Chapter 6: Markov Chains

Ben Ridenhour

18-Oct-23

Contents

Markov Chains

The only real novel part of Chapter 6 is the introduction of Markov chains. Markov chains are said to be "memoryless" because the next state of the chain strictly depends on the current state of the chain, i.e. $X_t \rightarrow X_{t+1}$. Markov chains are used in a wide variety of applications. The simplest of those would be in an iterative. If the model is linear, we can write this down in matrix notation as

$$\mathbb{x}_t = \mathbb{B}\mathbb{x}_{t-1}$$

where $\mathbb{B}$ is an $n \times n$ the transition matrix which defines the transition probabilities among n states. Clearly, we solve the for the equilibrium $\mathbb{x}^* = \mathbb{x}_t = \mathbb{x}_{t-1}$ by solving $\mathbb{x}^* = \mathbb{B}\mathbb{x}^*$ which gives $(\mathbb{B} - \mathbb{I})\mathbb{x}^* = 0$. This is the same solving for an eigenvalue $\lambda = 1$. What does it mean if there is no $\lambda = 1$? Should there ever be a case where there isn't an eigenvalue of 1? Let's do this for the example in the book of politics (pg. 227):

```
#transition matrix
B <- matrix(c(0.75, 0.05, 0.20, 0.20, 0.6, 0.2, 0.4, 0.2, 0.4), ncol = 3)
eigensys <- eigen(B)
eigensys$values #first element is lamda = 1
```

```
## [1] 1.00 0.55 0.20
```

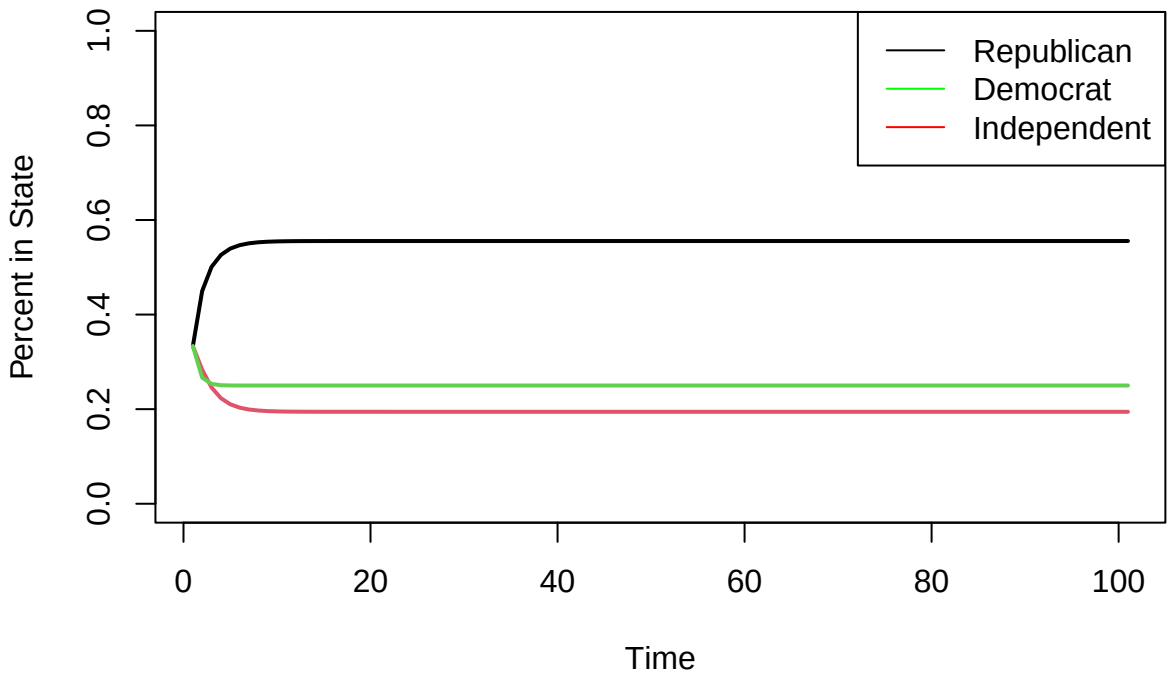
```
v1 <- eigensys$vectors[,1]
v1 / sum(v1) #make a probability, and ta-da!
```

```
## [1] 0.5555556 0.1944444 0.2500000
```

We can also iterate the equation and get to the equilibrium that way:

```
X <- data.frame("Republicans"=1/3,"Democrats" = 1/3, "Independents" = 1/3)  #equal proportions

for(i in 1:100){
  Xp <- as.vector(B%*%t(X[i,]))
  #names(Xp) <- names(X)
  X <- rbind(X, Xp)
```

```
}

matplot(X, type = "l", pch = 1:3, ylim = c(0,1), xlab="Time", ylab= "Percent in State", lty=1, lwd=2)
legend("topright", c("Republican","Democrat","Independent"), col=c("black","green","red"), lty=1)
```



More interestingly, Markov chains can be used to simulate stochastic behavior. In this scenario, we use *probability* to simulate what occurs in the system. How do you think we would do this? Try to do this for the above system and plot the dynamics!!!

