Dynamic N-Mixture Models

Heather Gaya

12/16/2020

N-mixture models are a flexible way to get abundance estimates count data. I've previously discussed closed N-mixture models, but those are way less fun than open population models! Today I'm going to use the example of bird point counts and show two ways to run a multi-season N-mixture model (5 years of data) in both JAGS and NIMBLE. In this case I'll be ignoring any possible density dependence, so it's not really quite as exciting as it sounds.

I'm assuming for now that everyone has downloaded JAGS and NIMBLE and has them setup on their computers. 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

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

Contents

Our Data

For once, we will be using real data, though we'll only be using a portion of it because I'm hoping to publish a paper with this data in the near future and the full dataset is a beast.

In this case, our dataset is point count data from 48 (from the total 109) sites in the Southern Appalachian Mountains. Point counts are performed for 10 minutes in 2.5 minute intervals. All species that are seen or heard are mapped as detected, so we know the general distance interval where each bird was detected at each interval. We also have a record of the time of survey (in minutes since midnight, standardized) and some environmental covariates about each site. Here's a subset of our data:

```
Covs <- read.csv("Site_Covs.csv")
Dets <- read.csv("Site_Obs.csv")
head(Dets, n =3)
##   X.1     X detectionID surveyID species  d1 d2 d3 d4 Year
## 1  47 13166       13254     1178    TUTI  95 95 95 95 2017
## 2  53 18606       18704     1628    TUTI  95 95 95 95 2019
## 3  56  6309        6373      602    OVEN -50 95 95 95 2016
head(Covs, n = 3)
##   X.1 X     Nameyear surveyID PointName Year Observer stand_time meanyeartemp
## 1   1 1 PCCow0012016      848  PCCow001 2016       10  0.9415574     15.97654
## 2   2 2 PCCow0012017     1215  PCCow001 2017        5 -1.4391482     16.01748
## 3   3 3 PCCow0012018     1370  PCCow001 2018       11  1.2189212     17.60273
##   elevation      asp   UTM.N  UTM.W
## 1      1340 14.58158 3879223 275815
## 2      1340 14.58158 3879223 275815
## 3      1340 14.58158 3879223 275815
```

Fun stuff!

The Meat of a Dynamic N-Mixture Model

Recall that the basic N-mixture model assumes that the population is closed, or is only looking at one moment in time. There is some expected number of individual at a site, λ, which is based on site covariates in the form of a linear equation. The actual number of individuals, N, is then drawn from a poisson distribution or some other distribution of your choosing. In a perfect world, the expected and the actual numbers would be equal ($\lambda = N$), but the real world has more variation than that.

In general, a closed-population N-mixture model looks like this:

$$\log(\lambda_i) = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots$$
$$N_i \sim \text{Poisson}(\lambda_i)$$

where λ_i is the expected value of abundance at site i, N_i is the realized value of abundance at site i and x_1 and x_2 are site covariates. Note that you can draw N_i from other distributions, but I prefer the poisson.

Simple Dynamic Model

An intuitive open population n-mixture model can be built simply by indexing everything in a close population model by another dimension (time) and calling it good. I mean, if we think the process is happening in year 1, why wouldn't that same process happen every year?

To write a model this way, we can say:

$$\log(\lambda_{it}) = \beta_0 + \beta_1 x_{it1} + \beta_2 x_{it2} + \cdots$$
$$N_{it} \sim \text{Poisson}(\lambda_{it})$$

where λ_{it} is the expected value of abundance at site i at time t, N_{it} is the realized value of abundance at site i at time t and x_1 and x_2 are site covariates. Note that your covariates can be indexed by time (things like precipitation, temperature, etc.) or just by site (elevation, forest type, etc.).

In the case of our bird data, we think the expected abundance of our species of interest (let's say the Ovenbird) is influenced by the elevational gradient and the average temperature of the site during the breeding season. We could write our model as:

$$\log(\lambda_{it}) = \beta_0 + \beta_1 Elev_i + \beta_2 Temp_{it} + \cdots$$
$$N_{it} \sim \text{Poisson}(\lambda_{it})$$

We could also let our β coefficients be indexed by time if we thought the way ovenbirds react to temperature is changing over time or add additional complexities if we wanted to.

