Distance Sampling

Heather Gaya

6/20/21

Time to talk about Distance Sampling! Distance sampling can be used in a wide variety of models from occupancy and N-mixture models to individual-based models.

Before you use NIMBLE make sure R, and Rtools or Xcode are updated on your computer (otherwise a weird "shared library" error can come up). JAGS is downloadable here: https://sourceforge.net/projects/mcmc-jags/ and NIMBLE can be found here: https://r-nimble.org/download. For this tutorial I am using NIMBLE version 0.10.1

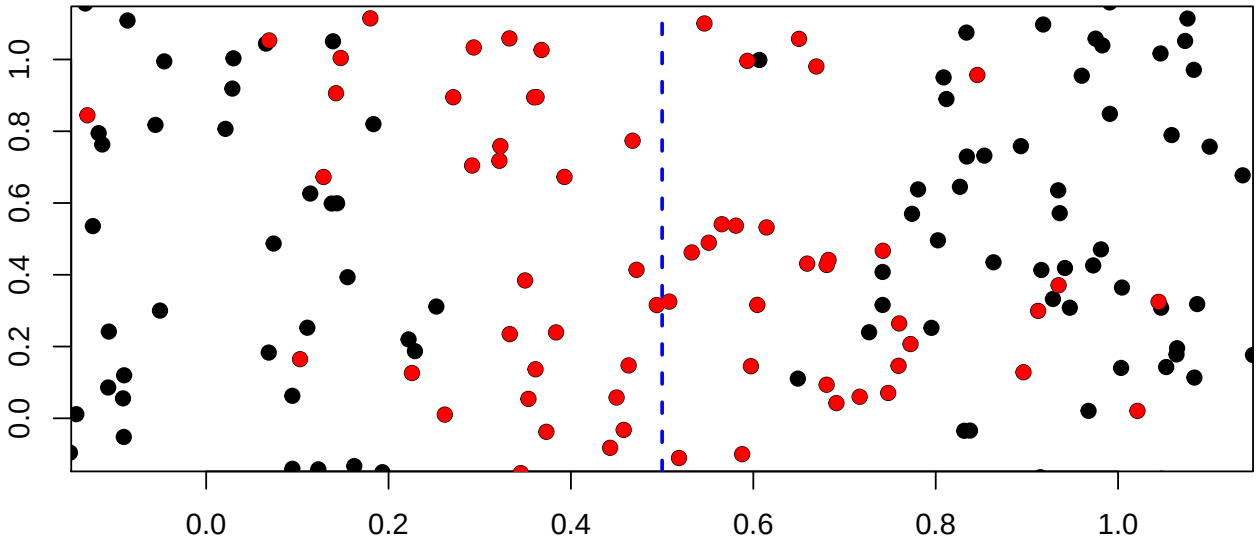
Please send any questions or suggestions to heather.e.gaya(at)gmail.com or find me on twitter: doofgradstudent

Contents

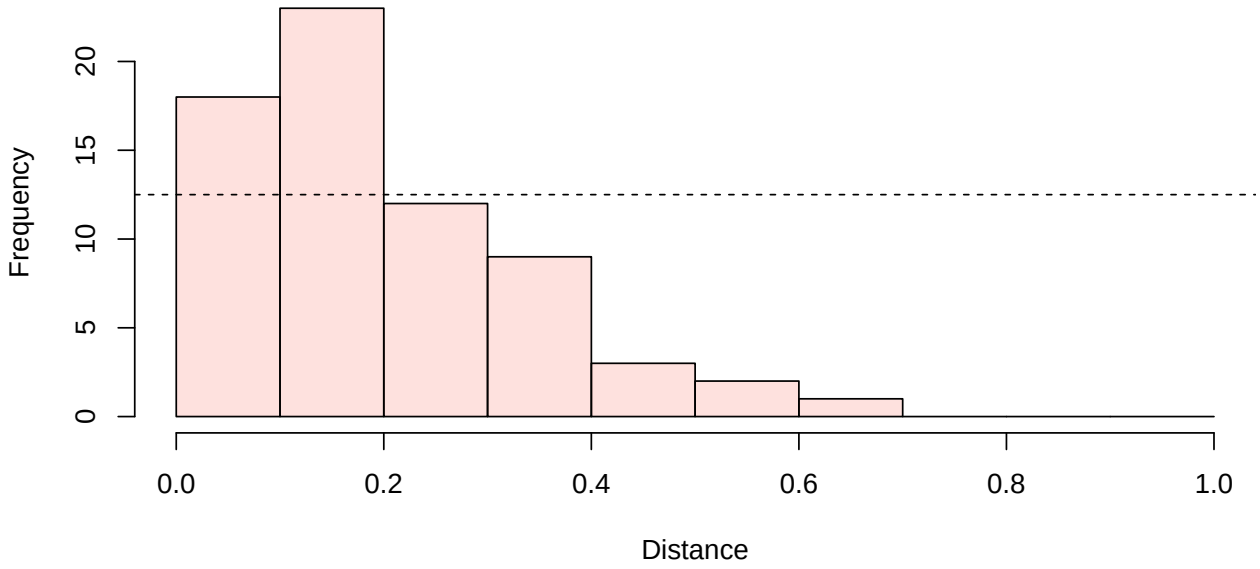
What is Distance Sampling?

Distance sampling is a versatile way of using the distance of an animal at detection to help estimate the probability of detecting an animal, allowing for estimates of abundance or density. The main idea behind distance sampling is that in general, it is easier to detect things closer to you than things that are farther away (some variants on this for sampling from planes or something that limits your sight until a certain distance is reached). The rate of "drop off" for detection tells you something about what proportion of the available animals you actually detected, which allows for density or abundance estimates.

For line transects, that might look something like the plot below. The dots are the true distribution of animals on plot, the blue line is the transect, with red dots indicating detected individuals. Distance is measured as perpendicular to the transect. For conventional distance sampling, detection probability on the transect at distance 0 is assumed to be 1 (perfect detection).

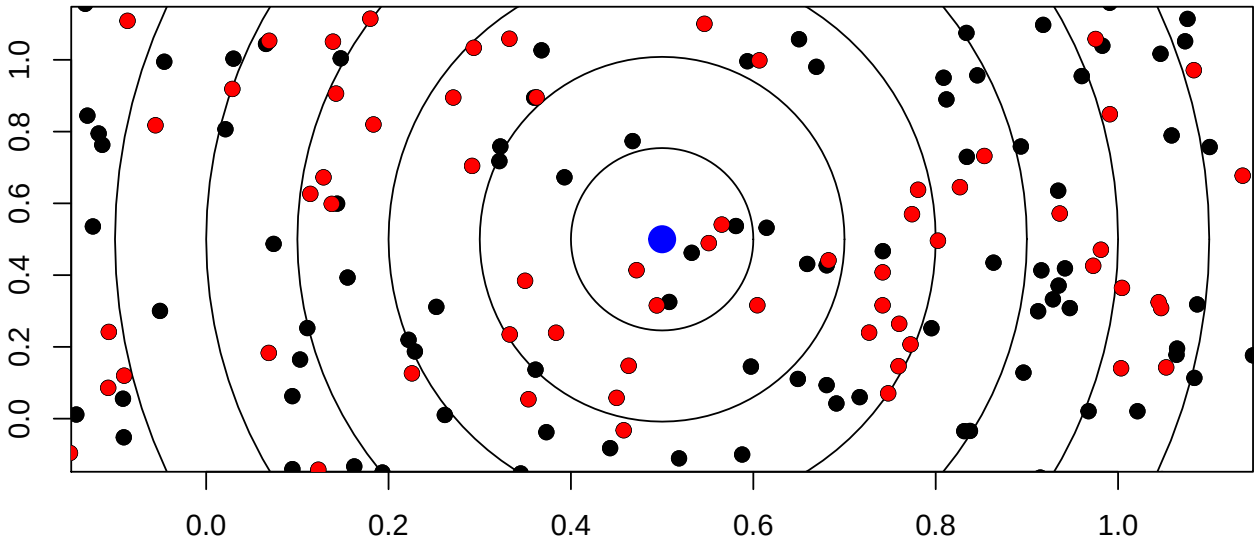


Below the histogram shows the perpendicular distances of detected animals, though the true distribution is roughly uniform (dotted line).

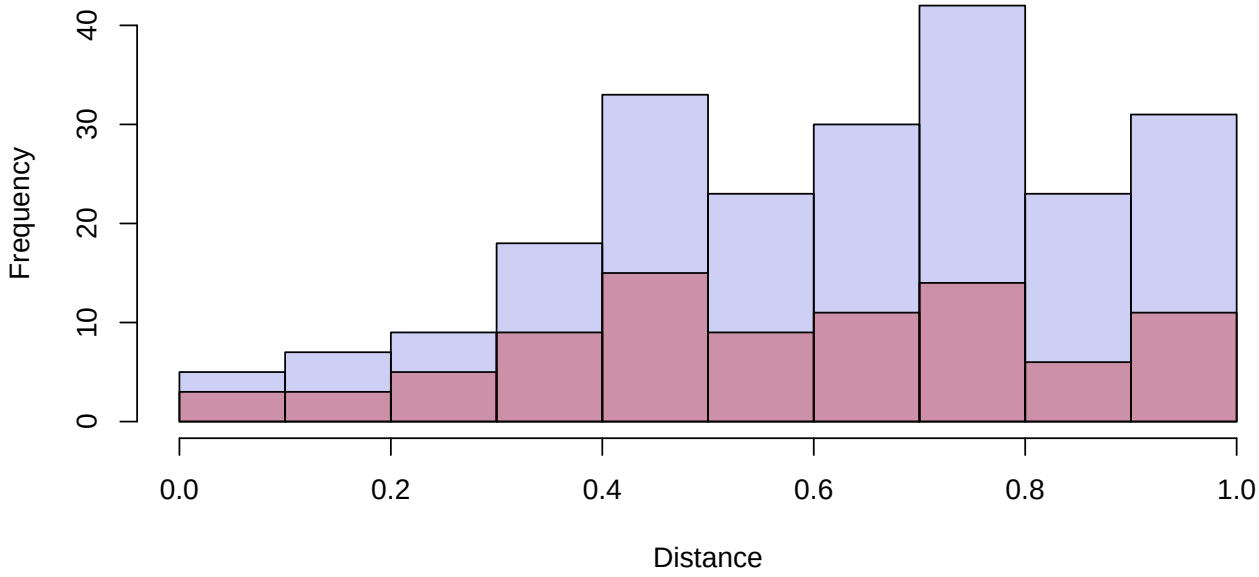


For point transects, there is a slight difference. As a circle moves away from the center there is an increasing area associated with any given distance from the center of the circle. If animals are randomly located in the circle, then we expect the true distribution of distances to actually show an increase in farther distance bins.

Below, the blue dot shows the location of the observer with .1 unit rings to show distance bins. The red points indicate observed individuals.



While we still have a higher chance of seeing/detecting individuals closer to us (red bars), the true distribution of distances will no longer be uniform (blue bars). Instead, more individuals will be farther away from us.



With this in mind, let's go through a few scenarios where we might use distance sampling.

Some Example Scenarios

There are many ways to use distance sampling and many extensions that allow you to relax assumptions. For now we'll work on some of the basic applications. The most important things to know before you start are what those assumptions are! The 4 assumptions are:

- Animals are distributed independent of lines (or points)
- On the line, detection is certain (100% detection at distance 0, this can be relaxed under some models)
- Distances are recorded correctly

- Animals don't move before detection (and no double counting of individuals)

Line-Transect Surveys - Sheep

Line transects are used for many different species but the most basic setup is randomly placed, equally spaced parallel transects. In some cases, it may be easiest to count how many objects/individuals are in distance bins rather than exact distances. For instance, maybe we are counting sheep from a car (why? no idea, but you are) and can't measure the exact distance of each individual sheep. We want to know if sheep are doing better in our burned pastures that removed hardwoods or our unburned pastures that include clumps of trees. In this case, our data might look like this:

Distance.Bin	Plot.A	Plot.B	Plot.C	Plot.D	Plot.E	Plot.F
0	18	10	7	20	18	5
5	10	6	4	18	10	4
10	8	5	2	14	8	3
15	4	1	1	7	5	1
20	2	1	1	2	1	0

For this survey, we drove 5000 m of transect in each plot.

We also have data about the percent hardwoods in each plot in a separate data file.

Plot	Hardwood
A	10
B	20
C	80
D	25
E	15

Point Count Surveys - Bird Surveys

Point count surveys are commonly used to survey for birds. For my current research, we use point counts to record songbird presence and abundance in North Carolina, USA. Each point count is surveyed for 2.5 minutes in 4 consecutive surveys and the location/distance of each bird detected is estimated in each survey period. Unless you have a very abundant bird it's often hard to get precise estimates from just one point, so for this example we'll estimate the abundance of a species across multiple points within the same plot. Here's some real data on Ovenbirds from my own research. If the bird was not detected in a time period, the distance was recorded as NA.

PointName	Wind	Noise	Temp	distanceP1	distanceP2	distanceP3	distanceP4
BearPC01	1	2	20.14	65	90	85	85
BearPC01	1	2	20.14	20	90	85	85
BearPC01	1	2	20.14	20	NA	NA	NA
BearPC01	1	2	20.14	5	5	NA	NA
BearPC01	1	2	20.14	15	25	25	20

Line-Transect Surveys - Gopher Tortoises

Gopher tortoises are a tortoise species that create burrows in sandy soils in the southern united states. For my MS, I went out and surveyed 100s of miles of transect looking for burrows and estimating tortoise densities. The survey design is fairly simple - randomly placed, equally spaced parallel transects are placed in gopher

tortoise habitat and observed walk along the transects looking for holes in the ground. When one is found, the burrow is scoped to determine occupancy. The burrow is also measured (to give an estimate of tortoise size), and the location of the burrow is recorded on a GPS. For ease of this example, we will only be working with occupied burrows. So we end up with data that looks something like this:

ObsNum	Distance
1	13.10
2	20.41
3	1.61
4	19.98
5	21.62

We also know that we walked 11422.7 meters of transect (this is real data, so the length is weird). Our interest is in the density of gopher tortoises at the site we surveyed.

General Model Format

There's many ways that distance sampling models can be written, but they all operate under the same idea - detection is easier when animals or objects are closer and decreases as they get farther away. In these examples I'm going to be using the half-normal distribution to describe that pattern of decrease, but some people will use alternative distributions so just keep that in mind if you see an exponential or hazard rate function used in distance sampling in the scientific literature.

Here's the general framework. Detection probability p for individual i is based on the distance (or distance bin) x from the observer and a latent σ parameter that adjusts how fast probability drops off. This is then multiplied by the probability $h(x)$ of being in a given distance bin.

$$p = e^{\frac{-x^2}{2\sigma^2}} h(x)$$

For line transect data, the underlying assumption is a uniform distribution of individuals at all distances from the line, so $h(x)$ drops out.

$$p = e^{\frac{-x^2}{2\sigma^2}}$$

For point count data, we need to account for the increasing probability of being at a given distance as you move away from the observer.

$$p = \int 2\pi x e^{\frac{-x^2}{2\sigma^2}} dx$$

Now before that equation above freaks you out, it's really just combining the equation of area in a circle and the half normal function. The math may look a little scary, but when it comes to the coding I think it starts to make a lot more sense.

The number of animals detected n is simply a function of abundance N and probability of detection p:

$$n = pN$$

You can also put various constraints in your model to explain variation in N between sites or years, but for now we'll stick to the simplest scenario.

Code in JAGS

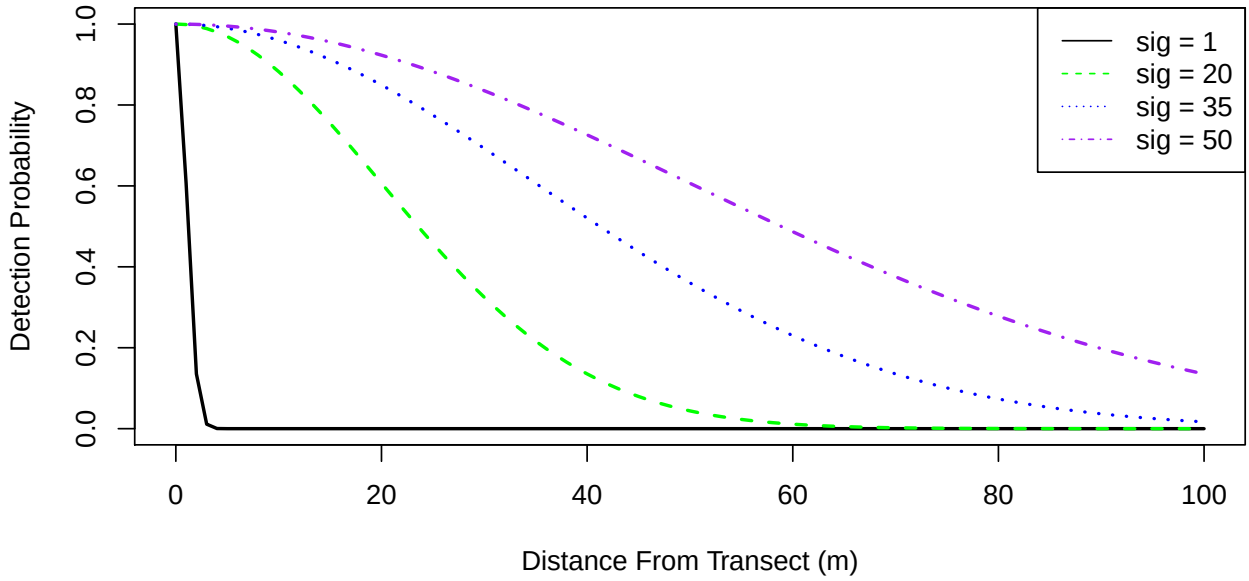
The code we use to describe this model will change a little bit from each scenario, but the basic idea is the same.