Sometimes, there may be a very low probability mass of particular events occurring. In these cases we may want to use a method tau leaping developed for use in the Gillespie algorithm. See here for details on this method and the efficient step size algorithm for tau leaping.

Finally, the Metropolis-Hastings algorithm is often combined with Markov chain-Monte Carlo (MCMC) simulation to help search parameter space. The Metropolis-Hastings algorithm is essentially a way to propose x_{t+1} and decide if the chain should move to that new state, or simply stay put. When a MCMC simulation converges we see *stationarity*, or essence we observe white noise around a fixed point (which is hopefully the appropriate estimate of a parameter.) See here for details on the Metropolis-Hastings algorithm.

Using the Gillespie Algorithm

We briefly discussed the Gillespie algorithm for simulating stochastic dynamic systems using Markov chains. Essentially, we determine the expected number of "reactions" to occur in some time step and then we pick one of those reactions depending its "probability mass." Returning to the political example we used before. Recall that the following transitions were declared:

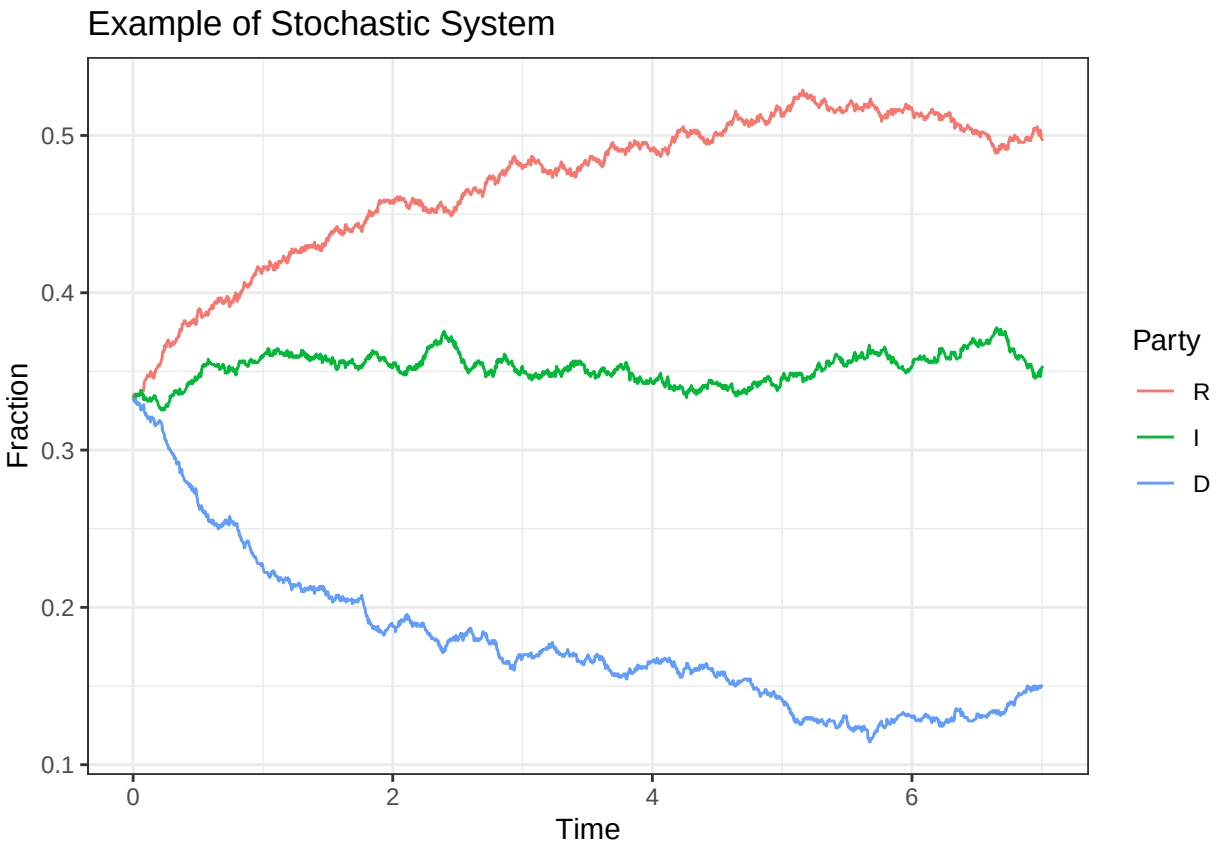
$$R \xrightarrow{0.2} I$$
$$R \xrightarrow{0.05} D$$
$$I \xrightarrow{0.2} R$$
$$I \xrightarrow{0.2} D$$
$$D \xrightarrow{0.4} R$$
$$D \xrightarrow{0.2} I$$