This version of the model is nice because you don't really have to worry about temporal effects or really think about how the individuals arrive at the site or anything like that. They simply are attracted to environmental covariates and more of them arrive when the covariates are favorable. It's a step up from running a bunch of years of data independently because you have the option of sharing the β values between years, which means you have more data to inform your parameters.

However, this formulation ignores temporal autocorrelation and it also is a little weird to think about for some species. For instance, if you had a long-lived species that doesn't migrate or an animal where adults doesn't tend to move from their home territories, we might expect that the population levels from the year before will have a pretty big impact on the current year's abundance. For my own work, Ovenbirds captured and banded at our mist net arrays in North Carolina are often caught on the same plot year after year and will return to their previous year's territories even as populations rise and fall. To better understand these dynamics, we need a slightly more complicated model.

Dail and Madsen (2011) Model

Luckily, ecologists are pretty smart and someone came along and invented a more complicated model! This model comes from a paper by Dale and Madsen (2011) which can be found here: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1541-0420.2010.01465.x It's also used in Applied Hierarchical Modeling by Marc Kéry & Andy Royle, which is an all around great resource for anyone doing Bayesian ecology analysis.

BIDE Model

But before we get to that, let's take a step back. What causes populations to change? An easy framework is the BIDE model - population changes are really just a combination of births, immigration (moving into the pop), deaths and emigration (moving out of the pop). Under this framework, the population at time $t+1$ (N_{t+1}) is just $N_t + B + I - D - E$. This formulation specifies births and deaths in terms of individuals, but we can also think of it in terms of rates (birth rate, death rate, etc.).

Often, we can't really separate deaths and emigration because an animal could wander out of a study site and not come back but still be alive and kicking. Similarly, new animals can show up in an area that have been alive for quite some time but never detected. So instead we end up with apparent survival (animals that did not die and did not emigrate) and apparent recruitment (new animals in the population, either through birth or movement). Since these are both rates, we can now think of the population at $t+1$ as:

$$N_{t+1} = N_t S_t + N_t G_t$$

where S_t is apparent survival and G_t is apparent recruitment.

Note: This will come up in a minute, but does it really make sense for immigration to be influenced by the previous year's population? Probably not, but we'll get there.

Apparent Recruitment and Survival

So this is all well and good, you might say to yourself, but how does that relate to N-mixture models? All we have is count data, how the heck are we supposed to get survival and recruitment parameters out of that? This is where the magic happens.

First we need to deal with Year 1, where we have no previous year's data to work with. In year 1 (or time step 1, whatever you want to call it), we will model the population's expected abundance (λ) from environmental covariates, just like we do with a simple N-mixture model or the independent form of the dynamic N-mixture model.

$$\log(\lambda_{it}) = \beta_{\lambda 0} + \beta_{\lambda 1} Elev_i + \beta_{\lambda 2} Temp_{it}$$
$$N_{it} \sim \text{Poisson}(\lambda_{it})$$

Next, we need to model apparent survival rate for each year at each site ϕ_{it} so that we can model apparent survival in terms of counts S_{it}. A logit link and a binomial distribution is a reasonable way to do this. We can add covariates too!

Continuing with the ovenbird example, we know that survival of adult birds is pretty high during the breeding season and predation of adult birds isn't too much of an issue. But since we're modeling apparent survival, we need to consider movement out of a site as well. We know ovenbirds can move away from sites as temperature changes, so we'll include temperature in our model for ϕ_{it}.

$$logit(\phi_{it}) = \beta_{\phi 0} + \beta_{\phi 1} Temp_{it}$$
$$S_{it} \sim \text{Binomial}(N_{i(t-1)}, \phi_{it})$$

