Lab 2 - Matrix Algebra and Linear Regression in R

Lab Goals

This lab is intended to

1. Review matrix operations in R. In particular, we will examine

a. functions to define matrices in R

b. matrix operations and in particular the difference between element-wise and matrix multiplication.

c. matrix transposes and inverses

d. eigen analysis of a matrix

2. Review linear regression in R, and reconstructing the elements of `lm()` from scratch.
3. Conduct a brief simulation to verify some theoretical results.

Matrix Algebra in R

Defining vectors

We start by constructing a vector (a 1-dimensional array) by simply imputting a set of numbers using the *concatination* function `c()`:

```
x=c(1,2,3)
x
```

```
## [1] 1 2 3
```

This is not always very conventient, so we can also some shortcuts. We can get the numbers from 1 to 3 by

```
x = 1:3
x
```

```
## [1] 1 2 3
```

This is good for runs of integers, but we might also want other sequences, for this we can use

```
x = seq(from=1,to=3,by=1)
x
```

```
## [1] 1 2 3
```

Important Note: The function `seq()` has *named* arguments `from` (the starting point of the sequence), `to` (the ending point) and `by` (the size of the steps to take). These are read in this order by default so that `seq(1,3,1)` gives the same answer as above. However, you can also change the order, `seq(by=1,from=1,to=3)` also gives the same output, but `seq(3,1,1)` produces an error (try it!). While R programmers (the course staff included) often skip the argument names, *it is good coding practice to always put them in, so you know what R is treating as which argument*; we will try to stick to this in BTRY/STCI 4030.

Question: 1. How would you use this to produce a vector (1,1.5,2,2.5,3,3.5,4)? What about (4,3,2,1)?

2. What if I wanted to produce 15 numbers between 1 and 4 without calculating the step size? (Hint, look at the help function for other arguments.

We might also want to look at repeated numbers, for this try

```
rep(x=1,times=3)
```

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

Notice that you can also repeat a vector:

```
rep(x=x,times=22)
```

```
##  [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2
## [36] 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

Question: what if we wanted to repeat each element of `x` twice to get (1,1,2,2,3,3)?

Defining matrices (2 dimensional arrays)

The `matrix()` command is the starting place here.

```
A=matrix(data=1:6,nrow=2,ncol=3)
```

Notice that this fills the 2-by-3 matrix going down rows, we might want to go the other way

```
B=matrix(data=1:6,byrow=T,ncol=3)
```

Here R has worked out that it needs 2 rows if `data` is of length 6 and there are 3 columns. *It is nonetheless good practice to specify all dimensions, so that you get a warning if something isn't how you expect it to be.*

We can also combine vectors by row or by column:

```
C=cbind(1:3,4:6)                        # column bind
D=rbind(1:2,3:4,5:6)                    # row bind
```

This also works if we want to stack matrices – try row binding `A` and `B`.

Question: What happens if you try to row bind `A` and `C`?

Matrix operations

Addition and Subtraction

Addition and subtraction are defined element-wise for matrices. We just add up the matching elements

```
A+B
```

```
##      [,1] [,2] [,3]
## [1,]    2    5    8
## [2,]    6    9   12
```

```
A-B
```

```
##      [,1] [,2] [,3]
## [1,]    0    1    2
## [2,]   -2   -1    0
```

But note that both matrices have to have the same dimension

```r
A+C                                         # non-conformable
```

```
## Error in A + C: non-conformable arrays
```

Question: Give two ways that you might try to add A and C.

Transpose

The function `t()` takes the transpose of a matrix:

```r
t(B)
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```r
t(A)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
```

Multiplication

Matrix multiplication is a little more tricky – there are two types of matrix product.

The `*` operator takes an *element-wise* product – multiplying the corresponding elements together:

```r
A*B                                         # element-wise product
```

```
##      [,1] [,2] [,3]
## [1,]    1    6   15
## [2,]    8   20   36
```

(like addition, `A` and `B` have to have the same dimensions. You can't do `A*C` – try it).

You can produce standard matrix muliplication using

```r
A%*%C
```

```
##      [,1] [,2]
## [1,]   22   49
## [2,]   28   64
```

so long as the inner dimensions match. Not that in this case