Gopher Tortoise Hierarchical Distance Model

Starting with our gopher tortoise data, we can write out our detection equation in JAGS language. We'll use a hierarchical distance sampling model for this data. Note that the distance of each burrow is going to change between individuals, so we'll need to index it.

```
p[i] <- exp(-x[i]^2/(2 * sig^2))
```

Now we need a prior for *sig* since we don't know what it is. To make this prior reasonable, we can think about what values might make sense. Here's a visual of how values of *sig* changes the detection function.



For our gopher tortoises, the transects were spaced 100m apart, so we know the maximum distance will be 50m. I also know we rarely see burrows more than 30m away (a quick look at the data will confirm this). So we probably want our sigma to be somewhere around 20 or less (since that shows a curve with very low probability of detection at 30m). Let's see how the model runs with a vague prior on sigma bound between 5 and 30. We can always adjust later.

```
p[i] <- exp(-x[i]^2/(2 * sig^2))
sig ~ dunif(5, 30)
```

Now we can link that to our data y, the record of all the individuals we detected.

```
y[i] ~ dbin(p[i])
```

Now comes the tricky bit. Nothing in the above really mentions how much space we're analyzing here. Sure you could get a number if you divided $\frac{\sum y}{mean(p)}$ but it doesn't really mean anything. So instead, we're going to use something fancy called data augmentation. Our first step is to add a bunch of extra individuals to the bottom of our dataframe (we do this outside JAGS, back in R). We don't know anything about them except we didn't see them.

To make sure we keep track of the real vs. added data, we'll add a variable w to keep track. For the ones we saw, $w = 1$. But for the new data, $w = NA$ since we don't know if the individuals were actually there or not. We'll let the model decide that.

We also want to keep track of if we SAW the burrow or not. This will become the y variable in our code. In this case, all the burrows in our real data will have $y = 1$, and all the ones we didn't see will have $y= 0$. It's important to understand that y is an OBSERVABLE variable but w is a LATENT variable.

```
newburrows <- data.frame(Distance = c(burrows$Distance, rep(NA, 100)),
    w = c(rep(1, nrow(burrows)), rep(NA, 100)), y = c(rep(1, nrow(burrows)),
        rep(0, 100)))
head(newburrows, n = 2)
##   Distance w y
## 1 13.10248 1 1
## 2 20.40732 1 1
tail(newburrows, n = 2)
##     Distance  w y
## 149       NA NA 0
## 150       NA NA 0
```

Now let's go back to thinking about our model. Now that we have some individuals in our dataset that don't have distances, we'll have to create a prior for the distribution of distances. Luckily, one of the main assumptions of distance sampling is that animals are uniformly spread out across the area you're sampling! In this case, we're sampling animals from 0 to 50m distances. This means we already know the prior we want to use.

```
x[i] ~ dunif(0, 50)
```

Awesome! Okay, two last things to consider. Where do those pesky w's come into the model and how does this tell us a density estimate exactly?

First up, we know that you can only detect an individual if it's real. Or, we hope so anyway :) That means that we can change what we wrote for y.

```
y[i] ~ dbern(p[i] * w[i])
```

Now we just need to tell the model how w is distributed. And in this case, it's just another coin flip, so another bernouli.

```
w[i] ~ dbern(psi)
```

Though of course we now have to put a prior on psi, which could be anything from 0 to 1. Psi isn't actually meaningful in it of itself, because it's just tell us what proportion of our new dataset is real. If we added 10000 rows of unknowns instead of 100, psi would come out much smaller.

```
psi ~ dunif(0, 1)
```

Finally, our density is just the total number of real burrows $\sum w$ divided by the total area of interest. In our case, we walked 11422.7m of transect which are spaced 100 m apart for a total area of $(11422.7)(50)(2) = 1142270m^2$ or 114.227 hectares.

```
D = sum(w)/114.227
```

Awesome! Here's that full model all together.

```
modelstring.torts = "
  model
{
for (i in 1:n.torts){
    w[i] ~ dbern(psi) #real?
    x[i] ~ dunif(0,50) #distance
    p[i] <- exp(-x[i]^2/(2*sig^2)) #p detection
    y[i] ~ dbern(p[i]*w[i]) #detected?
    }
```

```
sig ~ dunif(5,30)
psi ~ dunif(0,1)

N <- sum(w) #total torts
D <- N/114.227 #density in torts/hectare
}
"
```

Let's get that model up and running.

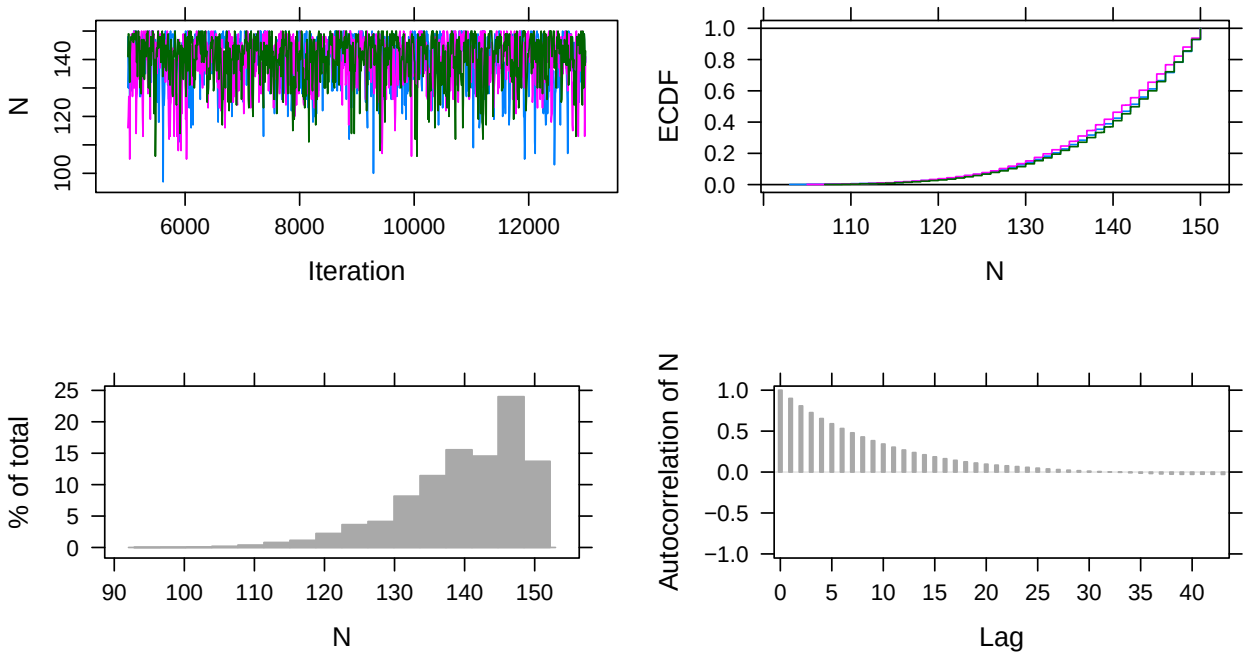
```
library(runjags)
jd.torts <- list(n.torts = nrow(newburrows), x = newburrows$Distance, w = newburrows$w,
    y = newburrows$y)
ji.torts <- function(x) {
    list(sig = runif(1, 10, 30), psi = runif(1, 0, 1))
}
jp.torts <- c("N", "D", "sig", "psi")
Torts <- run.jags(model = modelstring.torts, monitor = jp.torts, data = jd.torts,
    n.chains = 3, inits = ji.torts, burnin = 4000, sample = 8000, adapt = 1000,
    method = "parallel")
## Warning: You attempted to start parallel chains without setting different PRNG
## for each chain, which is not recommended. Different .RNG.name values have been
## added to each set of initial values.
## Calling 3 simulations using the parallel method...
## Following the progress of chain 1 (the program will wait for all chains
## to finish before continuing):
## Welcome to JAGS 4.3.0 on Fri Jun 25 19:53:10 2021
## JAGS is free software and comes with ABSOLUTELY NO WARRANTY
## Loading module: basemod: ok
## Loading module: bugs: ok
## . . Reading data file data.txt
## . Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 250
##    Unobserved stochastic nodes: 202
##    Total graph size: 1215
## . Reading parameter file inits1.txt
## . Initializing model
## . Adapting 1000
## -------------------------------------------------| 1000
## +++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## Adaptation successful
## . Updating 4000
## -------------------------------------------------| 4000
## ************************************************** 100%
## . . . . . Updating 8000
## -------------------------------------------------| 8000
## ************************************************** 100%
## . . . . Updating 0
## . Deleting model
## .
## All chains have finished
```

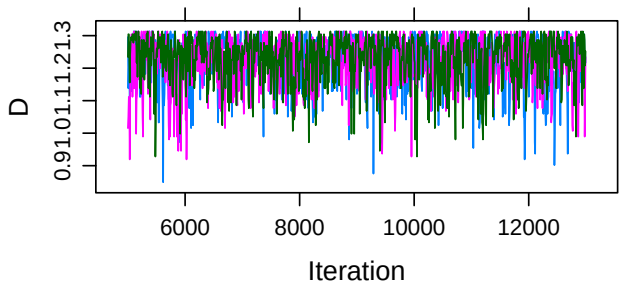
```
## Simulation complete.  Reading coda files...
## Coda files loaded successfully
## Calculating summary statistics...
## Calculating the Gelman-Rubin statistic for 4 variables....
## Finished running the simulation
mod.Torts <- summary(Torts)
```

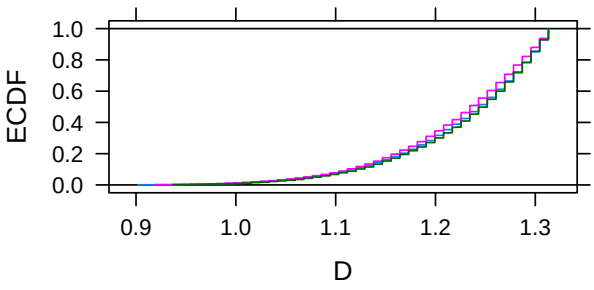
```
mod.Torts
##        Lower95      Median    Upper95        Mean          SD Mode       MCerr
## N   123.000000 142.0000000 150.000000 139.9165833 8.60205643  149 0.238750672
## D     1.076800   1.2431400   1.313170   1.2248994 0.07530654   NA 0.002090134
## sig  10.211000  12.2288500  14.570500  12.2997918 1.12027832   NA 0.013567714
## psi   0.807729   0.9422695   0.999999   0.9272938 0.05994583   NA 0.001659405
##     MC%ofSD SSeff      AC.10     psrf
## N       2.8  1298 0.34079924 1.002302
## D       2.8  1298 0.34079946 1.002302
## sig     1.2  6818 0.02992933 1.000258
## psi     2.8  1305 0.33274478 1.002683
```

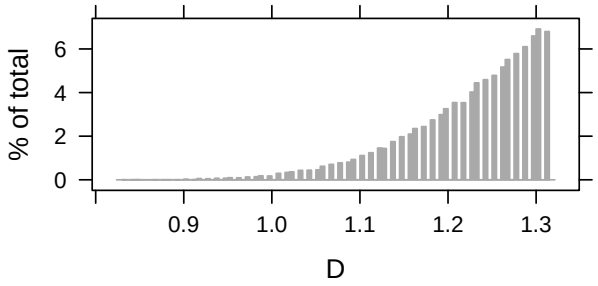
Always good practice to plot your chains.

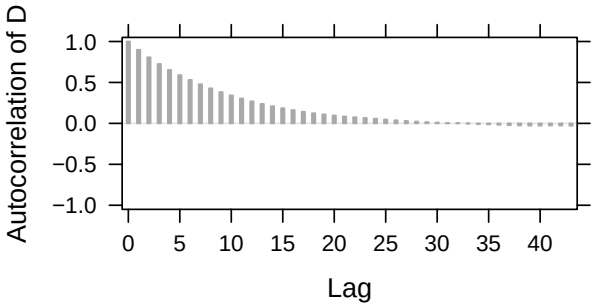
```
plot(Torts)
## Generating plots...
```