Now let's model apparent recruitment. This includes both new animals born in the population and immigrants. Since apparent recruitment can be any positive number, we can model the expected rate with a log link and then use a poisson to draw an integer value for the number of new "recruits". Maybe we think temperature and elevation impact recruitment:

$$log(\gamma_{it}) = \beta_{\gamma 0} + \beta_{\gamma 1} Elev_i + \beta_{\gamma 2} Temp_{it}$$
$$G_{it} \sim \text{Poisson}(N_{i(t-1)} * \gamma_{it})$$

Now we come back to that note I mentioned above. What happens if the previous year had no individuals at that site? In that case, we're drawing from a Poisson with $\lambda = 0$, which means we aren't allowing for previously unoccupied sites to become occupied, which is probably not what we want.

A work around is to model expected recruitment using the expected abundance from the previous time step rather than the realized abundance. Instead of:

$$G_{it} \sim \text{Poisson}(N_{i(t-1)} * \gamma_{it})$$

We can use:

$$G_{it} \sim \text{Poisson}(\lambda_{i(t-1)} * \gamma_{it})$$

This small change will help our model avoid getting "stuck" and will allow for more realistic changes from year to year across our sites.

Okay great! We have our state-process model all worked out, so let's put it all together.

Putting it Together

Here's the full state process model in one go.

Year 1:

$$\log(\lambda_{it}) = \beta_{\lambda 0} + \beta_{\lambda 1} Elev_i + \beta_{\lambda 2} Temp_{it}$$
$$N_{i1} \sim \text{Poisson}(\lambda_{i1})$$

Year > 1:

$$logit(\phi_{it}) = \beta_{\phi 0} + \beta_{\phi 1} Temp_{it}$$
$$S_{it} \sim \text{Binomial}(N_{i(t-1)}, \phi_{it})$$
$$log(\gamma_{it}) = \beta_{\gamma 0} + \beta_{\gamma 1} Elev_i + \beta_{\gamma 2} Temp_{it}$$
$$G_{it} \sim \text{Poisson}(\lambda_{i(t-1)} * \gamma_{it})$$
$$N_{it} = S_{it} + G_{it}$$

Why This Works

One thing that can seem sort of suspicious about this whole model is that if feels like we're making up dynamics from data. Since we have no data on all these parameters specficially, how is this any different from the simple model we had before? Well, even if we don't know anything about what influences survival and recruitment of a species, we have some general ideas about how populations work in general. And in general, populations do not bounce around without reason from year to year. Remember how we've been using a Poisson distribution to go from expected abundance to realized abundance? The thing about a Poisson is that the mean is equal to the variance. So even though the expected (mean) population may not change much from year to year, the realized population can change dramatically!

By using the combination of a Poisson and a Binomial, the expected abundance of the population won't change dramatically but the variance (how much the realized abundance bounces around) will be much smaller. Plus it's just way more fun than the simple model :)

Modeling Detection

We have a lot of options for modeling detection, just like we do with closed population N-mixture models. The only thing that really changes is the indexing of your model. I'm going to use a binomial N-mixture model for this example, because distance sampling can get a little messy, but I plan on doing a future tutorial specifically on distance sampling! We could also use a removal model framework and model what time period we first heard/saw/detected the bird in. Lots of options!

So for now, we'll imagine we didn't collect distance data, just counts per time period. As I mentioned in the intro, our point counts are performed for 10 minutes in 2.5 minute intervals. We also record what time of day we started the point count so we can use that as a covariate for detection. Notice that we could add in interval specific covariates if we had any.

$$logit(p_{it}) = \alpha_0 + \alpha_1 time_{it}$$
$$y_{ijt} \sim \text{Binomial}(N_{it}, p_{it})$$

Our data y_{ijt} is simply the counts of birds detected at site i in time interval j in year t.

Coding the Model in JAGS

Alright, time to turn this math into code! For the purpose of the code I like to call lambda's beta's "psi", things relating to survival "phi" and recruitment betas "gamma" but you can call them whatever you want.

I've also stuck a uniform prior on all the beta coefficients.