```r
A%*%B                                       # non-conformable
```

```
## Error in A %*% B: non-conformable arguments
```

does not work.

In BTRY/STSCI 4030, we will be particularly interested in a matrix operating on a vector. We can achieve this by defining our vector as an n-by-1 matrix

```r
y=matrix(1:3,3,1)
A%*%y
```

```
##      [,1]
## [1,]   22
## [2,]   28
```

But R will also interpret a vector in exactly this way

```
A%*%x                                    # x is treated the same as a 3 by 1 matrix
```

```
##      [,1]
## [1,]   22
## [2,]   28
```

Note that R does keep different types of object in memory, so it believes that `y` is a matrix but that `x` is not:

```
is.matrix(y)
```

```
## [1] TRUE
```

```
is.matrix(x)
```

```
## [1] FALSE
```

Nonetheless, we can perform element-wise calculations

```
x*y
```

```
##      [,1]
## [1,]    1
## [2,]    4
## [3,]    9
```

As well as obtaining the inner product (just a number)

```
t(x)%*%y                                 # inner product of two vectors
```

```
##      [,1]
## [1,]   14
```

And the outer product – a matrix of ever possible combination of element products)

```
y%*%t(x)                                 # outer product of two vectors
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    2    4    6
## [3,]    3    6    9
```

Question

Matrix multiplication and addition follow the usual order of operations and in particular, $(A+B)C = AC+BC$

1. Show that this is the case for our A, B and C.
2. In general, by looking at $[(A+B)C]_{ij}$, show that this always holds (pen and paper exercize).

Matrix inverses

We can only invert a square matrix, so let's produce one, say:

```
M = matrix(data=c(1,1,-1,0),nrow=2,ncol=2)
```

We can now solve this matrix with

```
iM = solve(M)
```

and we can verify that this is indeed the inverse

```
M%*%iM
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Exercize verify that this is a left inverse as well as a right inverse. Why should they be the same? Verify that `M` is the inverse of `iM`.

Let's try this on a larger, more complicated matrix, say with random entries:

```
M=matrix(data=rnorm(9),nrow=3,ncol=3)
iM=solve(M)
iM%*%M
```

```
##              [,1]         [,2]         [,3]
## [1,] 1.000000e+00 0.000000e+00 1.110223e-16
## [2,] 0.000000e+00 1.000000e+00 2.775558e-17
## [3,] 1.110223e-16 5.551115e-17 1.000000e+00
```

What has happened here? You'll notice that all the entries of the product are *close to* what they should be and if we `round` them we get something sensible

```
?round                              # get help on "round" function
help(round)
round(iM%*%M,1)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

Here we get imperfect answers because when R calculates `iM`, the operations all drop some decimal places(it only stores numbers to about 16 places – see BTRY 3520), which corresponds to an accuracy of about 1e-15, which is what we see.

Spectral Decompositions

Generally, we will only need the spectral decomposition as a theoretical construct, but it's worth looking at briefly. The `eigen` function will do this for you. Here we will use a 3 by 3 matrix

```
D = diag(3) + matrix(1,nrow=3,ncol=3)
E = eigen(D)
```

Where the `values` element of the resulting list gives the eigenvalues and the `vectors` are the eigen-vectors. From this, we should be able to reconstruct D by:

```
E$vectors %*% diag(E$values) %*% t(E$vectors)
```

```
##      [,1] [,2] [,3]
## [1,]    2    1    1
## [2,]    1    2    1
## [3,]    1    1    2
```

Questions: 1. We saw in class that we can calculate D^2 by squaring the eigenvalues. Show that this is the case here, and that the eigenvalues of D^2 are the squares of `E$values`.

2. Veryify that $tr(D)$ is the sum of its eigenvalues.

Linear Regression

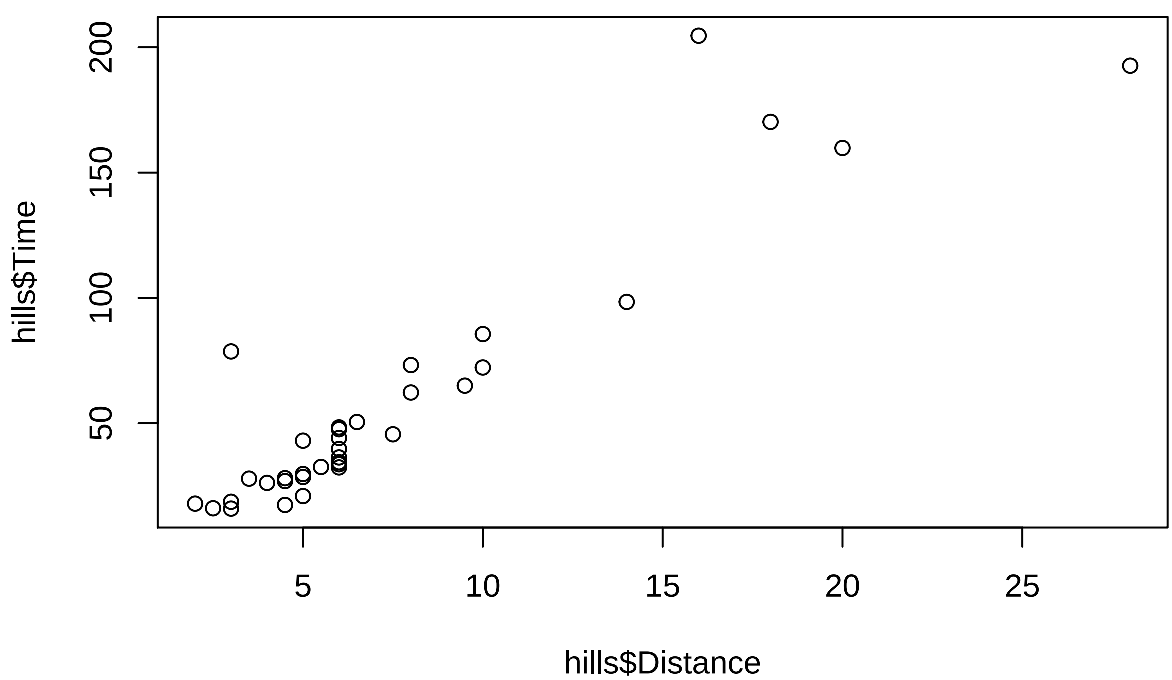
Here we will use a new data set. We did this manually before, but you can also write code to read the data in automatically:

```
hills=read.csv("hills.csv")
names(hills)
```

```
## [1] "Race"     "Distance" "Climb"    "Time"
```

We can take a look at what these data look like

```
plot(hills$Distance,hills$Time)
```



and fit a linear model

```
fit=lm(Time~Distance,data=hills)
summary(fit)
```

```
##
## Call:
## lm(formula = Time ~ Distance, data = hills)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -35.745  -9.037  -4.201   2.849  76.170
```

```
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -4.8407     5.7562  -0.841    0.406
## Distance      8.3305     0.6196  13.446 6.08e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 19.96 on 33 degrees of freedom
## Multiple R-squared:  0.8456, Adjusted R-squared:  0.841
## F-statistic: 180.8 on 1 and 33 DF,  p-value: 6.084e-15
```

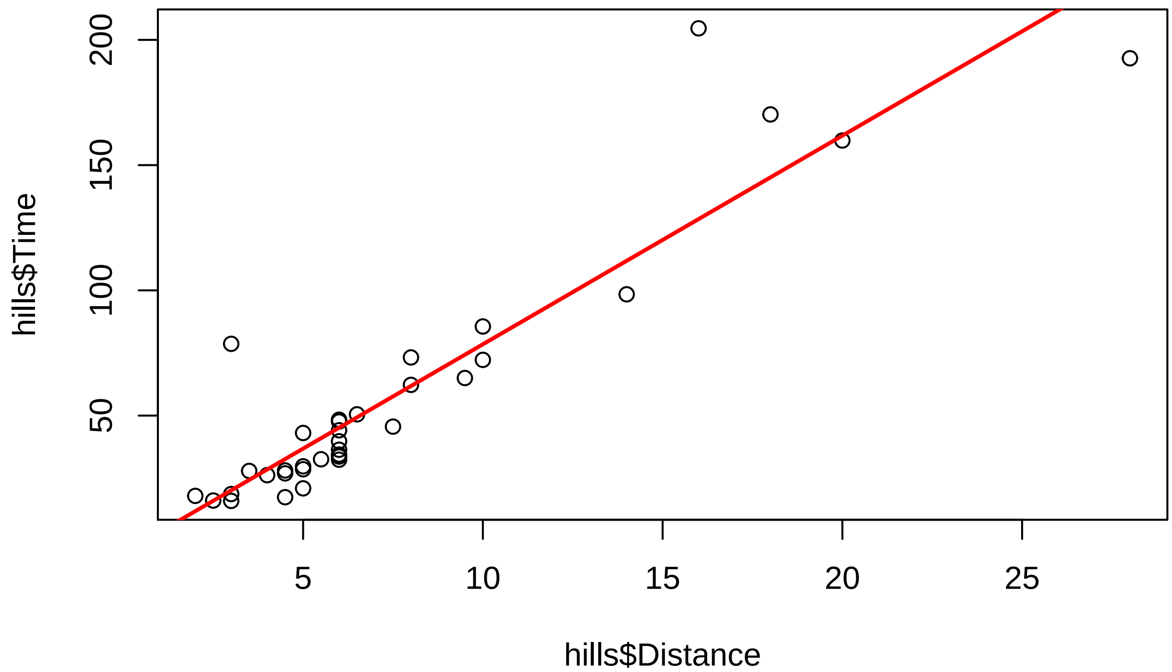
We can look at a sum of squares for this fit

