - beta[2] * x, mean = 0, sd = 15, log = TRUE)))
}
```

Call our Metropolis sampler

```
samples <- rw_metrop_multi(c(0, 0), 1000, 20000, cov = matrix(c(30.0, 0.0, 0.0, 1.0), nrow=2))
```

```
plot(samples[,1], samples[,2])
```

Covariance Matrix for the Proposal Distribution

We can actually play around with the covariance matrix in the proposal distribution. Negative correlation between β_0 and β_1 tend to provide better sampling results.