Thus the probability mass at any time t would be, e.g., $0.2R(t)$ for a republican becoming a democrat. Using the probabily mass for 6 reactions, we can use the algorithm to look at the stochastic behavior of the system. We do this in this manner:

```
state <- c("R"=300,"I"=300, "D"=300) #initial state
system <- list("state" = state, "Time" = 0)

probMass <- function(state) c(0.2*state[1], 0.05*state[1], 0.2*state[2], 0.2*state[2], 0.4*state[3], 0.2

update <- function(theta){
  with(theta,{
  p <- unlist(probMass(state))
  P <- cumsum(c(0,p/sum(p))) #convert to CDF by normalizing
  rand <- runif(1)
  rxn <- max(which(P < rand))
  rxnVec <- rep(0,6)
  rxnVec[rxn] <- 1
  transition <- matrix(c(-1,1,0,-1,0,1,1,-1,0,0,-1,1,1,0,-1,0,1,-1), nrow = 3)
  #as.vector in the next command is not necessary if you don't care about keeping a labelled vector
  list(state + as.vector(transition %*% rxnVec), "Time" = Time  - log(runif(1))/sum(p))
  })
}

Pol <- data.frame(t(unlist(system)))
clock <- system.time(
  while(max(Pol$Time) <= 7){
    i <- nrow(Pol)
    out <- update(list(state = Pol[i,1:3], Time = Pol$Time[i]))
    Pol <- rbind(Pol, data.frame(t(unlist(out))))
  }
)

colnames(Pol)[1:3] <- c("R","I","D")

library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.1.2
```

```
library(reshape2)

plotData <- melt(Pol, id = "Time", variable.name = "Party", value.name = "Count")
plotData$Fraction <- plotData$Count/900
ggplot(plotData, aes(x = Time, y = Fraction, group = Party, color = Party)) + geom_line() + theme_bw() +
```



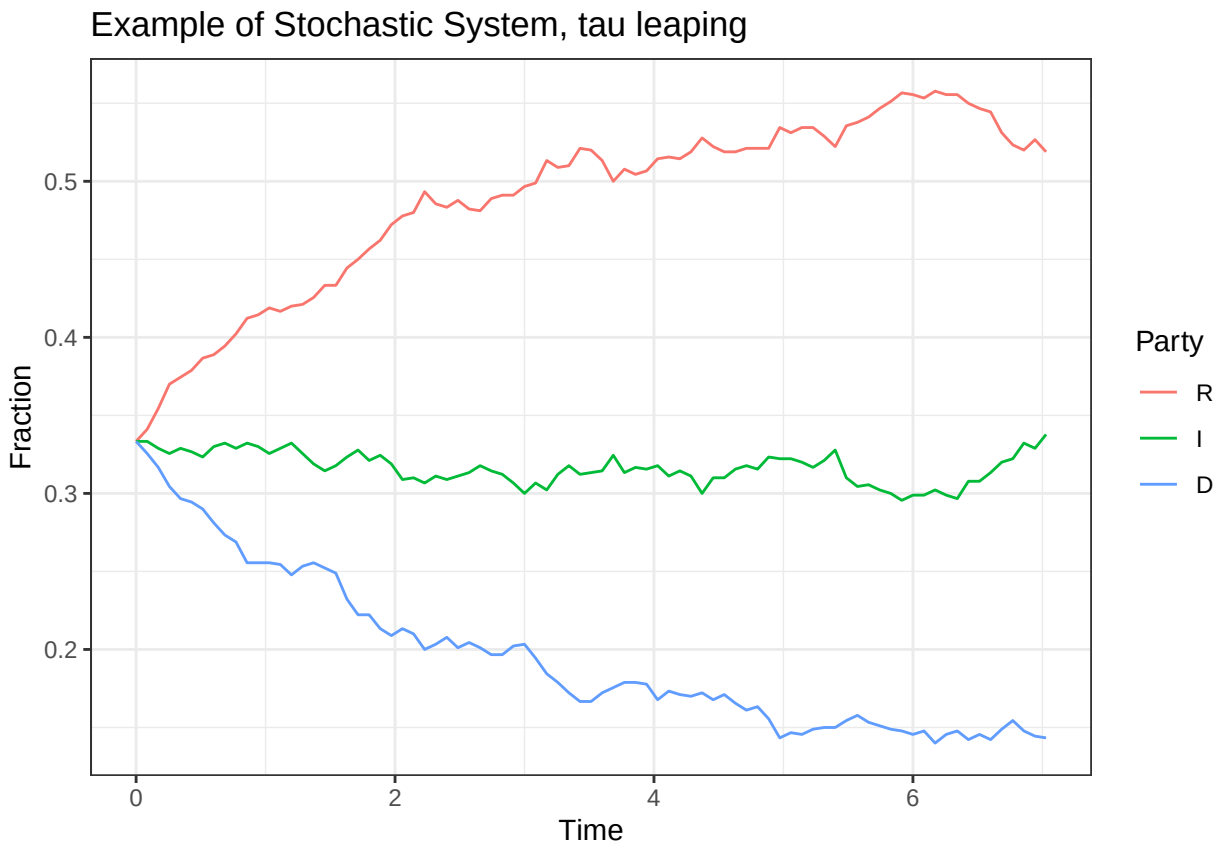
Note that the system has very similar time dynamics compared to Figure 6.5 in your book.