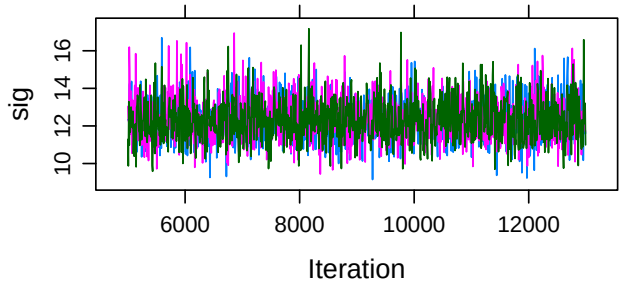


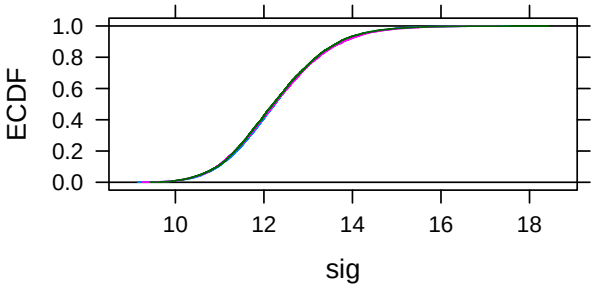

D
0.9
1.0
1.1
1.2
1.3
6000
8000
10000
12000
Iteration

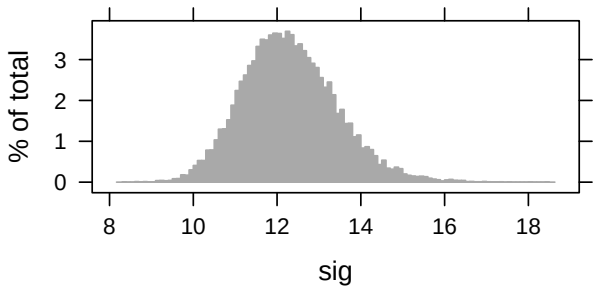

ECDF
0.0
0.2
0.4
0.6
0.8
1.0
0.9
1.0
1.1
1.2
1.3
D

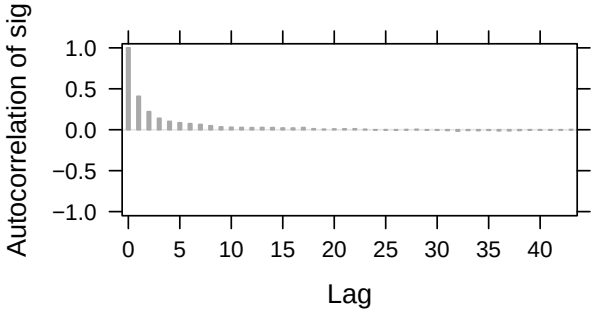

% of total
0
2
4
6
0.9
1.0
1.1
1.2
1.3
D

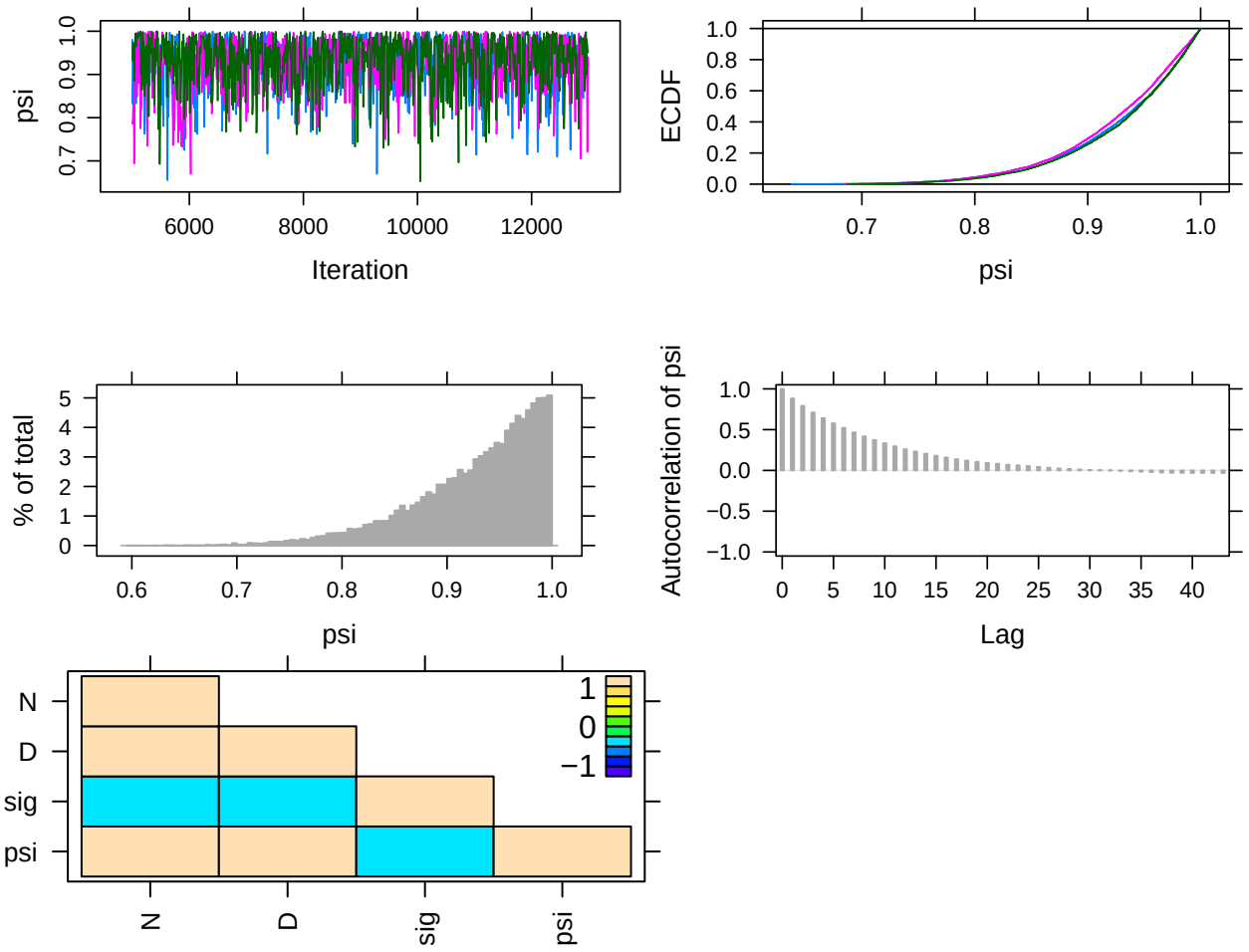

Autocorrelation of D
−1.0
−0.5
0.0
0.5
1.0
0
5
10
15
20
25
30
35
40
Lag


sig
10
12
14
16
6000
8000
10000
12000
Iteration


ECDF
0.0
0.2
0.4
0.6
0.8
1.0
10
12
14
16
18
sig


% of total
0
1
2
3
8
10
12
14
16
18
sig

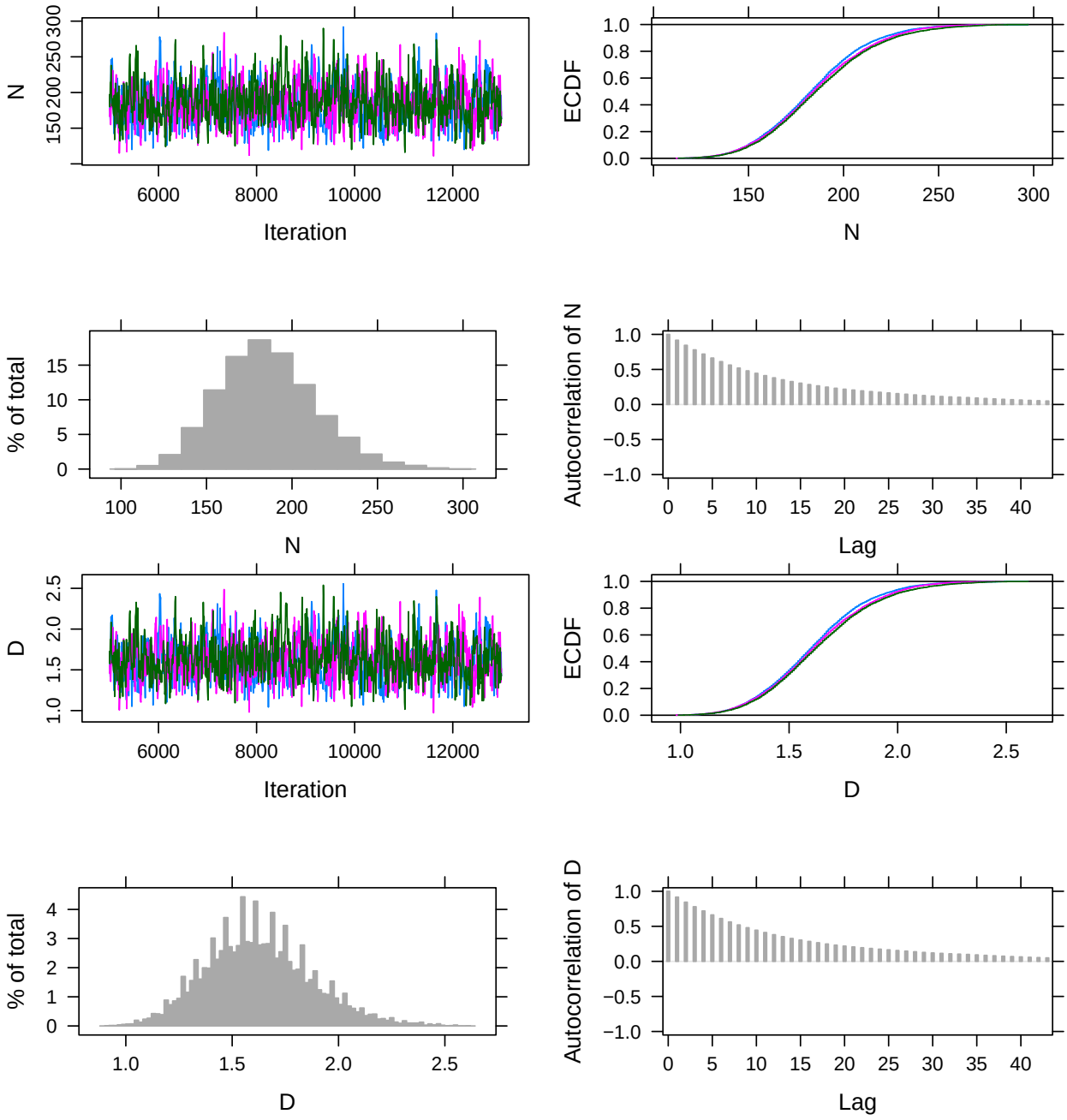

Autocorrelation of sig
−1.0
−0.5
0.0
0.5
1.0
0
5
10
15
20
25
30
35
40
Lag

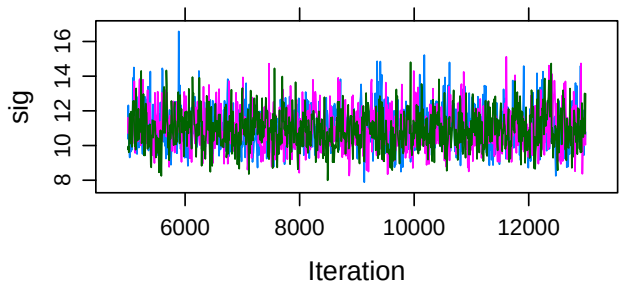


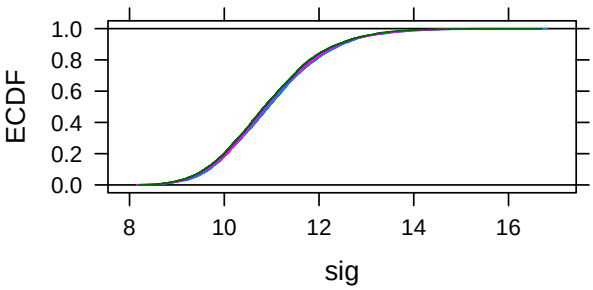
Uh-oh, psi (the proportion of burrows in our new dataset that are real) is pushing up against 1! That's no good. Let's add more augmented data and try again.

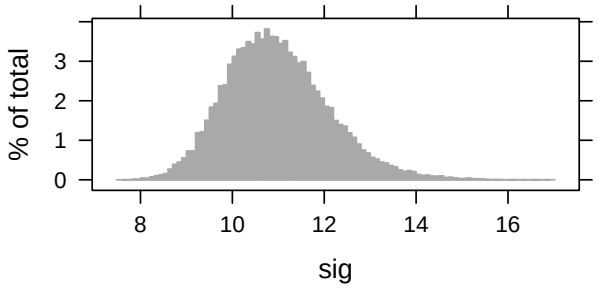
```
newburrows <- data.frame(Distance = c(burrows$Distance, rep(NA, 300)),
    w = c(rep(1, nrow(burrows)), rep(NA, 300)), y = c(rep(1, nrow(burrows)),
        rep(0, 300)))
jd.torts <- list(n.torts = nrow(newburrows), x = newburrows$Distance, w = newburrows$w,
    y = newburrows$y)
ji.torts <- function(x) {
    list(sig = runif(1, 10, 30), psi = runif(1, 0, 1))
}
jp.torts <- c("N", "D", "sig", "psi")
Torts <- run.jags(model = modelstring.torts, monitor = jp.torts, data = jd.torts,
    n.chains = 3, inits = ji.torts, burnin = 4000, sample = 8000, adapt = 1000,
    method = "parallel")
## Warning: You attempted to start parallel chains without setting different PRNG
## for each chain, which is not recommended. Different .RNG.name values have been
## added to each set of initial values.
## Calling 3 simulations using the parallel method...
## Following the progress of chain 1 (the program will wait for all chains
## to finish before continuing):
## Welcome to JAGS 4.3.0 on Fri Jun 25 19:53:20 2021
## JAGS is free software and comes with ABSOLUTELY NO WARRANTY
## Loading module: basemod: ok
```

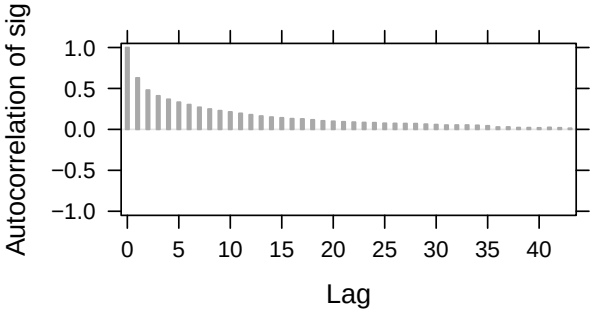
```
## Loading module: bugs: ok
## . . Reading data file data.txt
## . Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 450
##    Unobserved stochastic nodes: 602
##    Total graph size: 2815
## . Reading parameter file inits1.txt
## . Initializing model
## . Adapting 1000
## -------------------------------------------------| 1000
## +++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## Adaptation successful
## . Updating 4000
## -------------------------------------------------| 4000
## ************************************************* 100%
## . . . . . Updating 8000
## -------------------------------------------------| 8000
## ************************************************* 100%
## . . . . Updating 0
## . Deleting model
## .
## All chains have finished
## Simulation complete.  Reading coda files...
## Coda files loaded successfully
## Calculating summary statistics...
## Calculating the Gelman-Rubin statistic for 4 variables....
## Finished running the simulation
mod.Torts <- summary(Torts)
plot(Torts)
## Generating plots...
```

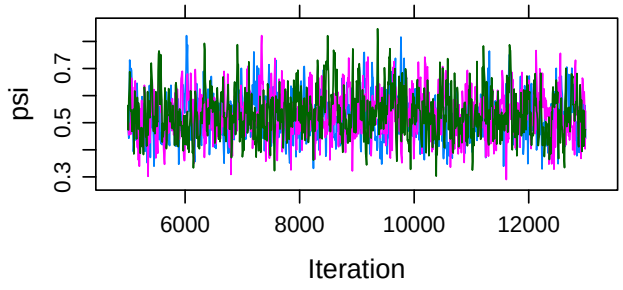

N
150 200 250 300
6000 8000 10000 12000
Iteration
ECDF
1.0 0.8 0.6 0.4 0.2 0.0
150 200 250 300
N
% of total
15 10 5 0
100 150 200 250 300
N
Autocorrelation of N
1.0 0.5 0.0 −0.5 −1.0
0 5 10 15 20 25 30 35 40
Lag
D
1.0 1.5 2.0 2.5
6000 8000 10000 12000
Iteration
ECDF
1.0 0.8 0.6 0.4 0.2 0.0
1.0 1.5 2.0 2.5
D
% of total
4 3 2 1 0
1.0 1.5 2.0 2.5
D
Autocorrelation of D
1.0 0.5 0.0 −0.5 −1.0
0 5 10 15 20 25 30 35 40
Lag

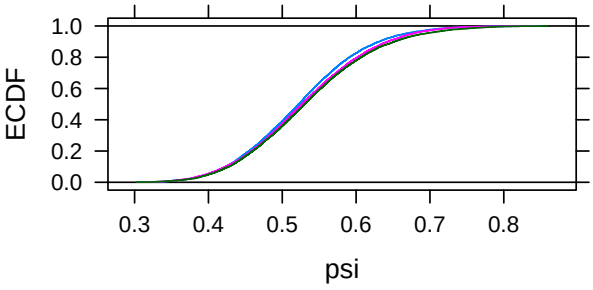


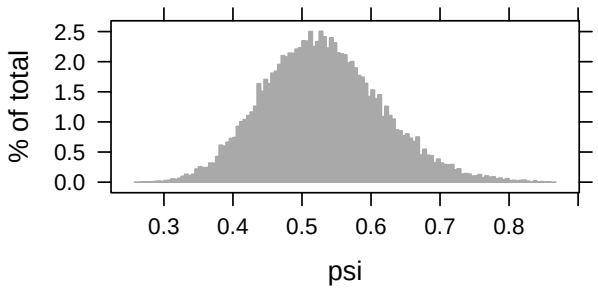


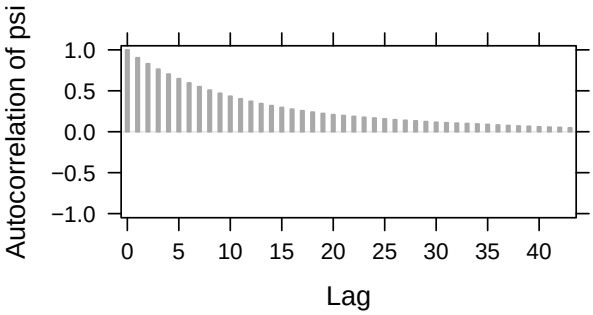


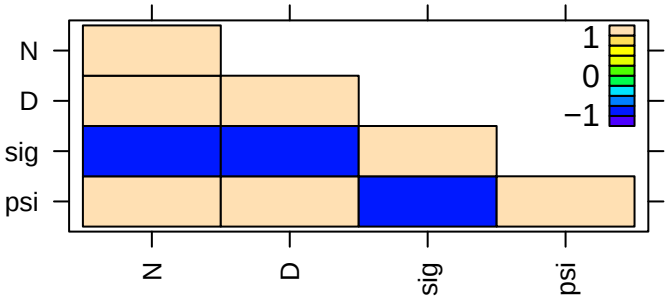












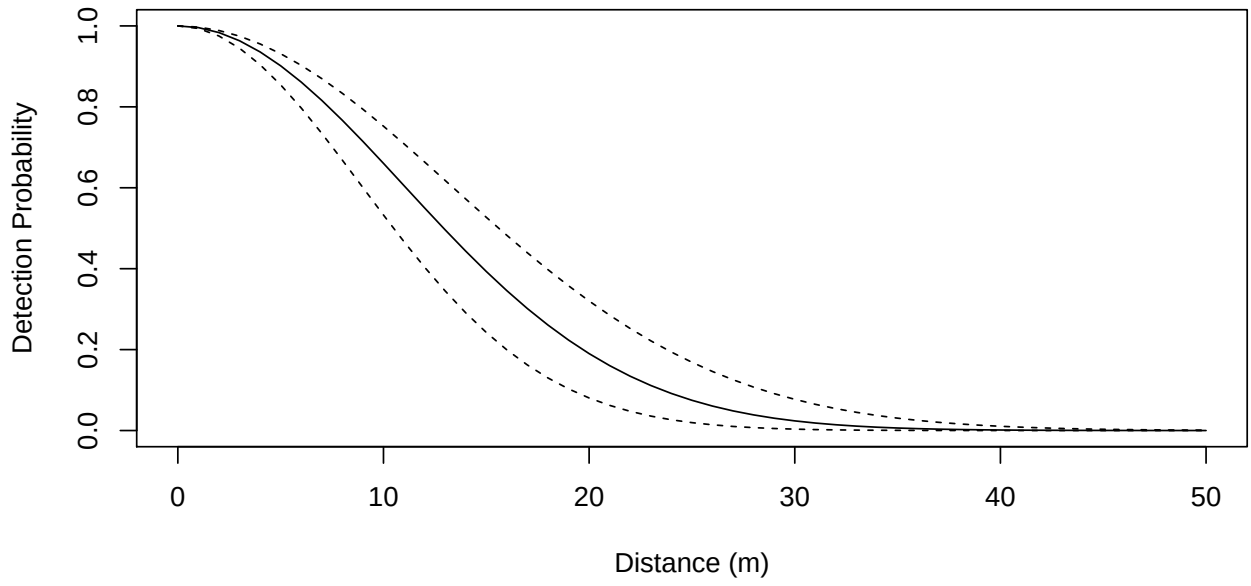
Oh yay, that looks way better.

```
mod.Torts
##         Lower95    Median    Upper95        Mean          SD Mode       MCerr
```

```
## N   131.000000 184.00000 241.000000 185.7228333 28.46153716  184 0.899127657
## D     1.146840   1.61083   2.109830   1.6259102  0.24916639   NA 0.007871407
## sig   8.909360  10.88440  13.251400  10.9760207  1.11994136   NA 0.025552547
## psi   0.371251   0.52643   0.702567   0.5301958  0.08483442   NA 0.002666823
##     MC%ofSD SSeff     AC.10     psrf
## N       3.2  1002 0.4424071 1.002520
## D       3.2  1002 0.4424072 1.002520
## sig     2.3  1921 0.2130615 1.001051
## psi     3.1  1012 0.4313619 1.003421
```

These results indicate an estimated gopher tortoise density of 1.65 tortoises per hectare (1.15 to 2.14). We could get more precision if we had walked a great length of transect or had more individuals in our dataset, but overall a decent estimate! We can also graph our detection curve, just for kicks.

```
xs <- seq(0, 50, by = 1)
plot(xs, exp(-xs^2/(2 * mod.Torts[3, 4]^2)), type = "l", xlab = "Distance (m)",
    ylab = "Detection Probability")
lines(xs, exp(-xs^2/(2 * mod.Torts[3, 1]^2)), lty = 2)
lines(xs, exp(-xs^2/(2 * mod.Torts[3, 3]^2)), lty = 2)
```



Line-Transect Surveys - Sheep

Time for our other line-transect example using sheep and binned data. Unlike with tortoises, we don't have individual-level data but unmarked grouped data to work with. We will use an N-mixture model format with some small modifications.

As per a normal N-mixture model, our first consideration is think about what creates abundance. In our sheep pasture, we think that sheep are more likely to be found in pastures with fewer hardwood seedlings. So we'll model sheep abundance based on hardwood percentage using a log-link and a poisson distribution.

```
for (i in 1:n.sites) {
    log(lambda[i]) <- beta0 + beta1 * hardwood[i]
    N[i] ~ dpois(lambda[i])  # Latent local abundance
}
```

Of course now we'll need priors for those beta terms. We can make these whatever we want and check our output later to make sure they make sense.

```
lambda.intercept ~ dunif(0, 300)
beta0 <- log(lambda.intercept)
beta1 ~ dnorm(0, 0.2)
```

Next we can think about our detection function. Since we don't have individual distances, we want to know the average probability of detection in each distance bin. We have a few options here. We could take the midpoint distance *midpt* of each distance bin and use that to calculate average detection probability *pbar* in each distance bin. This works if each distance bin is the same size.

```
pbar <- exp(-(midpt^2)/2 * sigma^2)
```

Or we can integrate the half-normal curve at each bound of the distance bin (the min and max distances) and then divide the difference by the area in our distance bin *b*. This allows for different sized distance bins. Note that this only works for the half-normal and you would need to integrate over the appropriate curve if you use something other than a half-normal detection function.