Simple Dynamic Model

For the simple model, the code is nice and straightforward. We can also add in a "derived parameter" of total abundance so we can see how the abundance of Ovenbirds is across all sites together.

```
modelstring.birds.simple = "
model {
for (t in 1:n.years){
  for (i in 1:n.sites){
    lambda[i,t] <- exp(psi.0 + psi.1*Elev[i]+psi.2*Temp[i,t])
    N[i,t] ~ dpois(lambda[i,t])

    logit(p[i,t]) <- alpha.0 + alpha.1*Time[i,t]
    for (j in 1:n.visit){
        y[i,j,t] ~ dbin(p[i,t], N[i,t])
      } #end j
    }#end i
    Total.N[t] <- sum(N[1:n.sites,t])
  } #end t

psi.0 ~dunif(-3,3)
psi.1 ~dunif(-3,3)
psi.2 ~dunif(-3,3)
alpha.0 ~dunif(-3,3)
alpha.1 ~dunif(-3,3)
}
"
```

Short and sweet.

Dail and Madsen (2011) Model

For the fancier model, we have to add in a few more terms.

```
modelstring.birds.fancy = "
model {
for (i in 1:n.sites){
  lambda[i,1] <- exp(psi.0 + psi.1*Elev[i]+psi.2*Temp[i,1])
  N[i,1] ~ dpois(lambda[i,1]) #year 1

  for (t in 2:n.years){
    lambda[i,t] <- exp(psi.0 + psi.1*Elev[i]+psi.2*Temp[i,t])
    logit(phi[i,t]) <- phi.0 + phi.1*Temp[i,t]
    S[i,t] ~ dbin(phi[i,t], N[i,t-1]) #probability then size

    log(gamma[i,t]) <- gamma.0 + gamma.1*Elev[i] + gamma.2*Temp[i,t]
    G[i,t] ~ dpois(lambda[i,t-1]*gamma[i,t])

    N[i,t] <- S[i,t] + G[i,t]
```

```
  } #end t

  for (t in 1:n.years){
    logit(p[i,t]) <- alpha.0 + alpha.1*Time[i,t]
    for (j in 1:n.visit){
      y[i,j,t] ~ dbin(p[i,t], N[i,t])
      } #end j
    }#end t again
  } #end i

psi.0 ~dunif(-3,3)
psi.1 ~dunif(-3,3)
psi.2 ~dunif(-3,3)
phi.0 ~ dunif(-4,3)
phi.1 ~ dunif(-3,3)
gamma.0 ~dunif(-3,3)
gamma.1 ~dunif(-3,3)
gamma.2 ~dunif(-3,3)
alpha.0 ~dunif(-1,3)
alpha.1 ~dunif(-3,3)

for (t in 1:n.years){
  Total.N[t] <- sum(N[,t])
  }
}
"
```

Data Cleanup

Before we can run these models, we'll need to get our data in order. Luckily for N-Mixture models the "data" is pretty straight forward. We'll need to give JAGS (or NIMBLE) the elevation, temperature, and survey time covariates, the number of sites with data, the number of years of data we are analyzing and the number of birds we saw at each site in each time interval in each year.

Let's get to work.

```
Covs <- read.csv("Site_Covs.csv")
Dets <- read.csv("Site_Obs.csv")
head(Dets, n =3)
##   X.1     X detectionID surveyID species  d1 d2 d3 d4 Year
## 1  47 13166       13254     1178    TUTI  95 95 95 95 2017
## 2  53 18606       18704     1628    TUTI  95 95 95 95 2019
## 3  56  6309        6373      602    OVEN -50 95 95 95 2016
head(Covs, n = 3)
##   X.1 X     Nameyear surveyID PointName Year Observer stand_time meanyeartemp
## 1   1 1 PCCow0012016      848  PCCow001 2016       10  0.9415574     15.97654
## 2   2 2 PCCow0012017     1215  PCCow001 2017        5 -1.4391482     16.01748
## 3   3 3 PCCow0012018     1370  PCCow001 2018       11  1.2189212     17.60273
##   elevation      asp   UTM.N  UTM.W
## 1      1340 14.58158 3879223 275815
## 2      1340 14.58158 3879223 275815
## 3      1340 14.58158 3879223 275815
```