Tau-leaping algorithm

The Gillespie algorithm however took 2361 steps in R to simulate the system, which isn't very computationally efficient (it took 3.633 seconds in R). This is particularly true when there are relatively minor changes in the system after around $t = 3$ (i.e., the steady state has been reached). To take advantage of this, we can use the "tau leaping" algorithm. Let's try this algorithm:

```
transProb <- function(state, variance = F) colSums(t(matrix(c(-1,1,0,-1,0,1,1,-1,0,0,-1,1,1,0,-1,0,1,-1

#g is the vector of the highest order of a reaction in the system
#all of our are linear so g = rep(1, 3)
update.leaping <- function(theta, pVec, epsilon = 0.03, g = rep(1,3)){
  with(theta,{
    mu <- transProb(state)
    var <- transProb(state, T)
    L <- sapply(epsilon*state/g, max, 1)
    tau <- min(L/abs(mu), L^2/var)
    ### Now advance the chain
    rxnVec <- rpois(6, unlist(probMass(state))*tau)
    transition <- matrix(c(-1,1,0,-1,0,1,1,-1,0,0,-1,1,1,0,-1,0,1,-1), nrow = 3)
    out <- state + as.vector(transition %*% rxnVec)
    #if we get too large of a step and produce a negative value
    #cut tau in half and try again
    while(min(out) < 0){
      tau <- tau/2
```

```
      rxnVec <- rpois(6, unlist(probMass(state, ...))*tau)
      out <- state + as.vector(transition %*% rxnVec)
    }
    list(state + as.vector(transition %*% rxnVec), "Time" = Time  + tau)
  })
}

Pol2 <- data.frame(t(unlist(system)))
clock2 = system.time(
  while(max(Pol2$Time) <= 7){
    i <- nrow(Pol2)
    out <- update.leaping(list(state = Pol2[i,1:3], Time = Pol2$Time[i]), c(0.2,0.05,0.2,0.2,0.4,0.2))
    Pol2 <- rbind(Pol2, data.frame(t(unlist(out))))
  }
)

colnames(Pol2)[1:3] <- c("R","I","D")

plotData <- melt(Pol2, id = "Time", variable.name = "Party", value.name = "Count")
plotData$Fraction <- plotData$Count/900
ggplot(plotData, aes(x = Time, y = Fraction, group = Party, color = Party)) + geom_line() + theme_bw() +
```



The tau-leaping algorithm only took 83 steps in R to simulate the system, which is *much* more computationally efficient (it only took 0.325 seconds).

NB: I have placed pdfs of the original Gillespie and tau-leaping articles in the Chapter 6 folder.