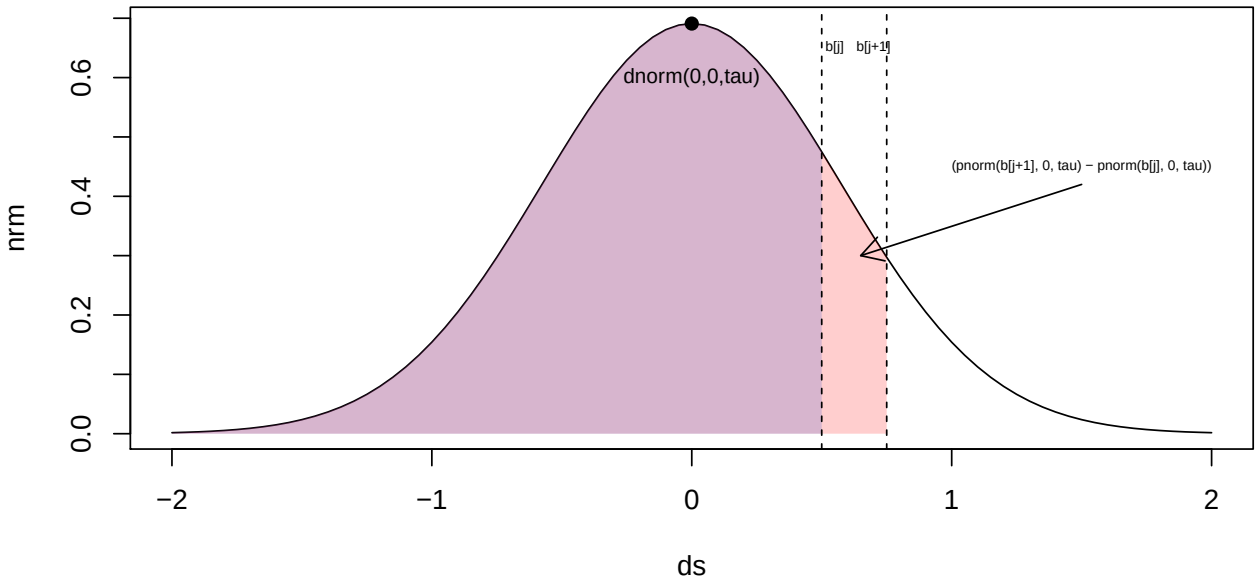
```
pbar <- ((pnorm(b[j + 1], 0, tau) - pnorm(b[j], 0, tau))/dnorm(0, 0, tau)/(b[j +
    1] - b[j])) * psi[j]
```

Okay, that looks scary, let's take a minute with that. What's happening?

First of all, *pnorm* is just asking "what's the probability of detection at distance b[j+1] (our distance bin's max distance) given a normal distribution with mean 0 and precision tau?". Then we ask that same question for the the distance bin's minimum distance (b[j]) and subtract the difference. Next, we divide by $dnorm(0, 0, tau)$ which is just a fancy way of asking "what's the density of a normal distribution centered on 0 at distance 0?. This allows us to adjust for the height of the normal distribution.

Next we divide everything by the distance covered by the distance bin. The *psi* term is just an adjustment for the total area within each bin - in our case we have 5 meter distance bins that go from 0 to 30 m, so our psi term will just be 5/30.

Stepping away from JAGS for a moment, here's what's happening above (the graph is hideous bear with me). We're just trying to calculate the area in the highlighted strip and adjust for the max height of the normal distribution. That's all we're doing.



Now that we have *pbar* for each distance bin calculated, we should probably add a prior for *tau* before we forget. Again, *tau* is just precision (1/sd^2) and acts similarly to the sigma parameter in the half normal function. Let's give it a vague prior. Note that we could add in an equation for *tau* if we had variables we

thought were affecting detection, though we don't have any in this example. Let's assume *tau* is the same across all plots.

```
tau <- 1/sd^2
sd ~ dunif(0, 30)
```

Now we just need to link abundance with our observed data via detection probability! We do this in two steps. First, the number of sheep we saw total n for each site is just a binomial draw from N. What's the probability of this draw? It's just 1-p(no detection).

In JAGS code, we can write:

```
pbar[i, nBins + 1] <- 1 - sum(pbar[i, 1:nBins])  #no detection = 1 - p(detection)
n[i] ~ dbin(1 - pbar[i, nBins + 1], N[i])  #modeling counts from abundance
```

Note that dbin in JAGS wants the probability followed by the size for the bernoulli, whereas R wants the reverse. That always trips me up.

Next, let's tell JAGS about the probability of being in any given distance bin. I've talked about multinomial distributions before but for those unfamiliar, we have to get rid of the "no detections" bin before we can do the calculation in JAGS. To do this, we use the conditional probability for each bin, which is just the probability of detection for that bin (*pbar*) divided by the probablity of being seen at all (1- $pbar[i, nBins + 1]$). Then we can model y, the counts of sheep in each distance bin, from a multinomial with conditional probabilities.

```
y[i, ] ~ dmulti(pbar[i, 1:nBins]/(1 - pbar[i, nBins + 1]), n[i])
```

Awesome! Now let's put all that together into one code block.

```
modelstring.sheep = "
model {
lambda.intercept ~ dunif(0, 300)
beta0 <- log(lambda.intercept)
beta1 ~ dnorm(0, 0.2)
sd ~ dunif(0,30)
tau <- 1/sd^2

for(i in 1:n.sites) {
  log(lambda[i]) <- beta0 + beta1*hardwood[i]
  N[i] ~ dpois(lambda[i])         # Latent local abundance
  for(j in 1:nBins) {
    ## Trick to do integration for *line-transects*
    pbar[i,j] <- ((pnorm(b[j+1], 0, tau) - pnorm(b[j], 0, tau)) /
                  dnorm(0, 0, tau) / (b[j+1]-b[j]))*psi[j]
  }

  pbar[i,nBins+1] <- 1-sum(pbar[i,1:nBins]) #no detection = 1 - p(detection)
  n[i] ~ dbin(1-pbar[i,nBins+1], N[i]) #modeling counts from abundance
  y[,i] ~ dmulti(pbar[i,1:nBins]/(1-pbar[i,nBins+1]), n[i]) #put n sheep into bins

  D[i] <- N[i]/(2*L*width)*10^-4 #density in hectares
}

totalAbundance <- sum(N[1:n.sites])

}
"
```

Before we can send our data to JAGS we need to create the "b" vector to designate the distance breaks. We

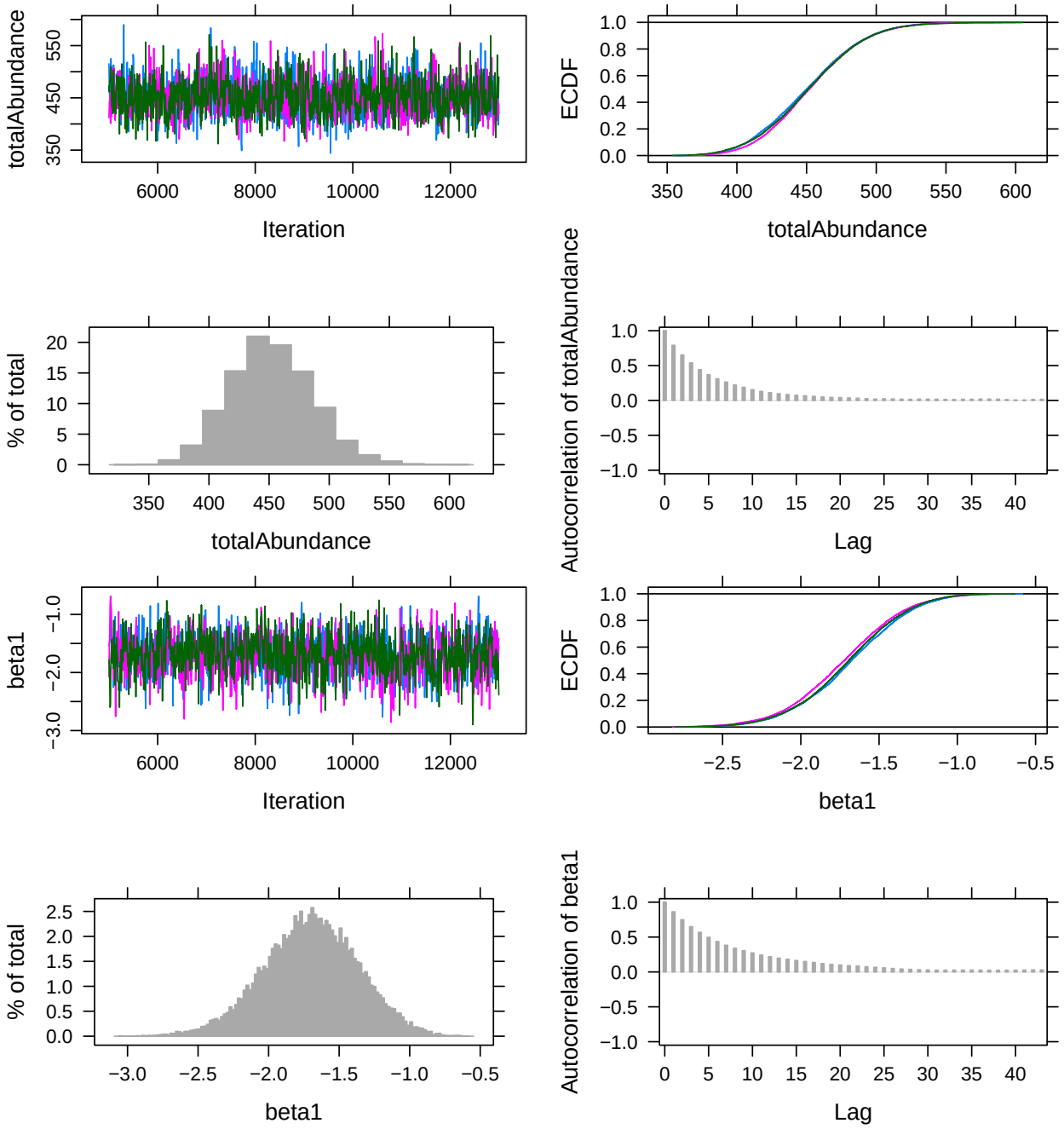
can see the relevant breaks from our data csv.

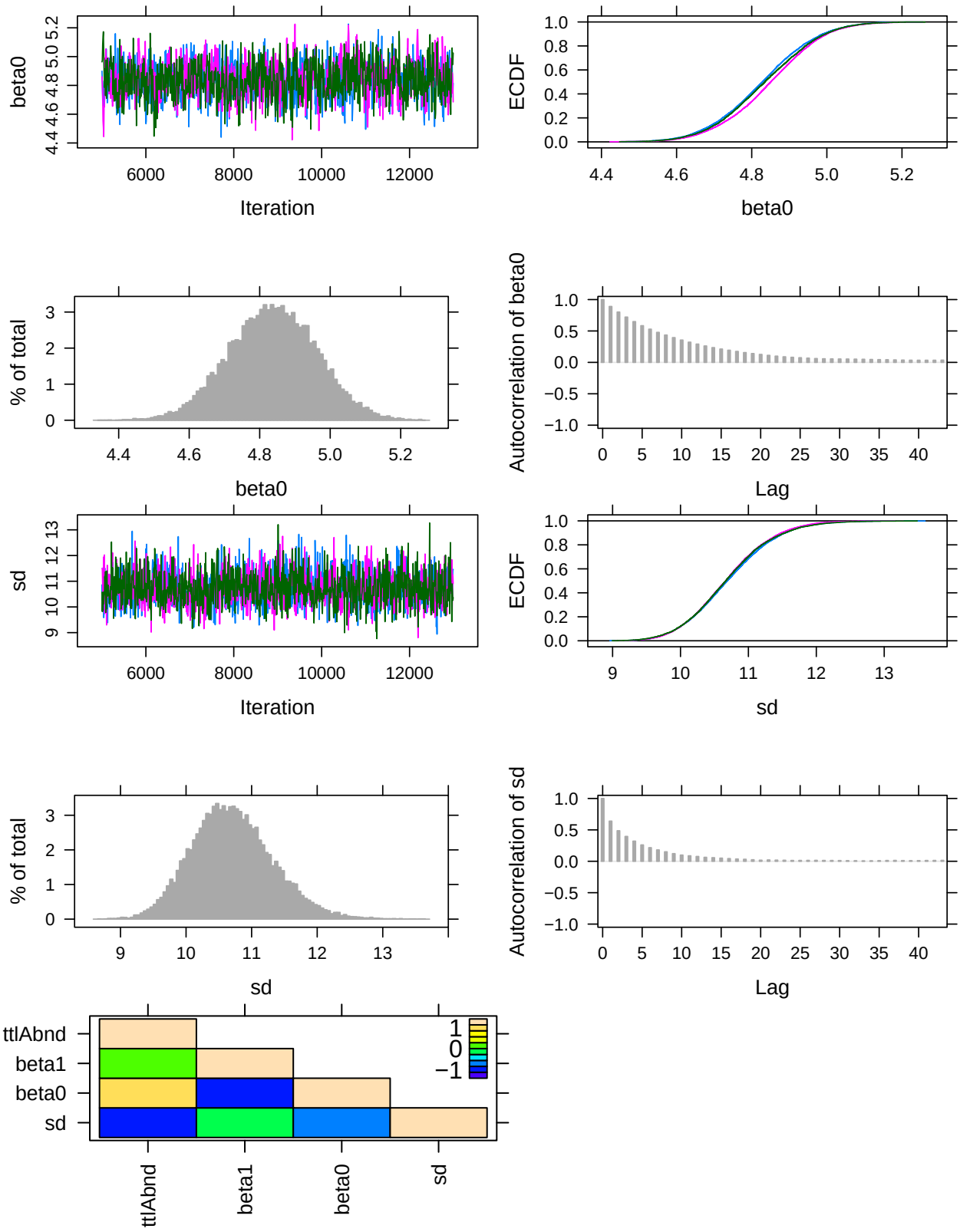
```
b <- seq(0, 30, by = 5)  #0-5, 5-10, 10-15, 15-20, 20-25, 25-30
n.Bins <- 6
```

One thing that can be annoying about running this model is choosing initial values. I like to initialize the N as very high to ensure that n is always less than or equal to N for all sites.

```
jd.sheep <- list(n.sites = 6, hardwood = Sheep_veg$Hardwood/100, y = as.matrix(Sheep[,
    2:7]), n = colSums(Sheep[, 2:7]), b = b, nBins = n.Bins, L = 5000,
    width = 30, psi = diff(b)/max(b))
ji.sheep <- function(x) {
    list(sd = 4, beta1 = 0, lambda.intercept = 100)
}
jp.sheep <- c("totalAbundance", "beta1", "beta0", "sd")
Sheep.jags <- run.jags(model = modelstring.sheep, monitor = jp.sheep, data = jd.sheep,
    n.chains = 3, inits = ji.sheep, burnin = 4000, sample = 8000, adapt = 1000,
    method = "parallel")
## Warning: You attempted to start parallel chains without setting different PRNG
## for each chain, which is not recommended. Different .RNG.name values have been
## added to each set of initial values.
## Calling 3 simulations using the parallel method...
## Following the progress of chain 1 (the program will wait for all chains
## to finish before continuing):
## Welcome to JAGS 4.3.0 on Fri Jun 25 19:53:45 2021
## JAGS is free software and comes with ABSOLUTELY NO WARRANTY
## Loading module: basemod: ok
## Loading module: bugs: ok
## . . Reading data file data.txt
## . Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 12
##    Unobserved stochastic nodes: 9
##    Total graph size: 158
## . Reading parameter file inits1.txt
## . Initializing model
## . Adapting 1000
## -------------------------------------------------| 1000
## ++++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## Adaptation successful
## . Updating 4000
## -------------------------------------------------| 4000
## ************************************************** 100%
## . . . . . Updating 8000
## -------------------------------------------------| 8000
## ************************************************** 100%
## . . . . Updating 0
## . Deleting model
## .
## All chains have finished
## Simulation complete.  Reading coda files...
## Coda files loaded successfully
## Calculating summary statistics...
```

```
## Calculating the Gelman-Rubin statistic for 4 variables....
## Finished running the simulation
plot(Sheep.jags)
## Generating plots...
```





Those plots are looking good! Now that we know they're converged we can add in the N and D variables and see what we found.