```r
anova(fit)
```

```
## Analysis of Variance Table
##
## Response: Time
##           Df Sum Sq Mean Sq F value    Pr(>F)
## Distance   1  71997   71997  180.79 6.084e-15 ***
## Residuals 33  13142     398
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

and extract the coefficients and add them to a plot

```r
plot(hills$Distance,hills$Time)
b=fit$coef
abline(b)
abline(b,lwd=2,col="red")
```



Reconstrucing Linear Regression

The purpose of BTRY/STSCI 4030 is to recover the mathematics that gets you these estimates, so let's reconstruct this.

First, to get things into 'standard' notation, we'll re-name some things:

```
y=hills$Time
n=length(y)
```

We also need an X matrix, which includes the column of 1's along with Distance,

```
X=cbind(rep(1,n),hills$Distance)
```

Now, we can build up the estimate $\hat{\beta} = (X^T X)^{-1} X^T y$ in steps

```
XtX=t(X)%*%X
iXtX=solve(XtX)
bhat=iXtX%*%t(X)%*%y
```

Exercize: compare this with the estimates from `lm()`: how would you measure the discrepancy between these in R?

In order to get a confidence intervals for the slope, we need to obtain the residual standard error. For this, we first need to calculate the residuals $r = y - X\beta$:

```
r=y - X%*%bhat
```

These are then squared and divided by the residual degrees of freedom to get the variance; we can get the standard deviation by obtaining their standard error

```
s2hat=sum(r^2)/(n-2)
shat=sqrt(s2hat)
```

Compare this with "Residual standard error" from the lm summary output.

Now we need to obtain the variance of $\hat{\beta}$, which we know is $\sigma^2(X^TX)^{-1}$. In this case, we'll just extract the diagonals (ie, get the variance of β_0 and β_1 rather than worry about their covariance):

```
se=sqrt(s2hat*diag(iXtX))
```

which you should compare to the standard error values from the lm summary. We'll use the standard error of the slope below, so it will be helpful to extract it

```
se2 = se[2]
```

A Simulation

We have nice theory that says that the estimate $\hat{\beta}_1$ ought to be approximately normally distributed with the standard deviation that we calculated above. Let's see if it's true.

To do this, we'll conduct a simulation. The interpretation of the statement above is that "If we repeated the experiment many times, the collection of estimates ought to have a normal-ish histogram". So let's try it.

Each of 1000 times, we'll create a new "data set",by taking the same slope, intercept and x values, but we'll generate our own standard errors. We can then "re-estimate" $\hat{\beta}$. We'll record the value each time we do it, and then we can create a histogram of these values.