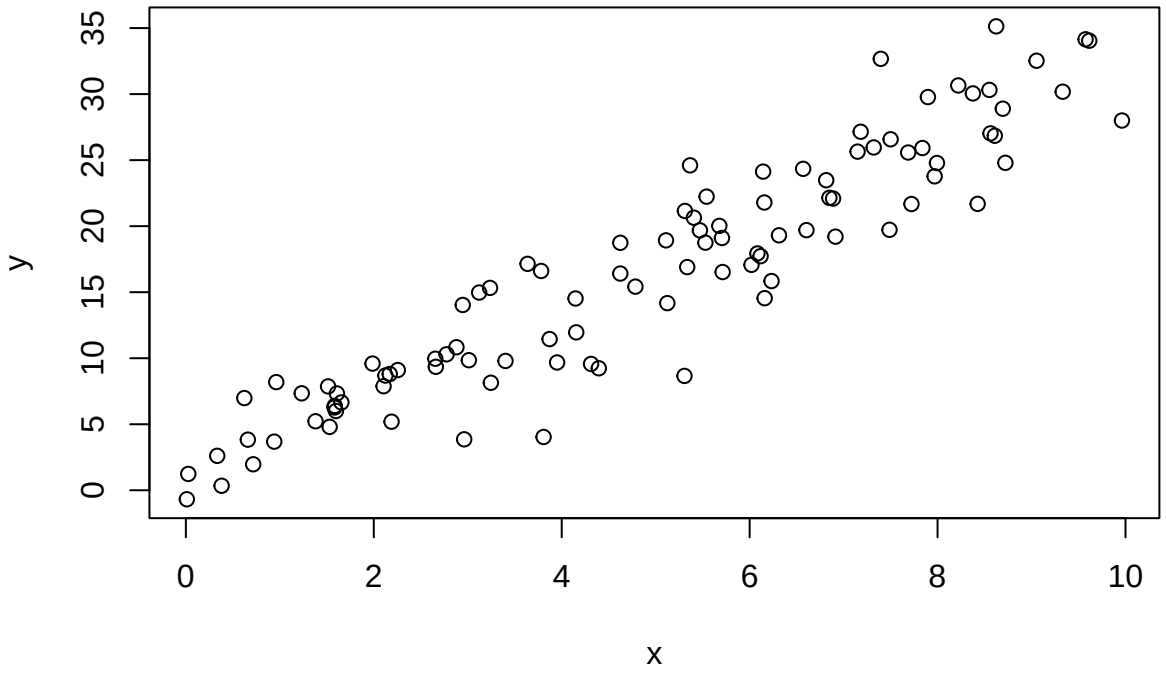
Using the Metropolis-Hastings Algorithm

The last thing we will investigate with Markov chains is their use in searching parameter space. To search parameter space, we initialize the chain to initial state Θ_0. We then will propose a new state based on a **proposal density**. Commonly, we might propose new values of the chain based on a multivariate normal distribution, however, other proposals kernels might be necessary depending on the problem. Furthermore we might need to constrain the proposal space if we know there are particular constraints on the parameters (e.g., > 0). Once we propose a new state, we need to use an **acceptance algorithm**. A very common of these is the Metropolis-Hastings algorithm. The algorithm to update the chain works as follows:

1. The proposal density should be symmetric around the current state of the chain Θ.
2. Generate a new proposal θ.
3. Calculate the probability of the proposal relative to the current state, $\alpha = P(\theta)/P(\Theta)$.
4. Generate a random uniform u. If $u > \alpha$, reject the proposal. Append the "winning" state to the chain.
5. Iterate 2–4 until "stationarity" is achieved and sufficient samples are present to examine statistically.

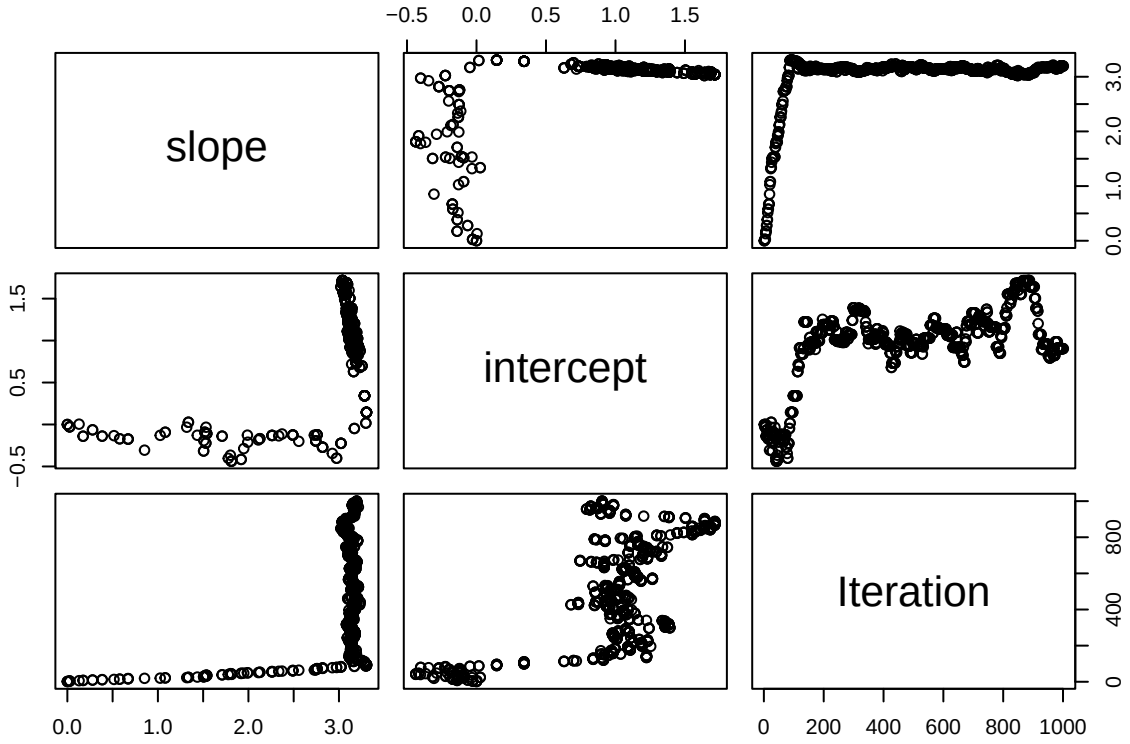
To see how this works, let's use the method on a simple regression problem.

```
x <- runif(100,0,10)
y <- rnorm(100, 3*x + 2, 3)

plot(x,y)
```