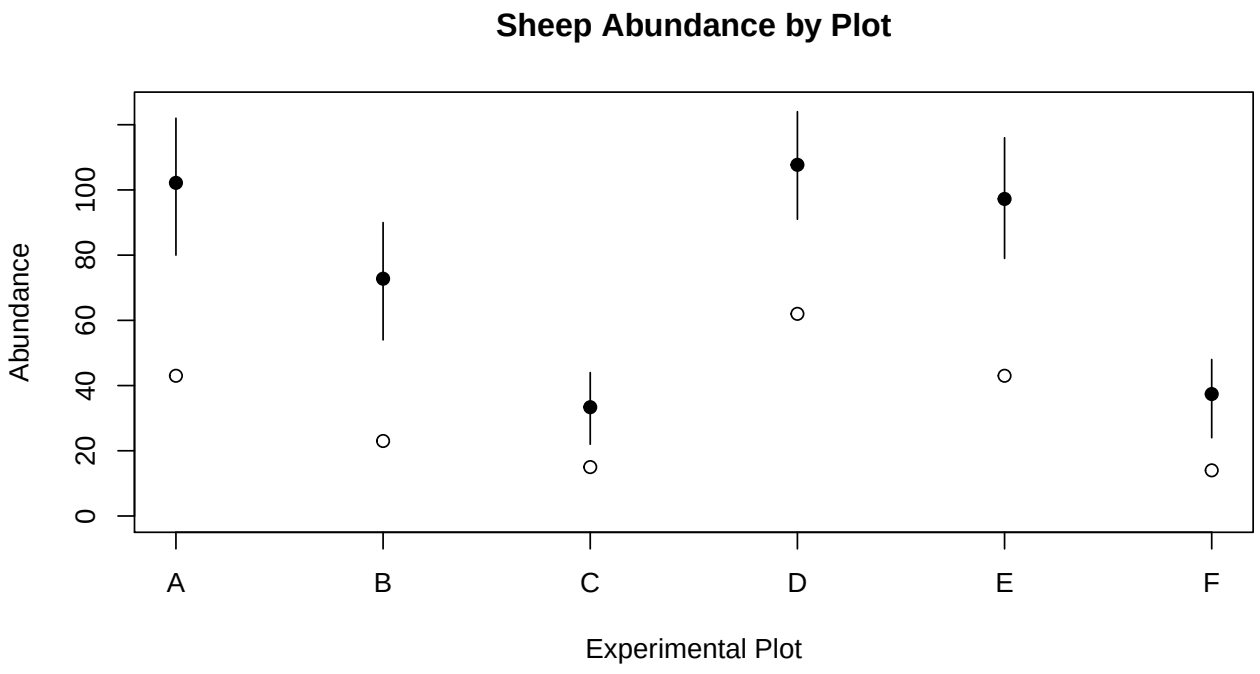
```
Sheep.jags <- extend.jags(Sheep.jags, add.monitor = c("N", "D"), sample = 2000)
## Calling 3 simulations using the parallel method...
## Following the progress of chain 1 (the program will wait for all chains
## to finish before continuing):
## Welcome to JAGS 4.3.0 on Fri Jun 25 19:53:51 2021
## JAGS is free software and comes with ABSOLUTELY NO WARRANTY
## Loading module: basemod: ok
## Loading module: bugs: ok
## . . Reading data file data.txt
## . Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 12
##    Unobserved stochastic nodes: 9
##    Total graph size: 158
## . Reading parameter file inits1.txt
## . Initializing model
## . Adapting 1000
## -------------------------------------------------| 1000
## +++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## Adaptation successful
## . NOTE: Stopping adaptation
##
## . . . . . Updating 2000
## -------------------------------------------------| 2000
## ************************************************** 100%
## . . . . Updating 0
## . Deleting model
## .
## All chains have finished
## Simulation complete.  Reading coda files...
## Coda files loaded successfully
## Calculating summary statistics...
## Calculating the Gelman-Rubin statistic for 16 variables....
## Finished running the simulation
mod.sheep <- summary(Sheep.jags)
```

```
mod.sheep
##                    Lower95        Median      Upper95          Mean
## totalAbundance  3.82000e+02  4.500000e+02  5.12000e+02  4.506522e+02
## beta1          -2.26314e+00 -1.694900e+00 -1.03655e+00 -1.690399e+00
## beta0           4.58697e+00  4.834585e+00  5.05492e+00  4.830662e+00
## sd              9.53653e+00  1.068240e+01  1.19397e+01  1.072229e+01
## N[1]            8.00000e+01  1.020000e+02  1.22000e+02  1.021698e+02
## N[2]            5.40000e+01  7.200000e+01  9.00000e+01  7.277050e+01
## N[3]            2.20000e+01  3.300000e+01  4.40000e+01  3.336500e+01
## N[4]            9.10000e+01  1.070000e+02  1.24000e+02  1.077055e+02
## N[5]            7.90000e+01  9.700000e+01  1.16000e+02  9.723967e+01
## N[6]            2.40000e+01  3.700000e+01  4.80000e+01  3.740167e+01
## D[1]            2.66667e-08  3.400000e-08  4.06667e-08  3.405661e-08
## D[2]            1.80000e-08  2.400000e-08  3.00000e-08  2.425683e-08
## D[3]            8.00000e-09  1.100000e-08  1.53333e-08  1.112167e-08
## D[4]            3.03333e-08  3.566670e-08  4.13333e-08  3.590183e-08
```

```
## D[5]            2.63333e-08  3.233330e-08  3.86667e-08  3.241322e-08
## D[6]            8.33333e-09  1.233330e-08  1.63333e-08  1.246722e-08
##                          SD Mode      MCerr MC%ofSD SSeff      AC.10     psrf
## totalAbundance 3.383536e+01  448 1.3803949     4.1   601 0.16718857 1.007314
## beta1          3.156641e-01   NA 0.0143521     4.5   484 0.22600455 1.000853
## beta0          1.203512e-01   NA 0.0062524     5.2   371 0.29091201 1.006202
## sd             6.168451e-01   NA 0.0226384     3.7   742 0.12564211 1.004205
## N[1]           1.079905e+01   98 0.4003876     3.7   727 0.14705271 1.004062
## N[2]           9.395560e+00   71 0.3134535     3.3   898 0.09339020 1.002417
## N[3]           5.718703e+00   33 0.1603214     2.8  1272 0.05184150 1.001287
## N[4]           8.690878e+00  105 0.2627801     3.0  1094 0.09621615 1.003435
## N[5]           9.942017e+00   95 0.3470495     3.5   821 0.14806858 1.006739
## N[6]           6.314056e+00   36 0.1738932     2.8  1318 0.04634046 1.001535
## D[1]           3.599682e-09   NA       Inf     Inf     0 0.14705281 1.004062
## D[2]           3.131853e-09   NA       Inf     Inf     0 0.09339042 1.002417
## D[3]           1.906234e-09   NA       Inf     Inf     0 0.05184125 1.001287
## D[4]           2.896959e-09   NA       Inf     Inf     0 0.09621596 1.003435
## D[5]           3.314005e-09   NA       Inf     Inf     0 0.14806861 1.006739
## D[6]           2.104686e-09   NA       Inf     Inf     0 0.04634078 1.001535
```

Just for fun we can quickly graph the abundance of sheep in each plot as well as the observed values.

```
plot(1:6, mod.sheep[5:10, "Mean"], pch = 19, main = "Sheep Abundance by Plot",
    xlab = "Experimental Plot", ylab = "Abundance", xaxt = "n", ylim = c(0,
        125))
axis(1, at = 1:6, labels = LETTERS[1:6])
for (i in 1:6) {
    lines(c(i, i), c(mod.sheep[4 + i, "Lower95"], mod.sheep[4 + i, "Upper95"]))
}
points(1:6, colSums(Sheep[, 2:7]))
```



Point Count Surveys - Ovenbirds

Much like our sheep example with line transects, our ovenbird model will use the N-mixture framework. In this case we are not interested in how different habitat types affect abundance, so our model for lambda will be very simple. Our points are far enough away from each other that we do not have to worry about birds appearing in multiple surveys.

Our data consists of 4 points on the same plot, so we can assume some level of homogeneity. We will assume the expected abundance is equal at all 4 points (we could add in random effects or covariates if we chose, but we will skip those for this example).

```
for (i in 1:n.pts) {
    log(lambda[i]) <- beta0
    N[i] ~ dpois(lambda[i])  #abundance at each point
}
```

Now we can move to detection, which will be very similar to the line-transect example except we are now working with a circle rather than a rectangular area.

As I mentioned at the top of this tutorial, we need to solve:

$$p = \int 2\pi x e^{\frac{-x^2}{2\sigma^2}} dx$$

where x is the distance from the center point and p is the probability of detection.

Unfortunately, we can't directly take the integral of this lovely function and we have to use approximation to get into a helpful form. To save you the trouble of working through some calculus, someone else has done all the work and come up with the following trick for integrating the probability of detection in a circle. We then adjust each distance bin by the area in the bin and the proportion of the total area that each bin takes up.

```
pbar[i, j] <- (sigma[i]^2 * (1 - exp(-b[j + 1]^2/(2 * sigma[i]^2))) - sigma[i]^2 *
    (1 - exp(-b[j]^2/(2 * sigma[i]^2)))) * 2 * 3.141593/area[j]
pi[i, j] <- psi[j] * pbar[i, j]
```

Whew, glad that's over with! Now let's allow detection to vary between points. If we look at our OVEN data we can see that Wind, Noise and Temperature were all taken at each point as possible detection covariates. Let's use noise to model variation in detection. Note that this is saying that noise is expected to decrease (or increase) the distance you can detect the ovenbird at, but still assumes 100% detection at distance 0.

We'll do a standard log-link because it's the easiest. Don't forget priors for the new terms we're adding in!

```
log(sigma[i]) <- alpha0 + alpha1 * noise[i]
alpha0 ~ dnorm(0, 0.5)
alpha1 ~ dnorm(0, 0.5)
```

Just like with our line-transect model, we can now calculate the multinomial cell probabilities and tell JAGS to connect this with our observation data.

```
pi[i, nBins + 1] <- 1 - sum(pi[i, 1:nBins])  #no detection = 1 - p(detection)
n[i] ~ dbin(1 - pi[i, nBins + 1], N[i])  #modeling counts from abundance
y[i, ] ~ dmulti(pi[i, 1:nBins]/(1 - pi[i, nBins + 1]), n[i])  #put birds into distance bins
```

Now we just need to stick some priors on our latent variables and we produce the following code:

```
modelstring.bird = "

model {
lambda.intercept ~ dunif(0, 200)
beta0 <- log(lambda.intercept)
alpha0 ~ dnorm(0, 0.25)
```

```
alpha1 ~ dnorm(0, 0.25)

for(i in 1:n.pts) {
  log(lambda[i]) <- beta0 #expected bird abund
  N[i] ~ dpois(lambda[i]) #realized bird abund

  log(sigma[i]) <- alpha0 + alpha1*noise[i] #detection covariates

  for(j in 1:nBins) {
    ## Trick to do integration for *point-transects*
    pbar[i,j] <- (sigma[i]^2 * (1-exp(-b[j+1]^2/(2*sigma[i]^2))) -
                  sigma[i]^2 * (1-exp(-b[j]^2/(2*sigma[i]^2)))) *
                 2*3.141593/area[j]
    pi[i,j] <- psi[j]*pbar[i,j]
  }

  pi[i,nBins+1] <- 1-sum(pi[i,1:nBins]) #no detection = 1 - p(detection)
  n[i] ~ dbin(1-pi[i,nBins+1], N[i]) #modeling counts from abundance
  y[i,] ~ dmulti(pi[i,1:nBins]/(1-pi[i,nBins+1]), n[i]) #connect with observed distances

}

totalAbundance <- sum(N[1:n.pts])

}
"
```

Time to clean our data up into a useful format! Once a bird is detected it's much easier to detect it again in future surveys (the 4 consecutive surveys are not independent) so we can just worry about the distance at the first detection for each individual. In a future tutorial I will touch on combining availability (time removal sampling) with distance sampling but this tutorial is already far too long (sorry!)

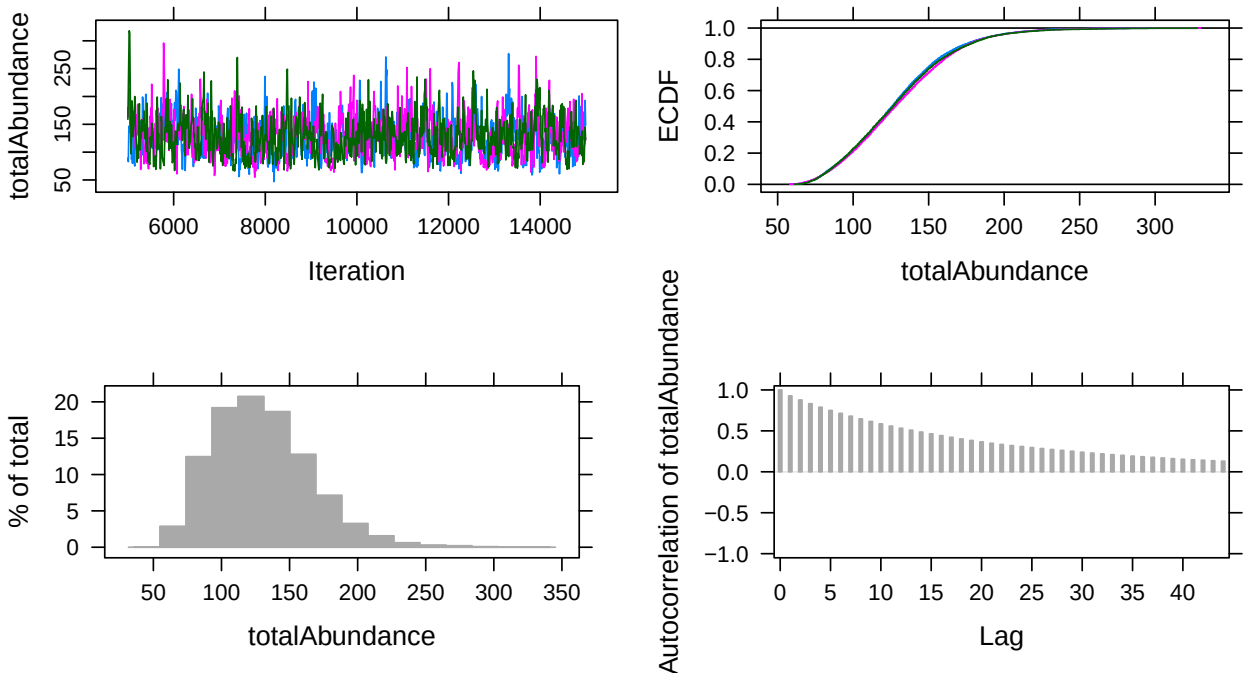
```
n.pts <- length(unique(OVEN$PointName))
noise <- c(2, 1, 0, 1)  #lazy way of assigning this
bin.dist <- 20  #0-100 by 20
bin.mids <- seq(10, 90, by = 20)  #0-20, 20-40, 40-60, 60-80, 80-100
breaks = c(-100, seq(0, 100, by = 20))
b <- seq(0, 120, by = 20)
area <- pi * b^2
psi <- (area[-1] - area[-7])/area[length(b) - 1]
# round distances into distance bins
OVEN$d1 <- as.numeric(as.character(cut(OVEN$distanceP1, breaks = breaks,
    labels = c(0, bin.mids))))
OVEN$d2 <- as.numeric(as.character(cut(OVEN$distanceP2, breaks = breaks,
    labels = c(0, bin.mids))))
OVEN$d3 <- as.numeric(as.character(cut(OVEN$distanceP3, breaks = breaks,
    labels = c(0, bin.mids))))
OVEN$d4 <- as.numeric(as.character(cut(OVEN$distanceP4, breaks = breaks,
    labels = c(0, bin.mids))))
n <- rep(NA, n.pts)
obs <- array(NA, dim = c(n.pts, length(bin.mids)))
for (i in 1:n.pts) {
    # a lengthy process to sort into bins
    sp <- OVEN[OVEN$PointName == levels(OVEN$PointName)[i], c("d1", "d2",
```

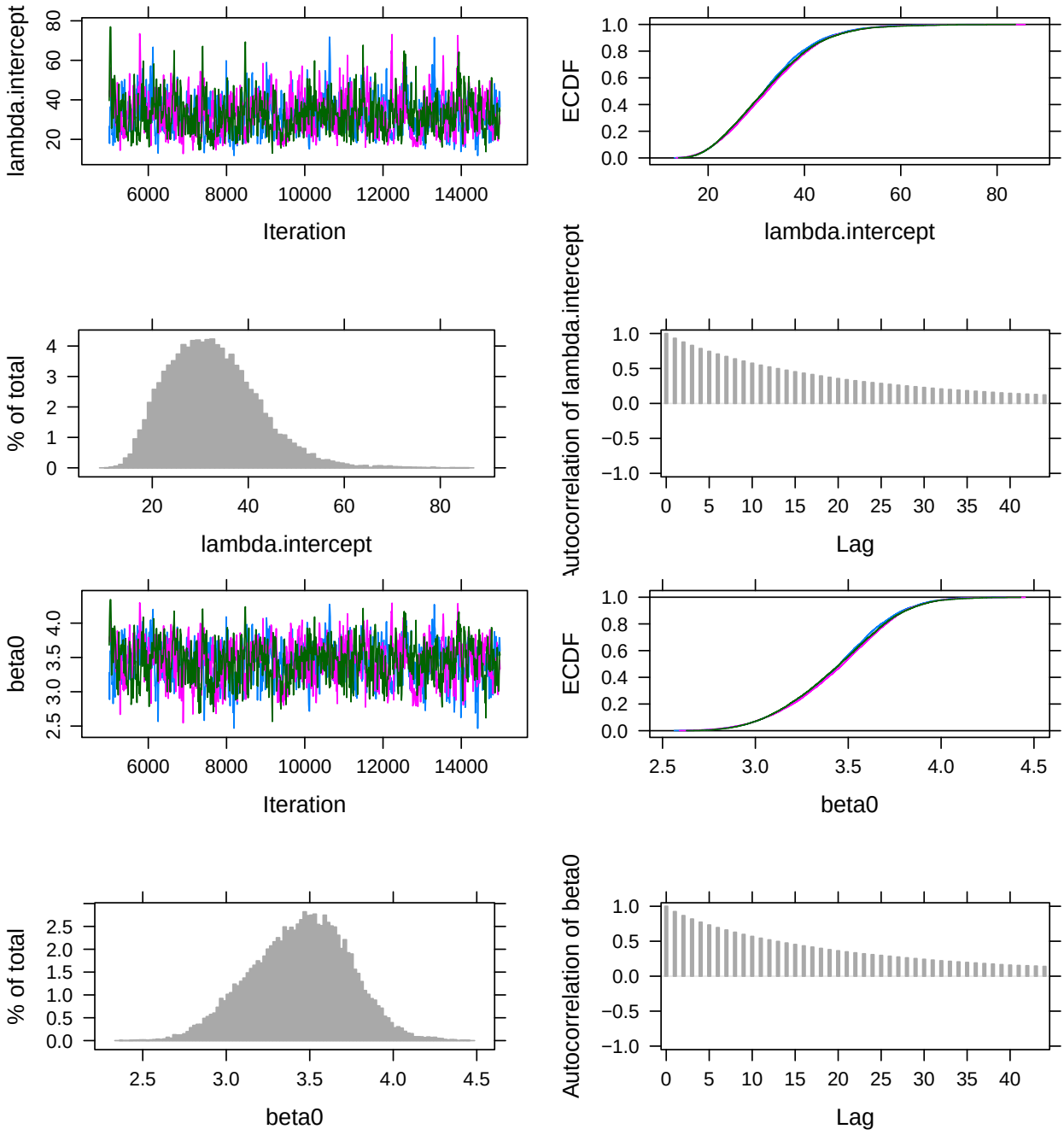
```
        "d3", "d4")]
    sp[is.na(sp)] <- -50  #make sure NA's don't mess up the n's
    n[i] <- sum(rowSums(sp[1:4] > 0) > 0)  #how many detected per point
    tt <- matrix(0, nrow = nrow(sp), ncol = 4)  #time of detection
    for (l in 1:nrow(sp)) {
        if (sum(sp[l, ] > 0) > 0) {
            tt[l, min(which(sp[l, ] > 0))] <- 1  #tells us when it was first detected
        }
    }
    obs[i, ] <- tabulate((sp * tt)[sp * tt > 0], nbins = 95)[bin.mids]
}
```