First we need to match up the surveyID with the point name, so we know when and where we saw each bird. We then want to filter our data so that we only keep the information on our example species, the

Ovenbird (OVEN). Finally, we want to know how many birds were detected in each time interval (d1, d2, d3, d4 represent the distance the bird was detected in the 4 time intervals).

```
library(tidyr)
library(dplyr)

birds <- merge(Dets[Dets$species == "OVEN",], Covs, by = "surveyID", all.y = T)
oven.detects <- birds %>%
    group_by(PointName, Year.y, stand_time, meanyeartemp, elevation, UTM.N, UTM.W) %>%
      summarize(t1 = sum(d1 >= 0), t2 = sum(d2 >= 0),
                t3 = sum(d3 >= 0), t4 = sum(d4 >= 0)) %>%
        replace_na(list(t1 =0, t2 = 0, t3 =0, t4 =0))
## `summarise()` regrouping output by 'PointName', 'Year.y', 'stand_time', 'meanyeartemp', 'elevation',
head(oven.detects[order(oven.detects$PointName, oven.detects$Year.y),], n = 7)
## # A tibble: 7 x 11
## # Groups:   PointName, Year.y, stand_time, meanyeartemp, elevation, UTM.N [7]
##   PointName Year.y stand_time meanyeartemp elevation  UTM.N  UTM.W    t1    t2
##   <fct>      <int>      <dbl>        <dbl>     <int>  <int>  <int> <dbl> <dbl>
## 1 PCCow001    2016      0.942         16.0      1340 3.88e6 275815     0     0
## 2 PCCow001    2017     -1.44          16.0      1340 3.88e6 275815     0     0
## 3 PCCow001    2018      1.22          17.6      1340 3.88e6 275815     0     0
## 4 PCCow001    2019     -1.06          17.5      1340 3.88e6 275815     2     2
## 5 PCCow001    2020      1.55          13.6      1340 3.88e6 275815     0     0
## 6 PCCow002    2016     -1.37          15.7      1325 3.88e6 274813     0     1
## 7 PCCow002    2017      0.560         15.9      1325 3.88e6 274813     2     1
## # ... with 2 more variables: t3 <dbl>, t4 <dbl>
```

Now we just need to squish this data into the right format and grab the covariates. Our model needs the data in a 3-D matrix where rows = sites, columns = intervals and the 3rd dimension = years. I prefer a for-loop for this operation but you can do it however you want.

```
oven.detects$st_temp <- scale(oven.detects$meanyeartemp) #standardize temp
oven.detects$st_elev <- scale(oven.detects$elevation) #standardize elevation
n.sites = length(unique(oven.detects$PointName)) #48 sites
n.years = length(unique(oven.detects$Year.y)) #5 years
st_Elev = array(NA, n.sites)
st_Temp = array(NA, dim = c(n.sites, n.years))
time = array(NA, dim = c(n.sites, n.years))
y <- array(NA, dim = c(n.sites, 4, n.years))
for (i in 1:nrow(oven.detects)){
  s = as.numeric(oven.detects$PointName[i])
  year = oven.detects$Year.y[i] - 2015 #first year = 2016
  y[s,,year] <- as.numeric(oven.detects[i,c("t1", "t2", "t3","t4")])
  st_Elev[s] <- oven.detects$st_elev[i]
  st_Temp[s,year] <- oven.detects$st_temp[i]
  time[s,year] <- oven.detects$stand_time[i]
}
```

Yay, our data is organized and good to go!