First we need to do a bit of preparation, creating matrices to record the y's and the coefficients:

```
N=1000            # This many simulations
Y=matrix(0,N,n)   # One row of y's for each simulation
B=matrix(0,N,2)   # One row of coefficient estimates for each simulation
```

Now we run through the simulations using a `for` loop. The syntax below says the following: 1. Set `i` to be each of the elements of `1:N` in turn. 2. Run the code in the braces for that value of `i` In this case, for each of `i` is 1 up to 1000, we simulate new data, estimate parameters and record these in the `i`th row of `B`:

Just to check that `Y` and `B` are of the right size:

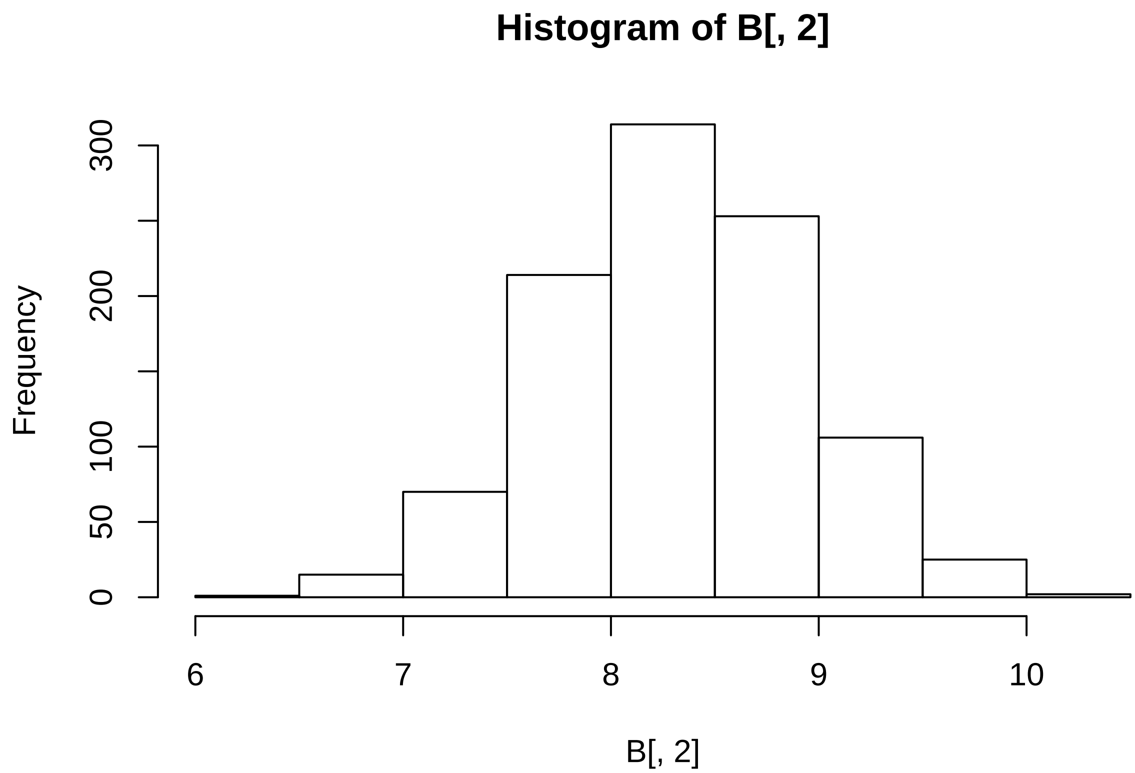
```
dim(Y); dim(B)
```

```
## [1] 1000   35
```

```
## [1] 1000    2
```