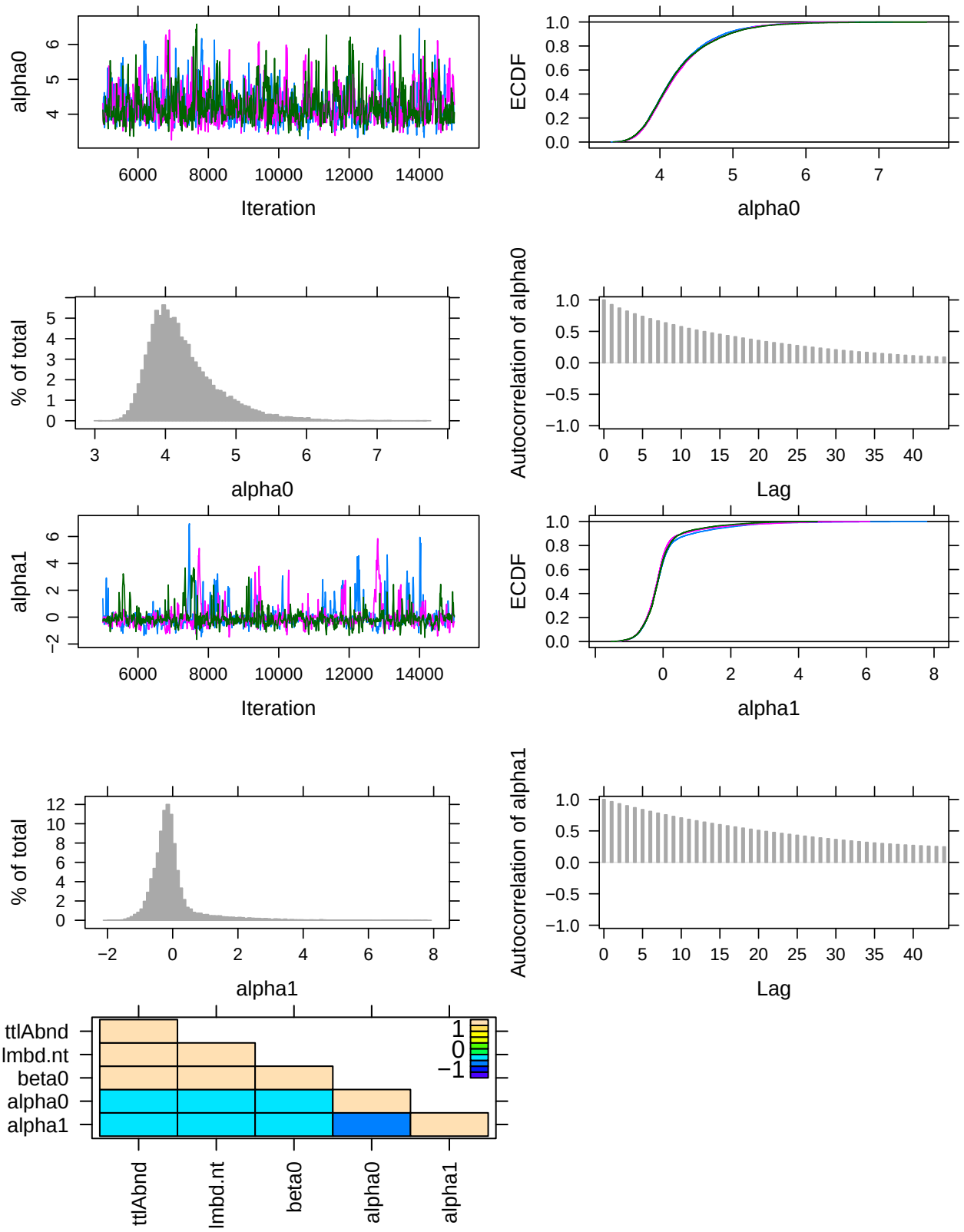
This ugly piece of code above is just rounding distances into bins and collecting the first distance bin each bird was detected in. But it does what we need it to do and now we can run the model in JAGS! I like to initialize detection as a very low value and abundance as a very high value to avoid issues with starting the MCMC chains.

```
jd.birds <- list(noise = noise, y = obs, n = n, b = b, n.pts = n.pts, psi = psi,
    nBins = 5, area = area[-1])
ji.birds <- function(x) {
    list(alpha1 = 0, alpha0 = 18, lambda.intercept = 50)
}
jp.birds <- c("totalAbundance", "lambda.intercept", "beta0", "alpha0",
    "alpha1")
birds.jags <- run.jags(model = modelstring.bird, monitor = jp.birds, data = jd.birds,
    n.chains = 3, inits = ji.birds, burnin = 4000, sample = 10000, adapt = 1000,
    method = "parallel")
## Warning: You attempted to start parallel chains without setting different PRNG
## for each chain, which is not recommended. Different .RNG.name values have been
## added to each set of initial values.
## Calling 3 simulations using the parallel method...
## Following the progress of chain 1 (the program will wait for all chains
## to finish before continuing):
## Welcome to JAGS 4.3.0 on Fri Jun 25 19:53:53 2021
## JAGS is free software and comes with ABSOLUTELY NO WARRANTY
## Loading module: basemod: ok
## Loading module: bugs: ok
## . . Reading data file data.txt
## . Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 8
##    Unobserved stochastic nodes: 7
##    Total graph size: 229
## . Reading parameter file inits1.txt
## . Initializing model
## . Adapting 1000
## -------------------------------------------------| 1000
## +++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## Adaptation successful
## . Updating 4000
## -------------------------------------------------| 4000
## ************************************************** 100%
```

```
## . . . . . Updating 10000
## -------------------------------------------------| 10000
## ************************************************** 100%
## . . . . Updating 0
## . Deleting model
## .
## All chains have finished
## Simulation complete.  Reading coda files...
## Coda files loaded successfully
## Calculating summary statistics...
## Calculating the Gelman-Rubin statistic for 5 variables....
## Finished running the simulation
plot(birds.jags)
## Generating plots...
```




lambda.intercept
Iteration
ECDF
lambda.intercept
% of total
lambda.intercept
Autocorrelation of lambda.intercept
Lag
beta0
Iteration
ECDF
beta0
% of total
beta0
Autocorrelation of beta0
Lag

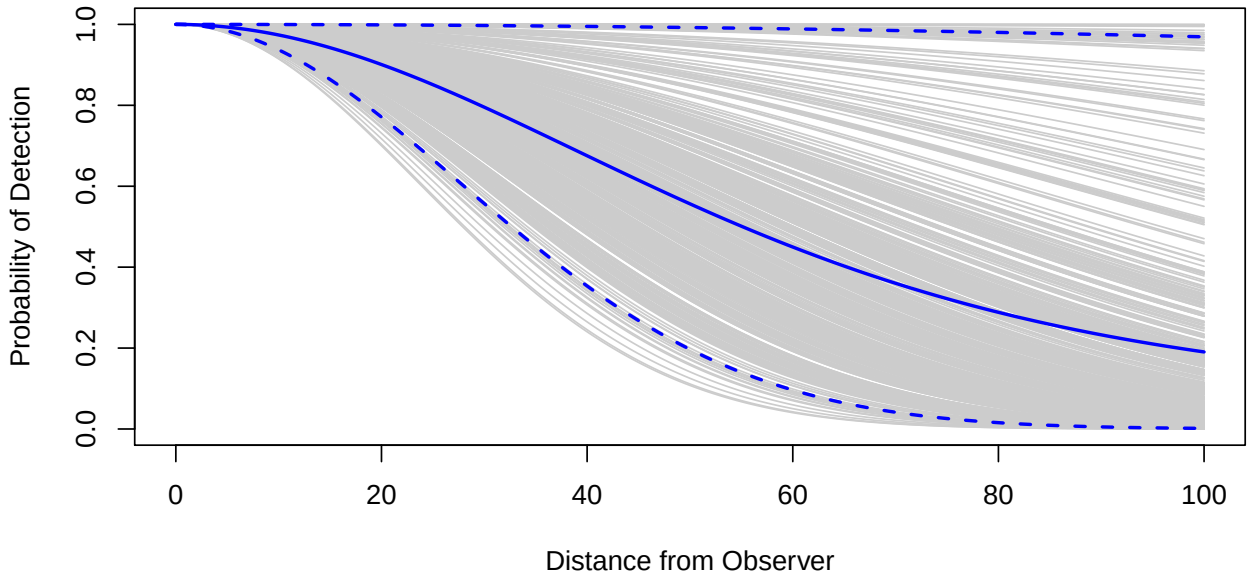


We can tell from our output graphs that we have a lot of uncertainty about the total abundance at the site. This makes sense, since our data set is pretty small and we don't have a lot of variation in our one detection covariate. If we want more precision in our output, we will likely need additional data or need to add in

time-removal sampling. Also notice that lambda.intercept, beta0 and total abundance are correlated - this makes total sense given what these variables are so we don't need to panic. But it's always a good idea to check the correlation plot before accepting your model output as reasonable!

Let's plot the estimated relationship between distance and detection probability when noise is a score of 2.

```
xs <- seq(0, 100, by = 0.5)
# grab the last 500 iterations of the mcmc chain
alpha0 <- birds.jags$mcmc[[1]][(nrow(birds.jags$mcmc[[1]]) - 499):nrow(birds.jags$mcmc[[1]]),
    "alpha0"]
alpha1 <- birds.jags$mcmc[[1]][(nrow(birds.jags$mcmc[[1]]) - 499):nrow(birds.jags$mcmc[[1]]),
    "alpha1"]
p <- array(NA, dim = c(500, length(xs)))
plot(c(0, 100), c(0, 1), col = "white", xlab = "Distance from Observer",
    ylab = "Probability of Detection")
for (i in 1:500) {
    p[i, ] <- exp(-(xs^2)/(2 * exp(alpha0[i] + alpha1[i] * 2)^2))
    lines(xs, p[i, ], col = "grey80")
}
post.mean <- colMeans(p)
post.lower <- apply(p, 2, quantile, prob = 0.025)
post.upper <- apply(p, 2, quantile, prob = 0.975)
lines(xs, post.mean, lty = 1, lwd = 2, col = "blue")
lines(xs, post.lower, lty = 2, lwd = 2, col = "blue")
lines(xs, post.upper, lty = 2, lwd = 2, col = "blue")
```



Looks like we can detect ovenbirds from quite a distance. This graph also really highlights the need for more data- look at how wide the CI is for detection probability!

Code in NIMBLE

Alright, time to do all that again for NIMBLE! Luckily it's almost exactly the same.

Gopher Tortoise Hierarchical Distance Model

As it turns out, nothing changes in the tortoise model when we send it to NIMBLE except that we save it as a NIMBLE model instead of a character string AND that we need to tell it how large w is. Very easy!

```
library(nimble)
nimbletorts <- nimbleCode({
    for (i in 1:n.torts) {
        w[i] ~ dbern(psi)  #real?
        x[i] ~ dunif(0, 50)  #distance
        p[i] <- exp(-x[i]^2/(2 * sig^2))  #p detection
        y[i] ~ dbern(p[i] * w[i])  #detected?
    }
    sig ~ dunif(5, 30)
    psi ~ dunif(0, 1)

    N <- sum(w[1:n.torts])  #total torts; the only meaningful change
    D <- N/114.227  #density in torts/hectare
})
```

Next we assign all our constants, data (things from distributions!), parameters and initial values.

```
newburrows <- data.frame(Distance = c(burrows$Distance, rep(NA, 300)),
    w = c(rep(1, nrow(burrows)), rep(NA, 300)), y = c(rep(1, nrow(burrows)),
        rep(0, 300)))
nc.torts <- list(n.torts = nrow(newburrows))  #constants
nd.torts <- list(x = newburrows$Distance, w = newburrows$w, y = newburrows$y)  #data
ni.torts <- list(sig = runif(1, 10, 20), psi = runif(1))
params.torts <- c("N", "D", "sig", "psi")
```

As always I recommend trying to run the code below with just one chain first and fixing issues before running it in parallel. Luckily I know my own code works :)

```
library(parallel)
cl <- makeCluster(3)
clusterExport(cl = cl, varlist = c("nc.torts", "nd.torts", "ni.torts",
    "params.torts", "nimbletorts"))
torts.out <- clusterEvalQ(cl = cl, {
    library(nimble)
    library(coda)
    preptorts <- nimbleModel(code = nimbletorts, constants = nc.torts,
        data = nd.torts, inits = ni.torts)
    preptorts$initializeInfo()
    mcmctorts <- configureMCMC(preptorts, monitors = params.torts, print = T)
    tortsMCMC <- buildMCMC(mcmctorts)  #actually build the code for those samplers
    Cmodel <- compileNimble(preptorts)  #compiling the model itself in C++;
    Comptorts <- compileNimble(tortsMCMC, project = preptorts)  # compile the samplers next
    Comptorts$run(niter = 30000, nburnin = 10000, thin = 1)  #if you run this in your console it will s
    # But it's doing something.
    return(as.mcmc(as.matrix(Comptorts$mvSamples)))
})
```

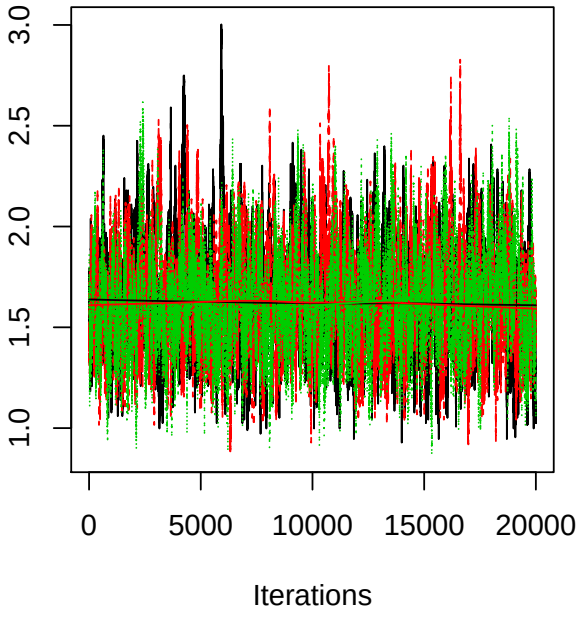
Let's see what we got!

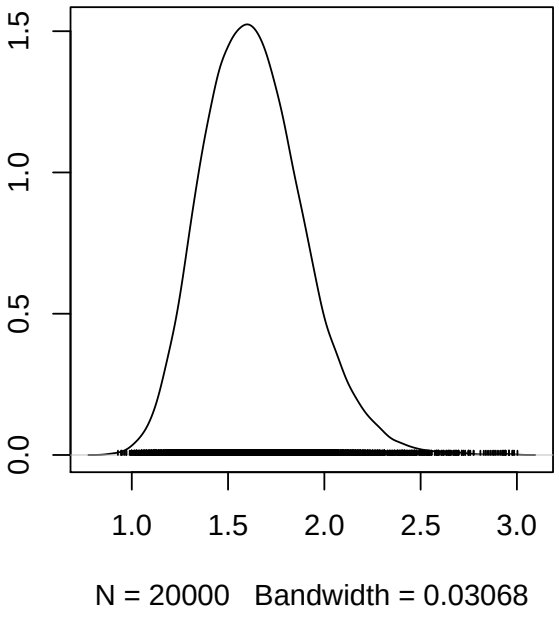
```
library(coda)
torts.mod.nimble <- mcmc.list(torts.out)
gelman.diag(torts.mod.nimble, multivariate = F)
## Potential scale reduction factors:
##
##     Point est. Upper C.I.
## D         1.01       1.02
```

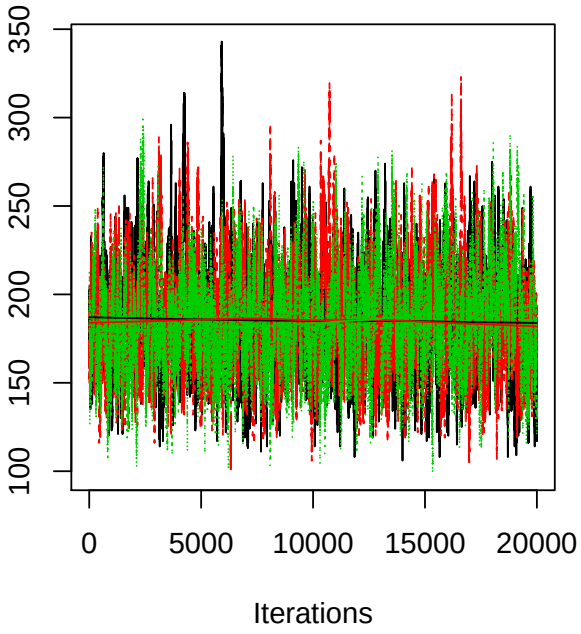
```
## N         1.01       1.02
## psi       1.01       1.02
## sig       1.00       1.00
```

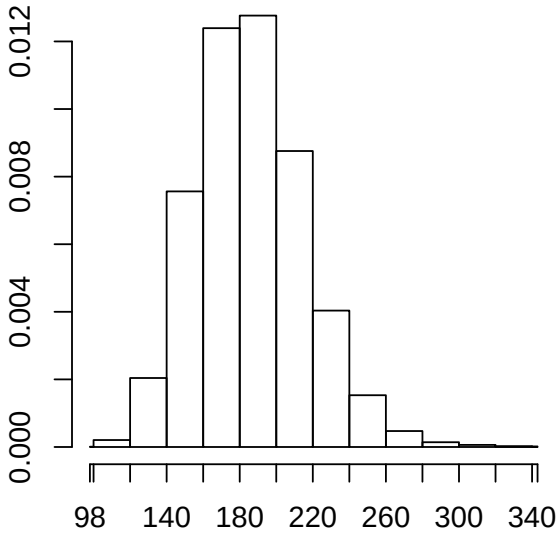
Time to plot.