Runing the Simple Model

Initial values for the simple dynamic model are, well, simple. The only thing we need to provide to JAGS to get our model to run is values of N that are always larger than y - that is, we always make sure the abundance of birds is higher than the number detected - and some initial detection values. Beyond that we have pretty smooth sailing to run the model.

```r
library(runjags)
params.birds.simple <- c("psi.0", "psi.1", "psi.2",
                         "alpha.0", "alpha.1", "Total.N")
jd.birds <- list(y = y,
                 Elev = st_Elev,
                 Temp = st_Temp,
                 Time = time,
                 n.sites = n.sites,
                 n.visit = 4,
                 n.years = n.years)
ji.birds.simple <- function(){list(
  alpha.0 = runif(1, 0, 1),
  alpha.1= runif(1),
  N = apply(jd.birds$y, c(1,3), max)+2
  )}

jags.birds.simple <- run.jags(model = modelstring.birds.simple,
                      inits = ji.birds.simple,
                     monitor = params.birds.simple,
                     data = jd.birds, n.chains = 3, burnin =  1000,
                     sample = 4000, method = "parallel", silent.jags = T)
## 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...
## All chains have finished
## Simulation complete.  Reading coda files...
## Coda files loaded successfully
## Finished running the simulation
simple.mod <- summary(jags.birds.simple)
simple.mod
##                 Lower95       Median    Upper95         Mean         SD Mode
## psi.0       0.104432000  0.226264500  0.3476010  0.225643652 0.06224157   NA
## psi.1      -0.190681000 -0.078205400  0.0294353 -0.078064161 0.05689629   NA
## psi.2      -0.000679311  0.106552500  0.2165600  0.107680222 0.05499820   NA
## alpha.0     0.344884000  0.520027500  0.6797810  0.519342683 0.08578268   NA
## alpha.1    -0.168070000  0.006690605  0.1868660  0.006009907 0.09114860   NA
## Total.N[1] 66.000000000 70.000000000 75.0000000 70.673166667 2.51579161   70
## Total.N[2] 61.000000000 65.000000000 70.0000000 65.486166667 2.50196529   65
## Total.N[3] 54.000000000 59.000000000 63.0000000 59.081583333 2.35804810   59
## Total.N[4] 62.000000000 68.000000000 73.0000000 68.230750000 2.87312704   68
## Total.N[5] 39.000000000 41.000000000 44.0000000 41.424166667 1.58538915   41
##                  MCerr MC%ofSD SSeff       AC.10     psrf
## psi.0      0.0009310236     1.5  4469 0.005455650 1.000561
## psi.1      0.0006964751     1.2  6674 0.002139342 1.000229
## psi.2      0.0006862228     1.2  6423 0.014576110 1.000124
## alpha.0    0.0019131296     2.2  2011 0.059671751 1.001783
## alpha.1    0.0019536691     2.1  2177 0.043392117 1.001338
## Total.N[1] 0.0471603004     1.9  2846 0.011838135 1.000084
## Total.N[2] 0.0453374326     1.8  3045 0.011936357 1.001852
## Total.N[3] 0.0429508046     1.8  3014 0.028367708 1.000678
## Total.N[4] 0.0534905618     1.9  2885 0.028593546 1.000365
## Total.N[5] 0.0259417597     1.6  3735 0.018709219 1.000399
```

What do these results tell us?

From the estimates of psi, we can see that there may be a slight negative relationship with elevation and a positive relationship with temperature. Our detection was not greatly influenced by the time of day of the survey but there was maybe a slightly positive relationship between detection and time. And most importantly, we see an estimate of a decrease in population across all sites together.

Let's see what the fancier model tells us.

Runing the Dail Madsen Model

Giving the model parameters and data is pretty easy, but initial values become a lot trickier under this model.

```
params.birds.fancy <- c("psi.0", "psi.1", "psi.2", "phi.0",
                        "phi.1", "gamma.0", "gamma.1", "gamma.2",
                        "alpha.0", "alpha.1", "Total.N")
jd.birds <- list(y = y,
                 Elev = st_Elev,
                 Temp = st_Temp,
                 Time = time,
                 n.sites = n.sites,
                 n.visit = 4,
                 n.years = n.years)
```

One way to go about doing it is to simulate some initial values that we know will at least let the model start running. They don't have to be reasonable end results, just anything that will help the model not run into issues where y (the observations) are larger than the sum of S+G.