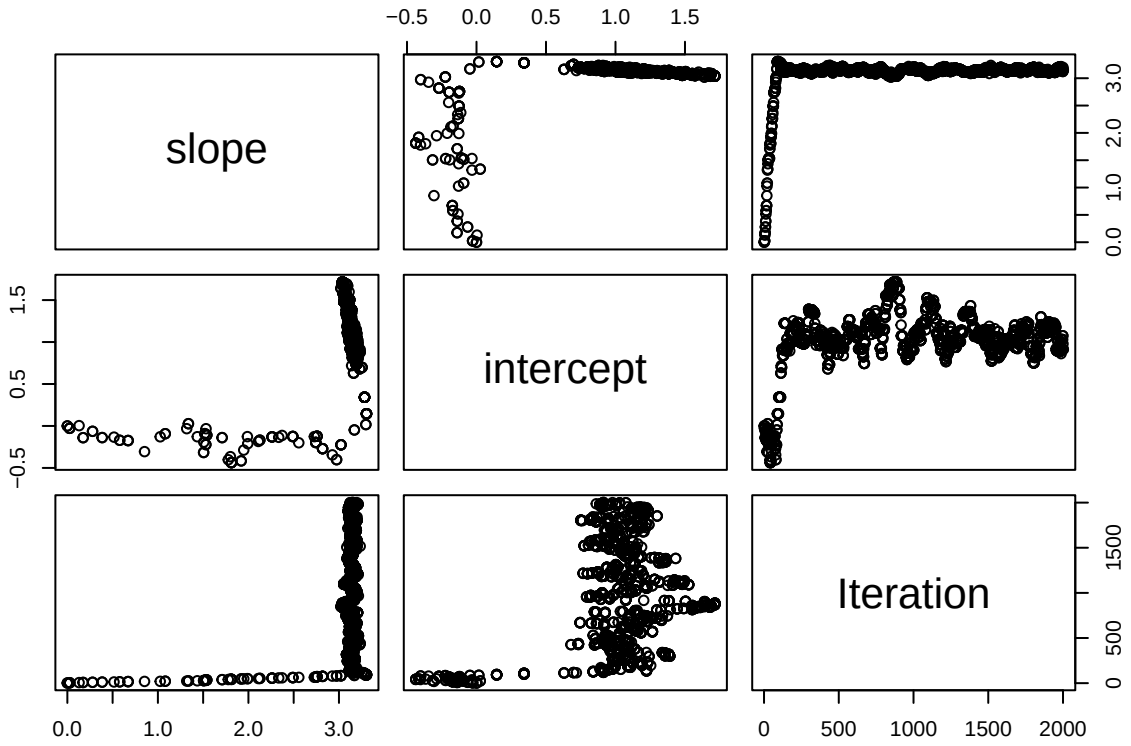



```
chain <- c("slope"=0,"intercept"=0) #initialize chain

Chain <- data.frame(t(chain))

proposal <- function(theta, sd = 0.1) rnorm(length(theta),theta,sd)
ll_theta <- function(theta, x, y) sum(dnorm(y,x*theta[1]+theta[2],log=T))

nSteps <- 1000
for(i in 1:nSteps){
  old <- unlist(Chain[i,])
  new <- proposal(old)
```

```
  log_alpha <- ll_theta(new,x,y) - ll_theta(old,x,y)
  next_prop <- if(log(runif(1)) > log_alpha) old else new
  Chain <- rbind(Chain,next_prop)
}

Chain$Iteration <- seq_len(nrow(Chain))
###plot the chain values
plot(Chain)
```