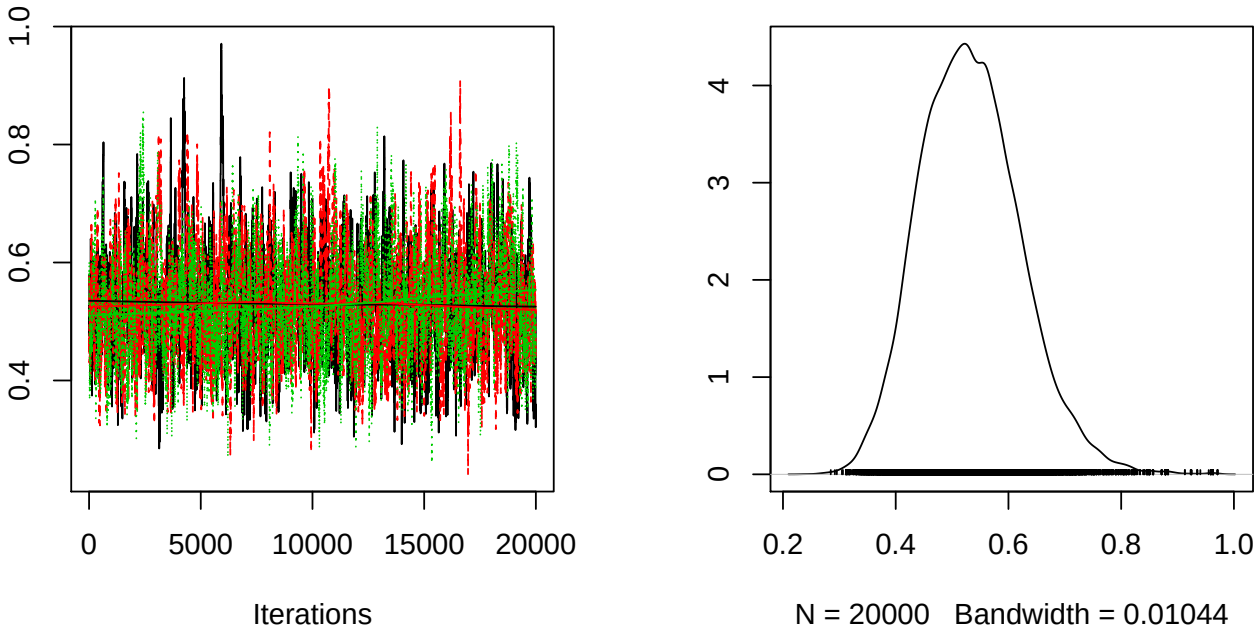
```
plot(torts.mod.nimble)
```











Looks like it converged! Don't forget to close your cluster when you're done.

```
stopCluster(cl)
```

We can see that the results are the same as from JAGS.

```
summary.torts <- summary(torts.mod.nimble)
summary.torts$quantiles
##             2.5%         25%         50%         75%       97.5%
## D     1.1818572   1.4444921   1.6195821   1.7946720   2.1973789
## N   135.0000000 165.0000000 185.0000000 205.0000000 251.0000000
## psi   0.3762761   0.4690965   0.5281734   0.5889213   0.7208266
## sig   9.0186950  10.1442641  10.8564469  11.6484062  13.4129122
```

```
### JAGS results for comparison:
mod.Torts
##        Lower95    Median    Upper95        Mean          SD Mode       MCerr
## N   131.000000 184.00000 241.000000 185.7228333 28.46153716  184 0.899127657
## D     1.146840   1.61083   2.109830   1.6259102  0.24916639   NA 0.007871407
## sig   8.909360  10.88440  13.251400  10.9760207  1.11994136   NA 0.025552547
## psi   0.371251   0.52643   0.702567   0.5301958  0.08483442   NA 0.002666823
##     MC%ofSD SSeff     AC.10     psrf
## N       3.2  1002 0.4424071 1.002520
## D       3.2  1002 0.4424072 1.002520
## sig     2.3  1921 0.2130615 1.001051
## psi     3.1  1012 0.4313619 1.003421
```

Sheep Line Transects

The sheep model also very easily translates into NIMBLE. The biggest thing we need to change is this line right here:

```
y[, i] ~ dmulti(pbar[i, 1:nBins]/(1 - pbar[i, nBins + 1]), n[i])
```

In NIMBLE, you can't calculate multinomial cell probability inside the multinomial. To avoid this problem we just need to make an object with the probabilities first. Note that we don't need to tell it how large the y

vector is - since this is provided as data it already knows what size to expect.

```
probs[i, 1:nBins] <- pbar[i, 1:nBins]/(1 - pbar[i, nBins + 1])
y[, i] ~ dmulti(probs[i, 1:nBins], n[i])
```

We can also move from variance to standard deviation if we want. Here's our new model:

```
library(nimble)
nimblesheep <- nimbleCode({
    lambda.intercept ~ dunif(0, 300)
    beta0 <- log(lambda.intercept)
    beta1 ~ dnorm(0, sd = 2.2)
    sd ~ dunif(0, 30)

    for (i in 1:n.sites) {
        log(lambda[i]) <- beta0 + beta1 * hardwood[i]
        N[i] ~ dpois(lambda[i])  # Latent local abundance
        for (j in 1:nBins) {
            ## Trick to do integration for *line-transects*
            pbar[i, j] <- ((pnorm(b[j + 1], 0, sd = sd) - pnorm(b[j], 0,
                sd = sd))/dnorm(0, 0, sd = sd)/(b[j + 1] - b[j])) * psi[j]
        }

        pbar[i, nBins + 1] <- 1 - sum(pbar[i, 1:nBins])  #no detection = 1 - p(detection)
        n[i] ~ dbin(1 - pbar[i, nBins + 1], N[i])  #modeling counts from abundance
        probs[i, 1:nBins] <- pbar[i, 1:nBins]/(1 - pbar[i, nBins + 1])
        y[, i] ~ dmulti(probs[i, 1:nBins], n[i])

        D[i] <- N[i]/(2 * L * width) * 10^-4  #density in hectares
    }

    totalAbundance <- sum(N[1:n.sites])

})
```

Alright, let's send this guy to NIMBLE.

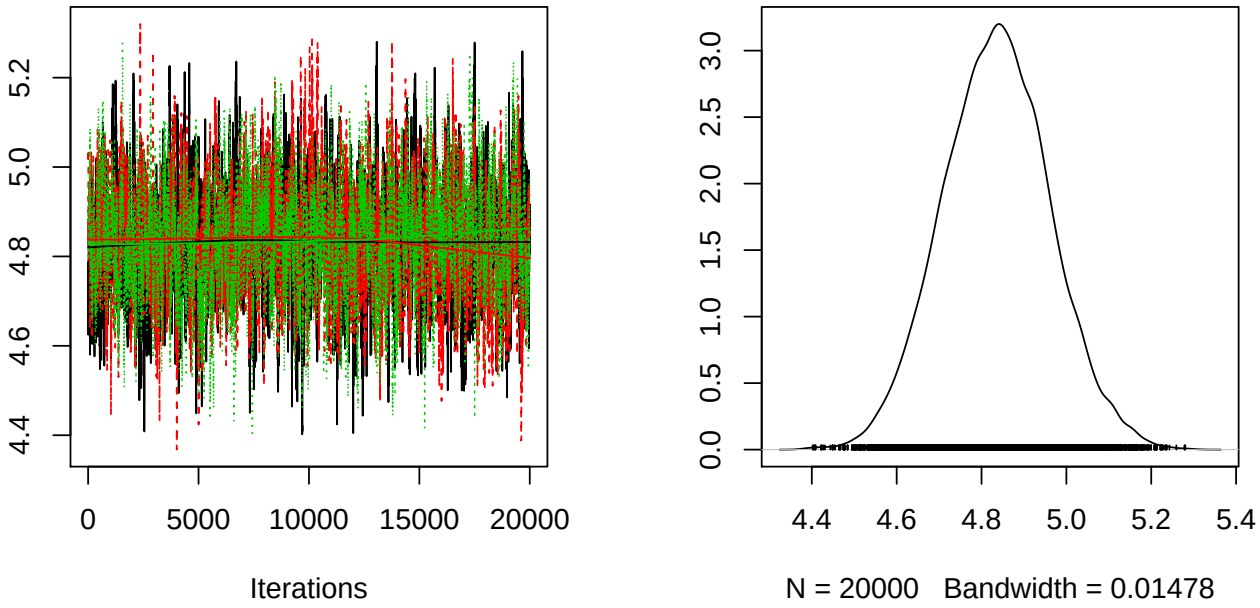
```
b <- seq(0, 30, by = 5)  #0-5, 5-10, 10-15, 15-20, 20-25, 25-30
nBins <- 6
nc.sheep <- list(n.sites = 6, hardwood = Sheep_veg$Hardwood/100, b = b,
    nBins = nBins, L = 5000, width = 30, psi = diff(b)/max(b))  #constants
nd.sheep <- list(y = as.matrix(Sheep[, 2:7]), n = colSums(Sheep[, 2:7]))  #data
ni.sheep <- list(sd = 4, beta1 = 0, lambda.intercept = 100)
params.sheep <- c("totalAbundance", "beta1", "beta0", "sd")
```

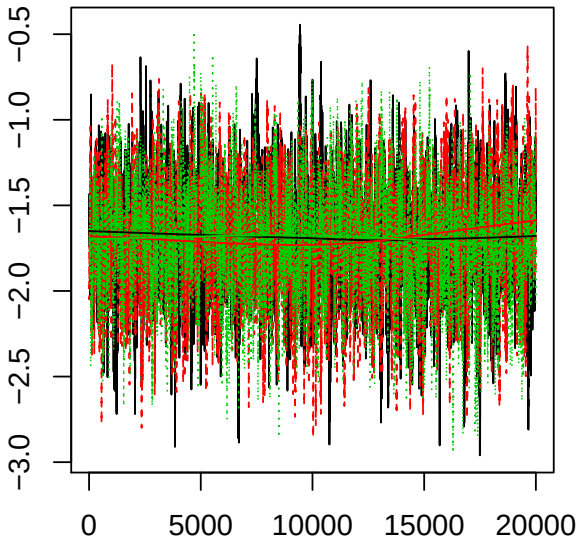
```
library(parallel)
cl <- makeCluster(3)
clusterExport(cl = cl, varlist = c("nc.sheep", "nd.sheep", "ni.sheep",
    "params.sheep", "nimblesheep"))
sheep.out <- clusterEvalQ(cl = cl, {
    library(nimble)
    library(coda)
    prepsheep <- nimbleModel(code = nimblesheep, constants = nc.sheep,
        data = nd.sheep, inits = ni.sheep)
    prepsheep$initializeInfo()
    mcmcsheep <- configureMCMC(prepsheep, monitors = params.sheep, print = T)
```

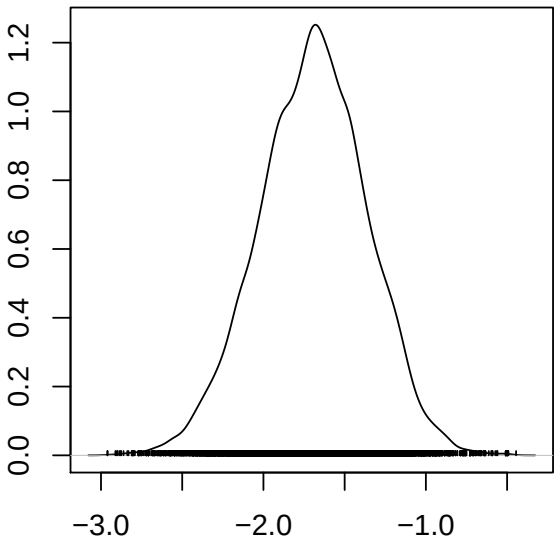
```
    sheepMCMC <- buildMCMC(mcmcsheep)  #actually build the code for those samplers
    Cmodel <- compileNimble(prepsheep)  #compiling the model itself in C++;
    Compsheep <- compileNimble(sheepMCMC, project = prepsheep)  # compile the samplers next
    Compsheep$run(niter = 30000, nburnin = 10000, thin = 1)  #if you run this in your console it will s
    # But it's doing something.
    return(as.mcmc(as.matrix(Compsheep$mvSamples)))
})
```

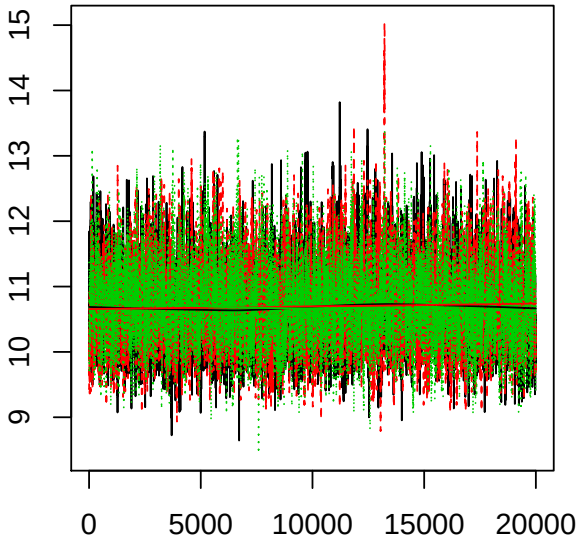
```
library(coda)
sheep.mod.nimble <- mcmc.list(sheep.out)
stopCluster(cl)
gelman.diag(sheep.mod.nimble, multivariate = F)
## Potential scale reduction factors:
##
##                Point est. Upper C.I.
## beta0                   1       1.01
## beta1                   1       1.00
## sd                      1       1.01
## totalAbundance          1       1.01
```

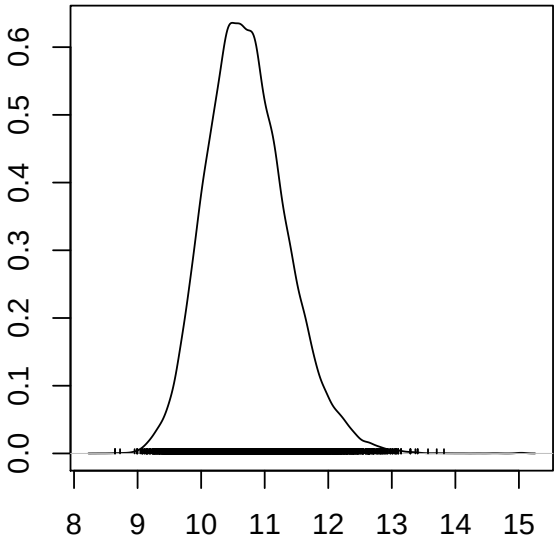
```
plot(sheep.mod.nimble)
```

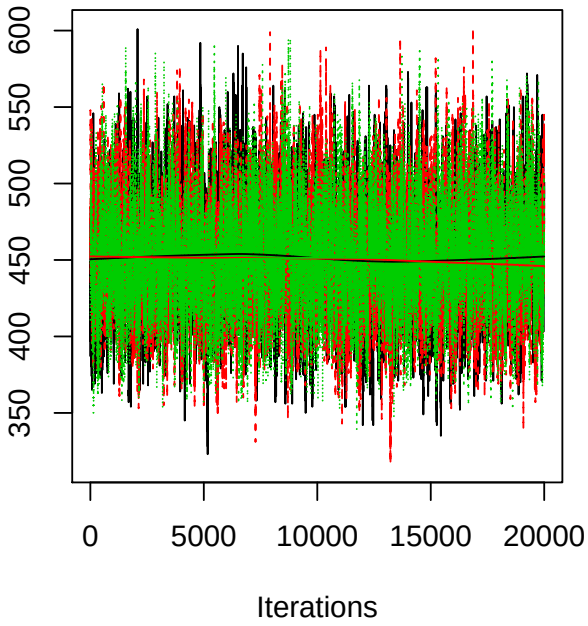


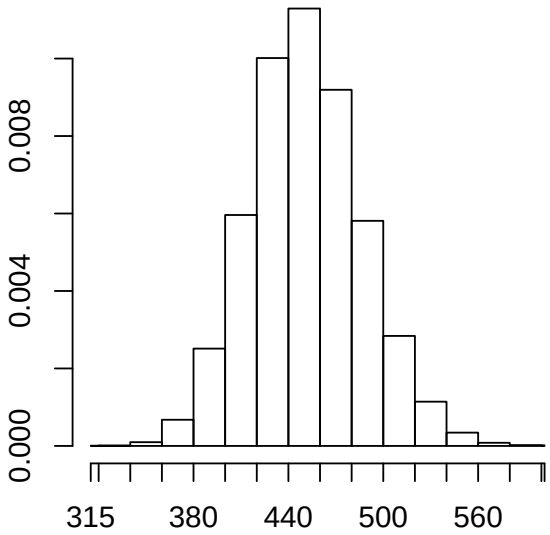

−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
0
5000
10000
15000
20000
Iterations