First, try to make sensible values for year 1. This is the relevant part of the model:

```
  lambda[i,1] <- exp(psi.0 + psi.1*Elev[i]+psi.2*Temp[i,1])
  N[i,1] ~ dpois(lambda[i,1])
```

We can make up a random value for detection and a fake N to start with and use this as data to run a quick glm. I like to make detection really low to ensure N will be high enough.

```
init.p <- .2
init.N1 <- apply(y[,,1],1, max)/init.p
coefs <- unname(glm(round(init.N1)~elevation + temp,
                    data = data.frame(init.N1 = init.N1, elevation = st_Elev, temp = st_Temp[,1]),
                    family = "poisson")$coefficients)
coefs #intercept, betas
## [1] 2.0112994 0.1133845 1.0466843
```

Next we have to come up with reasonable values for gamma and phi. Again, here's the relevant part of the model:

```
    lambda[i,t] <- exp(psi.0 + psi.1*Elev[i]+psi.2*Temp[i,t])
    logit(phi[i,t]) <- phi.0 + phi.1*Temp[i,t]
    S[i,t] ~ dbin(phi[i,t], N[i,t-1]) #probability then size

    log(gamma[i,t]) <- gamma.0 + gamma.1*Elev[i] + gamma.2*Temp[i,t]
    G[i,t] ~ dpois(lambda[i,t-1]*gamma[i,t])

    N[i,t] <- S[i,t] + G[i,t]
```

We can do this by simulating and checking that all y values are less than or equal to N. Rather than trying it once, seeing if it works, trying again if it didn't, etc. we can just have our for-loop "break" once it finds a solution.

```
n.sim = 50
N.sim <- S.sim <- G.sim <- array(NA, dim = c(n.sites, n.years, n.sim))
phi0.sim <- phi1.sim <- gamma0.sim <- gamma1.sim <- gamma2.sim <- array(NA, dim = c(n.sim))
phi.sim <- gamma.sim <- array(NA, dim = c(n.sites, n.years, n.sim))
lambda.sim <- array(NA, dim = c(n.sites, n.sim))
good <- array(NA, dim = c(n.years, n.sim))
works <- array(NA, n.sim)
```

```
for(q in 1:n.sim){
  phi0.sim[q]<- runif(1,-2,2)
  phi1.sim[q]<- runif(1,-1,1)
  gamma0.sim[q]<- runif(1,-2,2)
  gamma1.sim[q] <- runif(1,-1,1)
  gamma2.sim[q] <- runif(1,-1,1)

for (i in 1:n.sites){
    lambda.sim[i,q] <- exp(coefs[1] + coefs[2]*st_Elev[i]+coefs[3]*st_Temp[i,1])
    N.sim[i,1,q] <- rpois(1, lambda.sim[i,q])

 for (t in 2:n.years){
   phi.sim[i,t, q] <- plogis(phi0.sim[q] + phi1.sim[q]*st_Temp[i,t])
   S.sim[i,t,q] <- rbinom(1, N.sim[i,t-1,q], phi.sim[i,t,q])
   gamma.sim[i,t,q] <- exp(gamma0.sim[q] + gamma1.sim[q]*st_Elev[i] + gamma2.sim[q]*st_Temp[i,t])
   G.sim[i,t,q] <- rpois(1, N.sim[i,t-1,q]*gamma.sim[i,t,q])
   N.sim[i,t,q] <- S.sim[i,t,q] + G.sim[i,t,q]
 } #end t
} #end i
for (t in 1:n.years){
max.obs <- apply(y[,,t],1, max)
good[t,q] <- sum(N.sim[,t,q] - max.obs > 0)
}
works[q] <- min(good[,q]) == n.sites
if(is.na(works[q])) {works[q] <- FALSE} #deal with infinities
if(works[q] == TRUE){break} #if you find a workable init combo, stop looking
} #end q
works*1
##  [1]  0  0  1 NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [26] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
use <- which(works*1 == 1)
```