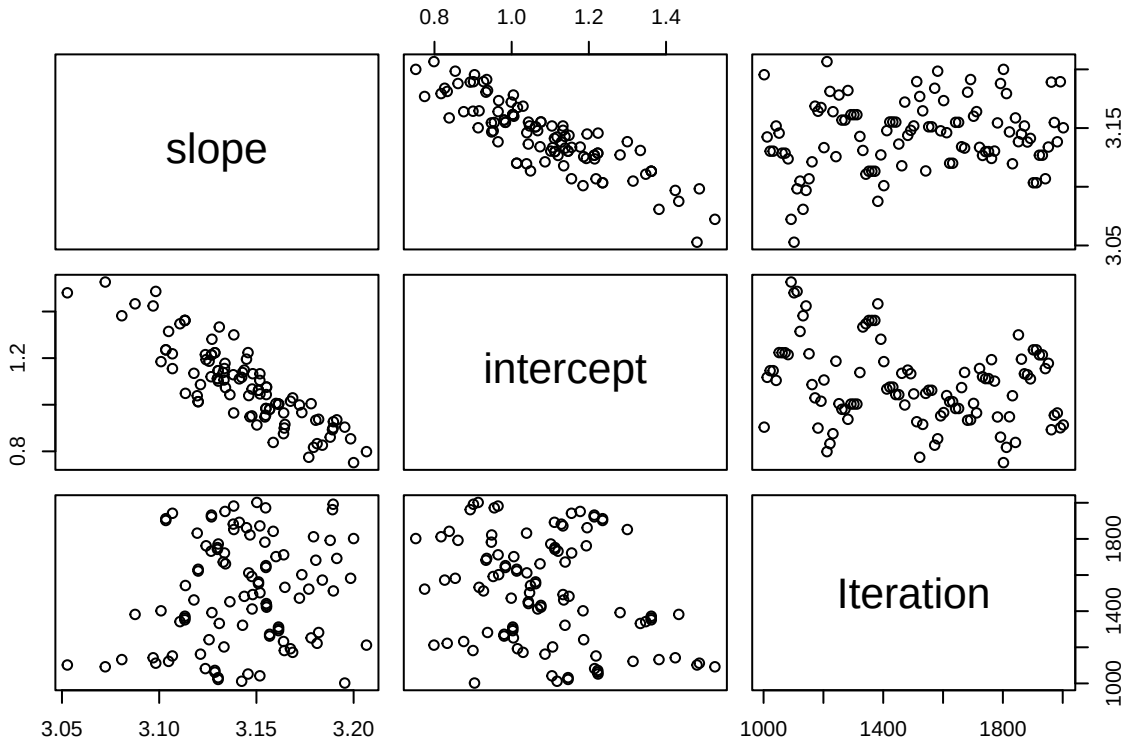



```
###still see significant autocorrelation for the intercept, iterate more
Chain$Iteration <- NULL

for(i in nrow(Chain):(nrow(Chain) + 1000)){
  old <- unlist(Chain[i,])
  new <- proposal(old)
  log_alpha <- ll_theta(new,x,y) - ll_theta(old,x,y)
  next_prop <- if(log(runif(1)) > log_alpha) old else new
  Chain <- rbind(Chain,next_prop)
}

Chain$Iteration <- seq_len(nrow(Chain))

plot(Chain)
```



Okay, the chain looks pretty stationary at this point. Now we would typcally throw away early samples where we are worried about autocorrelation/non-stationarity, this is called the "burn-in." We can easily drop the first 1000 samples as burn-in. To further reduce autocorrelation, we can pick samples every n iterations, this is called thinning. After discarding the the burn-in and thinning, we can look at the distributions of the parameters.

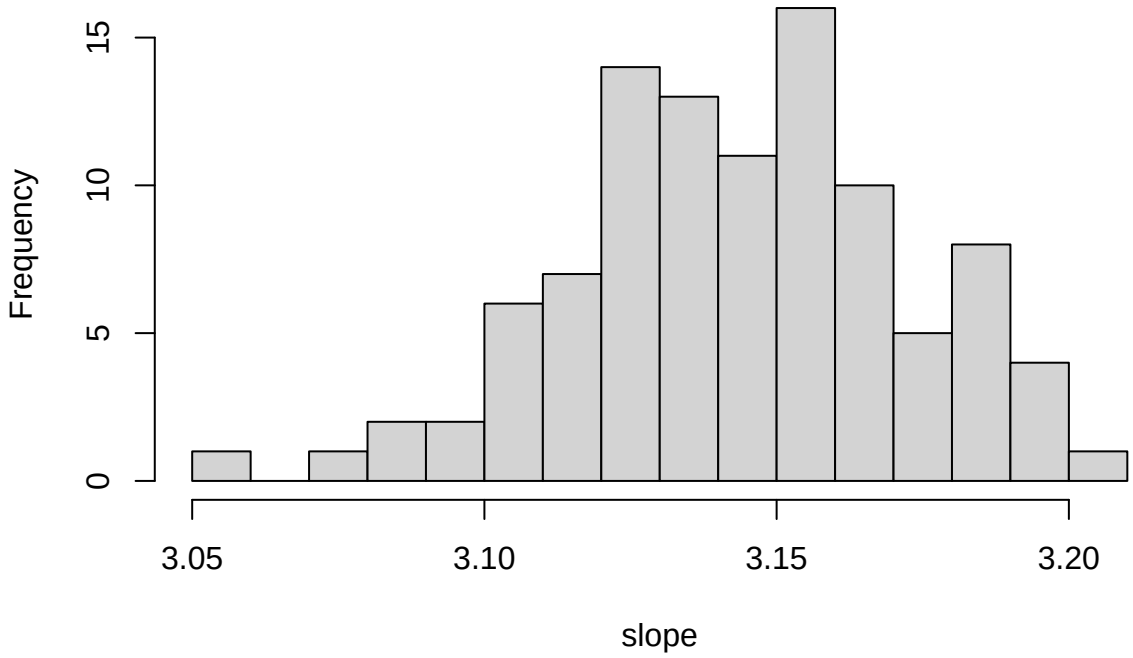
```
FinalChain <- Chain[seq(1002,2002,by=10),] #take every 10th sample in the last 1000 iterations

plot(FinalChain)
```