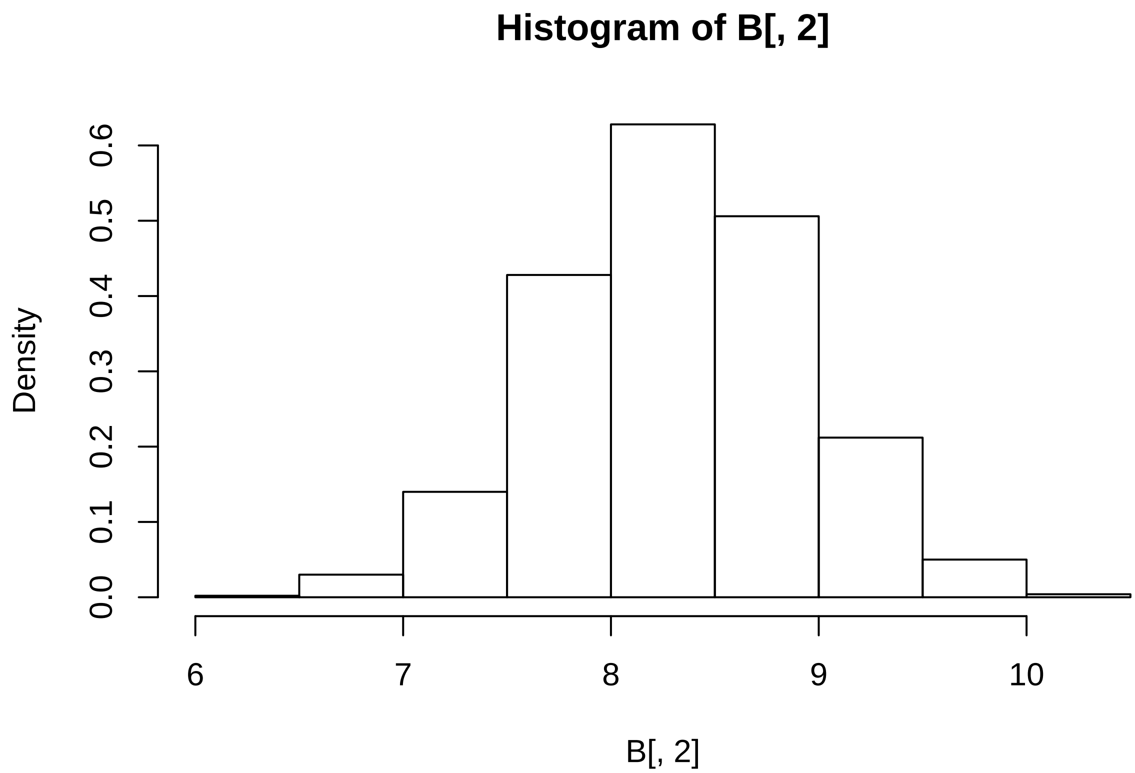
Now let's have a look at the histogram of simulated slopes

```
hist(B[,2])
```



We can change the option, so we get a probability rather than a frequency:

```
hist(B[,2],prob=TRUE)
```