1.2
1.0
0.8
0.6
0.4
0.2
0.0
−3.0
−2.0
−1.0
N = 20000 Bandwidth = 0.03914


15
14
13
12
11
10
9
0
5000
10000
15000
20000
Iterations


0.6
0.5
0.4
0.3
0.2
0.1
0.0
8
9
10
11
12
13
14
15
N = 20000 Bandwidth = 0.07359





Yay! Looking good.

```
summary(sheep.mod.nimble)
##
## Iterations = 1:20000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 20000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##                     Mean      SD Naive SE Time-series SE
## beta0              4.835  0.1259 0.000514       0.003501
## beta1             -1.695  0.3334 0.001361       0.008230
## sd                10.720  0.6362 0.002597       0.010681
## totalAbundance 451.898 35.1617 0.143547       0.637789
##
## 2. Quantiles for each variable:
##
##                    2.5%     25%     50%     75%   97.5%
## beta0             4.590   4.749   4.836   4.920   5.087
## beta1            -2.365  -1.917  -1.688  -1.469  -1.063
## sd                9.613  10.276  10.677  11.116  12.103
## totalAbundance 386.000 428.000 451.000 475.000 525.000
```

Point Count Ovenbird Surveys

Finally, we can move to our point count example. It's also very straightforward to move to NIMBLE. Once again, we need to specify the probabilities as a separate object. Otherwise we're good to go!

```
library(nimble)
nimblebirds <- nimbleCode({
    lambda.intercept ~ dunif(0, 100)
    beta0 <- log(lambda.intercept)
```

```
    alpha0 ~ dnorm(0, 0.25)
    alpha1 ~ dnorm(0, 0.25)

    for (i in 1:n.pts) {
        log(lambda[i]) <- beta0  #expected bird abund
        N[i] ~ dpois(lambda[i])  #realized bird abund

        log(sigma[i]) <- alpha0 + alpha1 * noise[i]  #detection covariates

        for (j in 1:nBins) {
            ## Trick to do integration for *point-transects*
            pbar[i, j] <- (sigma[i]^2 * (1 - exp(-b[j + 1]^2/(2 * sigma[i]^2))) -
                sigma[i]^2 * (1 - exp(-b[j]^2/(2 * sigma[i]^2)))) * 2 *
                3.141593/area[j]
            pi[i, j] <- psi[j] * pbar[i, j]
        }

        pi[i, nBins + 1] <- 1 - sum(pi[i, 1:nBins])  #no detection = 1 - p(detection)
        n[i] ~ dbin(1 - pi[i, nBins + 1], N[i])  #modeling counts from abundance
        probs[i, 1:nBins] <- pi[i, 1:nBins]/(1 - pi[i, nBins + 1])
        y[i, ] ~ dmulti(probs[i, 1:nBins], n[i])  #connect with observed distances

    }

    totalAbundance <- sum(N[1:n.pts])

})
```

If we hadn't already run the JAGS version we would need to reformat the data:

```
n.pts <- length(unique(OVEN$PointName))
noise <- c(2, 1, 0, 1)  #lazy way of assigning this
bin.dist <- 20  #0-100 by 20
bin.mids <- seq(10, 90, by = 20)  #0-20, 20-40, 40-60, 60-80, 80-100
breaks = c(-100, seq(0, 100, by = 20))
b <- seq(0, 120, by = 20)
area <- pi * b^2
psi <- (area[-1] - area[-7])/area[length(b) - 1]
# round distances into distance bins
OVEN$d1 <- as.numeric(as.character(cut(OVEN$distanceP1, breaks = breaks,
    labels = c(0, bin.mids))))
OVEN$d2 <- as.numeric(as.character(cut(OVEN$distanceP2, breaks = breaks,
    labels = c(0, bin.mids))))
OVEN$d3 <- as.numeric(as.character(cut(OVEN$distanceP3, breaks = breaks,
    labels = c(0, bin.mids))))
OVEN$d4 <- as.numeric(as.character(cut(OVEN$distanceP4, breaks = breaks,
    labels = c(0, bin.mids))))
n <- rep(NA, n.pts)
obs <- array(NA, dim = c(n.pts, length(bin.mids)))
for (i in 1:n.pts) {
    # a lengthy process to sort into bins
    sp <- OVEN[OVEN$PointName == levels(OVEN$PointName)[i], c("d1", "d2",
        "d3", "d4")]
    sp[is.na(sp)] <- -50  #make sure NA's don't mess up the n's
```

```
    n[i] <- sum(rowSums(sp[1:4] > 0) > 0)  #how many detected per point
    tt <- matrix(0, nrow = nrow(sp), ncol = 4)  #time of detection
    for (l in 1:nrow(sp)) {
        if (sum(sp[l, ] > 0) > 0) {
            tt[l, min(which(sp[l, ] > 0))] <- 1  #tells us when it was first detected
        }
    }
    obs[i, ] <- tabulate((sp * tt)[sp * tt > 0], nbins = 95)[bin.mids]
}
```

```
nc.birds <- list(noise = noise, b = b, n.pts = n.pts, psi = psi, nBins = 5,
    area = area[-1])  #constants
nd.birds <- list(y = obs, n = n)  #data
ni.birds <- list(alpha1 = 0, alpha0 = 18, lambda.intercept = 50)
params.birds <- c("totalAbundance", "lambda.intercept", "beta0", "alpha0",
    "alpha1")
```

Annnnd running time!

We can check if the model is ready to run in parallel before we commit to a full run:

```
prepbirds <- nimbleModel(code = nimblebirds, constants = nc.birds, data = nd.birds,
    inits = ni.birds)
## defining model...
## building model...
## setting data and initial values...
## running calculate on model (any error reports that follow may simply reflect missing values in model
## checking model sizes and dimensions... This model is not fully initialized. This is not an error. To
## model building finished.
prepbirds$initializeInfo()
## Missing values (NAs) or non-finite values were found in model variables: N, totalAbundance. This is
```

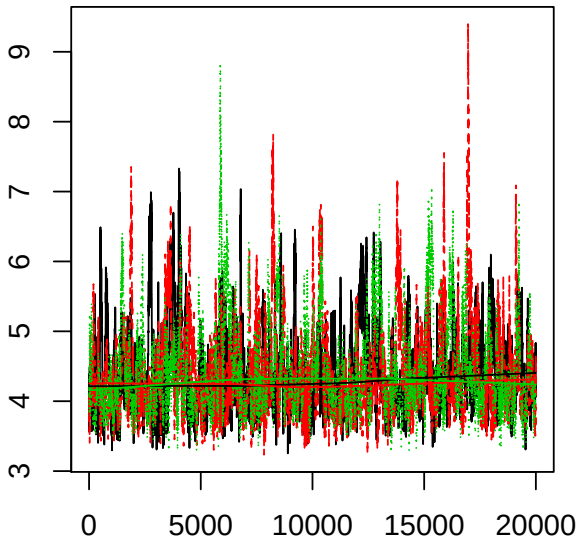
Okay, time to do a full parallel run.

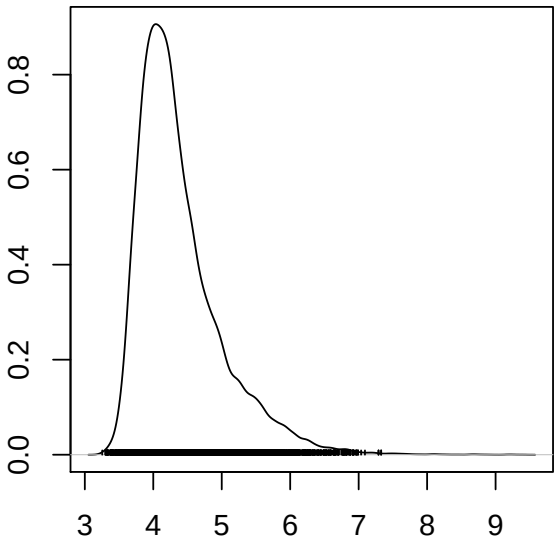
```
library(parallel)
cl <- makeCluster(3)
clusterExport(cl = cl, varlist = c("nc.birds", "nd.birds", "ni.birds",
    "params.birds", "nimblebirds"))
birds.out <- clusterEvalQ(cl = cl, {
    library(nimble)
    library(coda)
    prepbirds <- nimbleModel(code = nimblebirds, constants = nc.birds,
        data = nd.birds, inits = ni.birds)
    prepbirds$initializeInfo()
    mcmcbirds <- configureMCMC(prepbirds, monitors = params.birds, print = T)
    birdsMCMC <- buildMCMC(mcmcbirds)  #actually build the code for those samplers
    Cmodel <- compileNimble(prepbirds)  #compiling the model itself in C++;
    Compbirds <- compileNimble(birdsMCMC, project = prepbirds)  # compile the samplers next
    Compbirds$run(niter = 30000, nburnin = 10000, thin = 1)  #if you run this in your console it will s
    # But it's doing something.
    return(as.mcmc(as.matrix(Compbirds$mvSamples)))
})
```

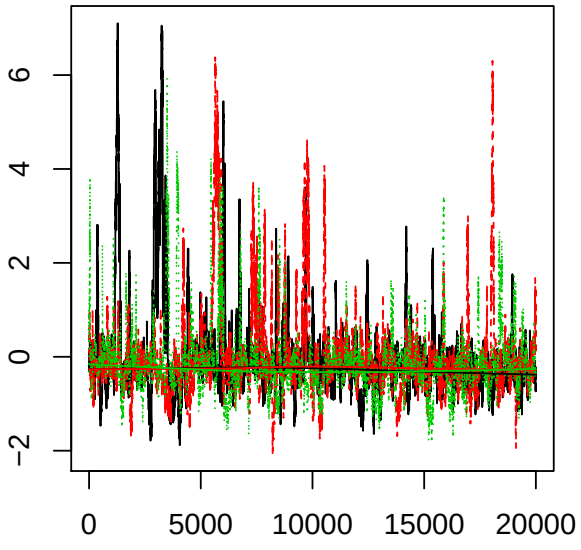
Let's see what we got.

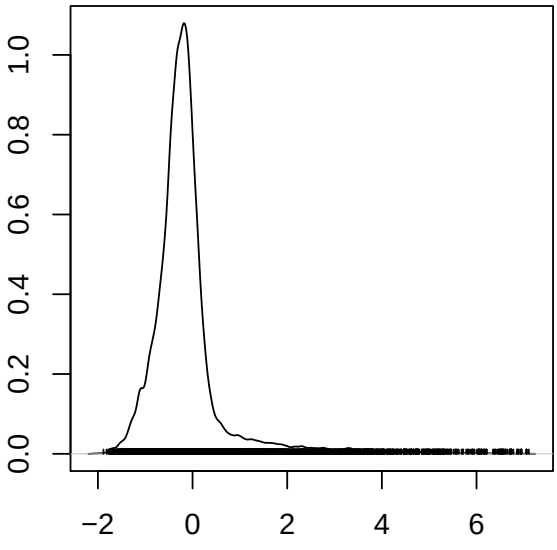
```
library(coda)
birds.mod.nimble <- mcmc.list(birds.out)
stopCluster(cl)
gelman.diag(birds.mod.nimble, multivariate = F)
## Potential scale reduction factors:
##
##                   Point est. Upper C.I.
## alpha0                  1.01       1.02
## alpha1                  1.04       1.07
## beta0                   1.00       1.01
## lambda.intercept        1.00       1.00
## totalAbundance          1.00       1.00
```

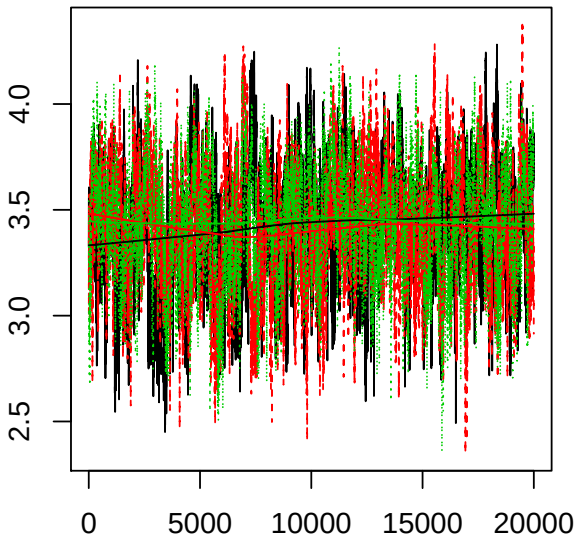
```
plot(birds.mod.nimble)
```

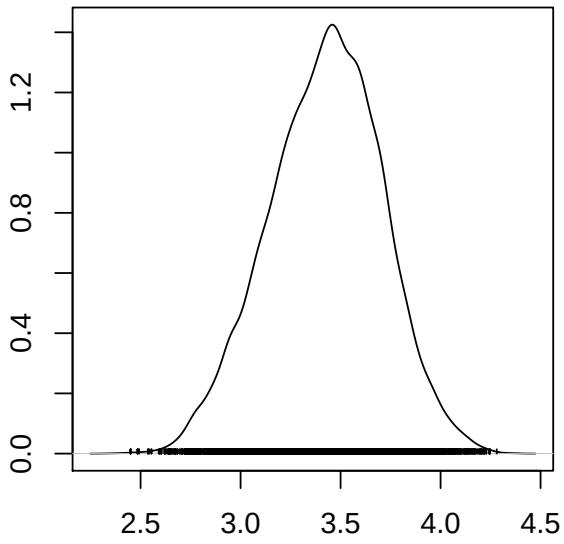


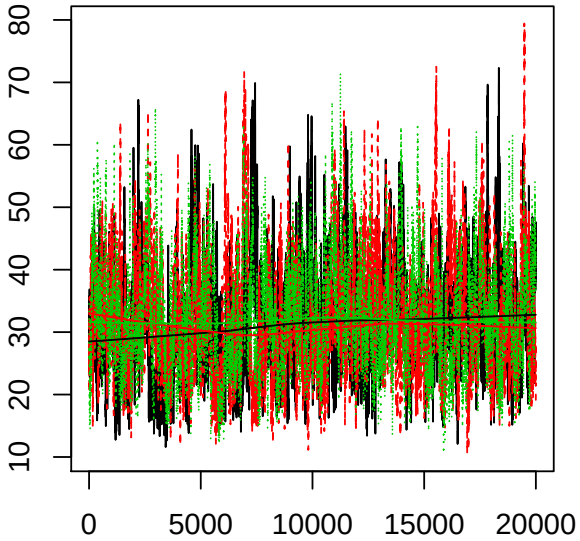


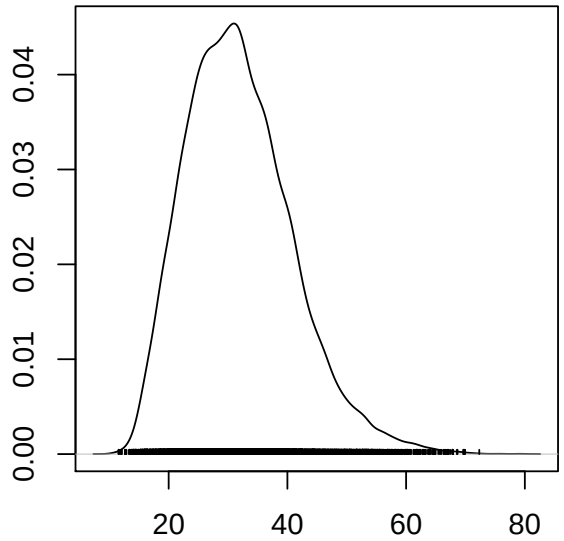


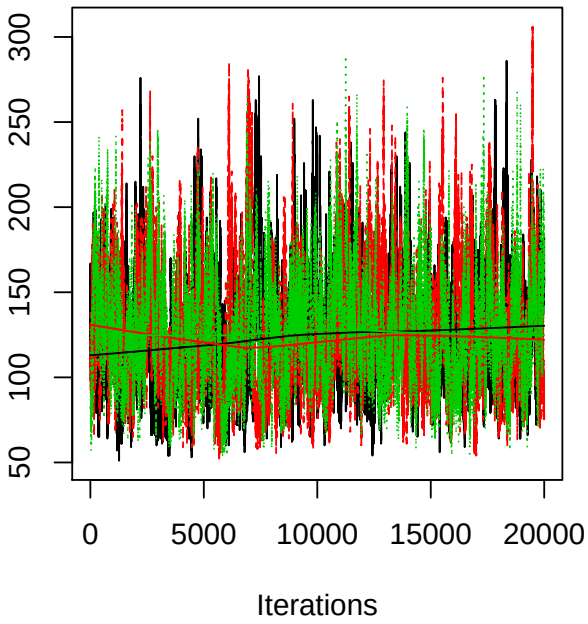


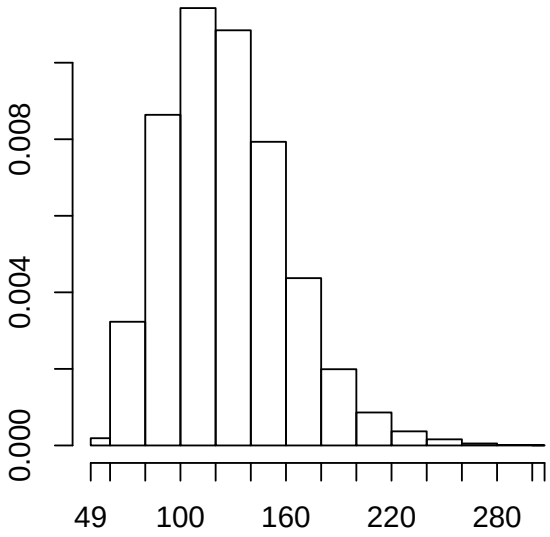

2.5
3.0
3.5
4.0
0
5000
10000
15000
20000
Iterations


0.0
0.4
0.8
1.2
2.5
3.0
3.5
4.0
4.5
N = 20000 Bandwidth = 0.03311


10
20
30
40
50
60
70
80
0
5000
10000
15000
20000
Iterations


0.00
0.01
0.02
0.03
0.04
20
40
60
80
N = 20000 Bandwidth = 1.05





Whoo looks good!

Future Directions

From these models you can expand to all sorts of distance sampling models! Modeling availability is a great addition that combines removal sampling and distance sampling. We can also make these models more complex by adding in additional covariates and using model selection.

Bayesian stats are fun y'all! So many possibilities!