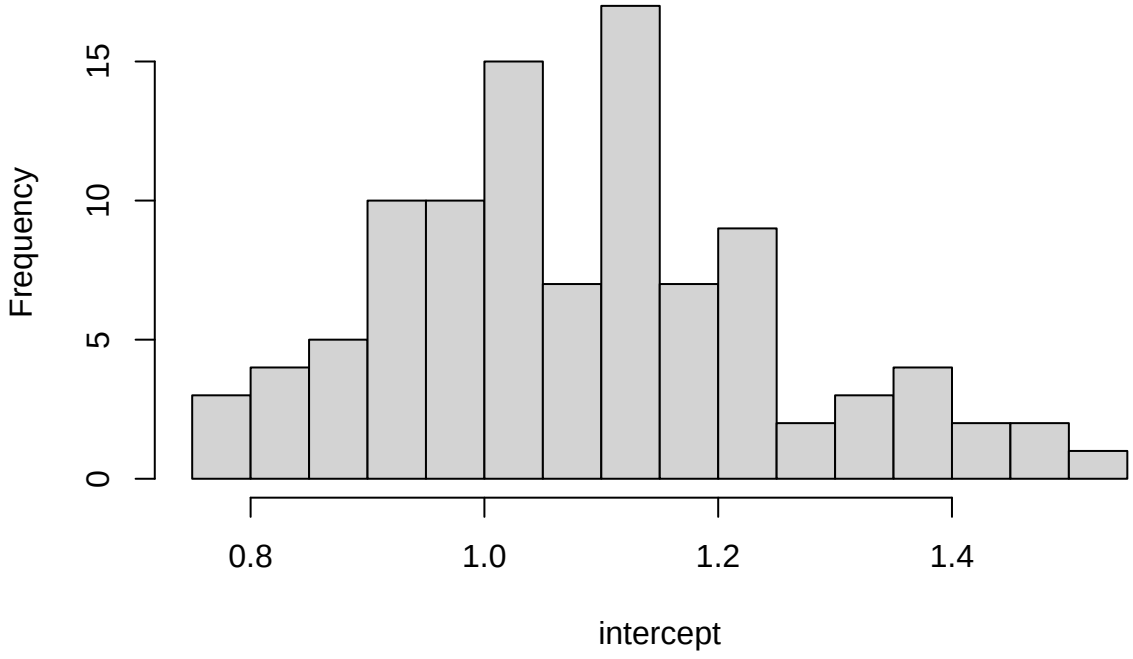

```
summary(FinalChain)
```

```
##      slope          intercept        Iteration
##  Min.   :3.053   Min.   :0.7515   Min.   :1002
##  1st Qu.:3.127   1st Qu.:0.9649   1st Qu.:1252
##  Median :3.145   Median :1.0735   Median :1502
##  Mean   :3.143   Mean   :1.0856   Mean   :1502
##  3rd Qu.:3.161   3rd Qu.:1.1866   3rd Qu.:1752
##  Max.   :3.207   Max.   :1.5266   Max.   :2002
```

```
hist(FinalChain$slope,20, xlab="slope", main = NULL)
```




```
hist(FinalChain$intercept,20, xlab="intercept", main = NULL)
```




```
quantile(FinalChain$slope,c(0.025,0.975)) #95% CI of slope
```

```
##     2.5%    97.5%
## 3.084188 3.196944
```

```
quantile(FinalChain$intercept,c(0.025,0.975)) #95% CI of intercept
```

```
##      2.5%     97.5%
## 0.8078594 1.4566185
```

```
summary(lm(y ~ x))
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -9.1223 -1.9667  0.2877  2.2678  8.3206
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1.1009     0.6678   1.649    0.102
## x             3.1436     0.1195  26.315   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.212 on 98 degrees of freedom
## Multiple R-squared:  0.876,  Adjusted R-squared:  0.8748
## F-statistic: 692.5 on 1 and 98 DF,  p-value: < 2.2e-16
```