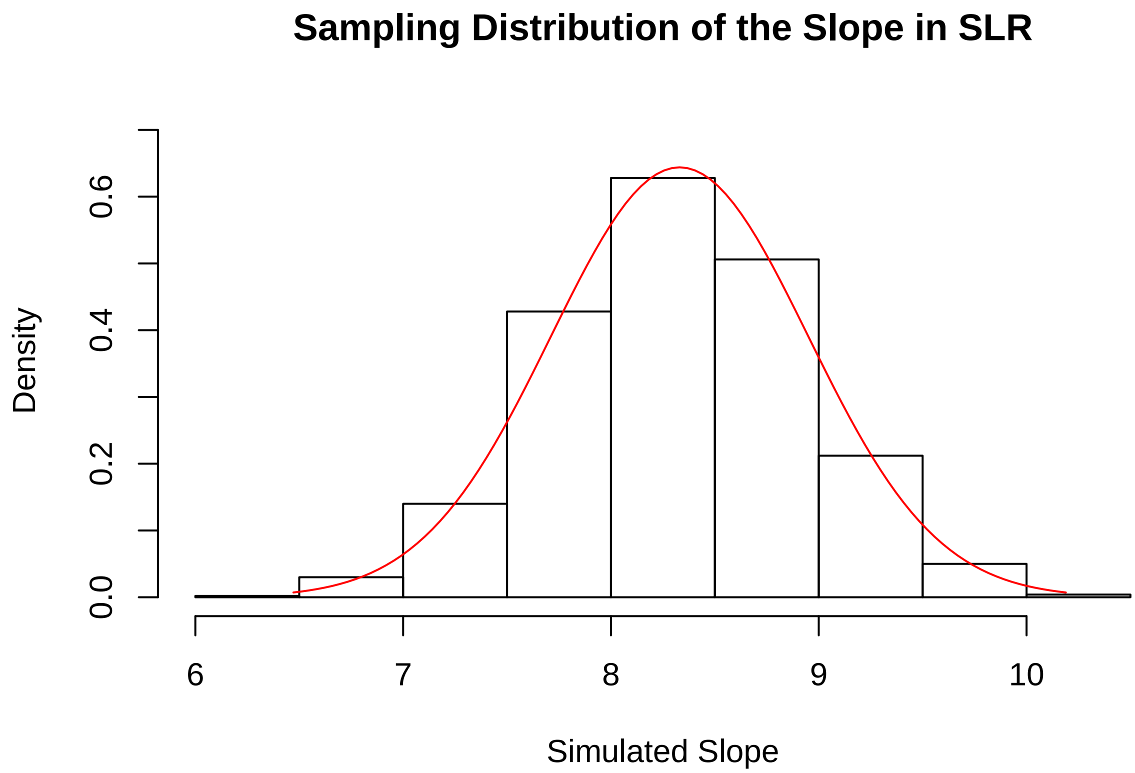


This should be approximately normal with the correct mean and varaince, so let's overlay the density that we hope to see.

First we define a set of plotting points, then we evaluate the density using `dnorm`:

Now we re-draw the histogram and overlay the density

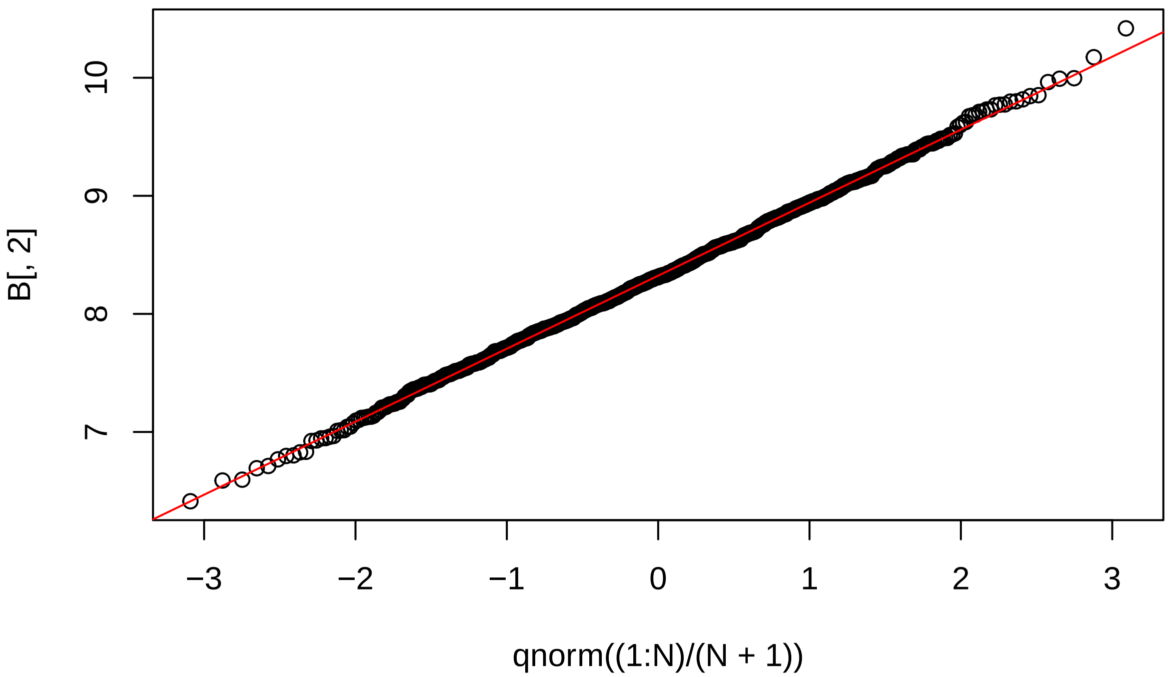
```
hist(B[,2],prob=TRUE,ylim=c(0,1.1*fmax),
     main="Sampling Distribution of the Slope in SLR",
     xlab="Simulated Slope")
lines(x,f,col="red")
```



This looks pretty good.

An alternative way of evaluating normality in our collection of estimated slopes is to produce a QQ-plot:

```
qqplot(qnorm((1:N)/(N+1)),B[,2])
qqline(B[,2],col="red")
```



B[, 2]
10
9
8
7
−3
−2
−1
0
1
2
3
qnorm((1:N)/(N + 1))