Once we have our "solution", we can use it to make our initial values list and send it to JAGS. Note that I've upped the burnin quite a bit on this one. The model will converge a lot slower than it did under the simple formulation.

```
ji.birds.fancy <- function() {
  list(psi.0 = coefs[1],psi.1 = coefs[2], psi.2 = coefs[3],
    phi.0 = phi0.sim[use], phi.1 = phi1.sim[use],
    gamma.0 = gamma0.sim[use], gamma.1 = gamma1.sim[use], gamma.2 = gamma2.sim[use])}

jags.birds.fancy <- run.jags(model = modelstring.birds.fancy,
                       inits = ji.birds.fancy,
                     monitor = params.birds.fancy,
                     data = jd.birds, n.chains = 3, adapt = 2000, burnin =  15000,
                     sample = 2000, method = "parallel", silent.jags = T)
```

```
## 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...
## All chains have finished
## Simulation complete.  Reading coda files...
## Coda files loaded successfully
## Finished running the simulation
mod.fancy <- summary(jags.birds.fancy)
mod.fancy
##                Lower95      Median    Upper95        Mean          SD Mode
## psi.0       0.13841900  0.37535150  0.5975290  0.37323119 0.11924843   NA
## psi.1      -0.29708900 -0.06705060  0.1677150 -0.06707128 0.12162170   NA
## psi.2      -0.31608800 -0.09415365  0.1740200 -0.09432068 0.12387774   NA
## phi.0      -3.99993000 -3.30405000 -2.1927700 -3.21421545 0.56002362   NA
## phi.1      -1.21849000 -0.30271200  0.8544720 -0.24564472 0.55320332   NA
## gamma.0    -0.54306400 -0.22980200  0.0875593 -0.22965365 0.16277904   NA
## gamma.1    -0.31034400 -0.03371500  0.2239200 -0.03296556 0.13925204   NA
## gamma.2    -0.00488391  0.13399500  0.2862760  0.13461172 0.07331900   NA
## alpha.0     0.36842300  0.53117350  0.7016790  0.53005367 0.08533004   NA
## alpha.1    -0.15789500  0.01199905  0.2029990  0.01310741 0.09219658   NA
## Total.N[1] 66.00000000 71.00000000 76.0000000 71.34916667 2.77888521   71
## Total.N[2] 61.00000000 65.00000000 70.0000000 65.23500000 2.51153863   65
## Total.N[3] 55.00000000 59.00000000 63.0000000 58.98533333 2.39222463   58
## Total.N[4] 62.00000000 67.00000000 72.0000000 67.32116667 2.75752654   67
## Total.N[5] 39.00000000 41.00000000 44.0000000 41.13366667 1.55087097   40
##                 MCerr MC%ofSD SSeff      AC.10     psrf
## psi.0      0.005150205     4.3   536 0.18451991 1.006931
## psi.1      0.005497795     4.5   489 0.18031766 1.002938
## psi.2      0.003547904     2.9  1219 0.04733859 1.005371
## phi.0      0.058700615    10.5    91 0.63003681 1.043890
## phi.1      0.080446942    14.5    47 0.69576407 1.034502
## gamma.0    0.008232635     5.1   391 0.28225519 1.013103
## gamma.1    0.006246324     4.5   497 0.15975077 1.001933
## gamma.2    0.004667908     6.4   247 0.24577513 1.009956
## alpha.0    0.002751993     3.2   961 0.07488274 1.000914
## alpha.1    0.002736694     3.0  1135 0.04494712 1.006191
## Total.N[1] 0.087580353     3.2  1007 0.08497958 1.000327
## Total.N[2] 0.065832433     2.6  1455 0.04151681 1.002012
## Total.N[3] 0.063972811     2.7  1398 0.05040736 1.002184
## Total.N[4] 0.074282343     2.7  1378 0.03523272 1.001356
## Total.N[5] 0.037586534     2.4  1702 0.04180279 1.002185
```

Okay, what have we learned from this model? (Note: It is also good practice to plot all these results and make sure we visually inspect the chains, but I didn't want this tutorial to be 100 pages.)

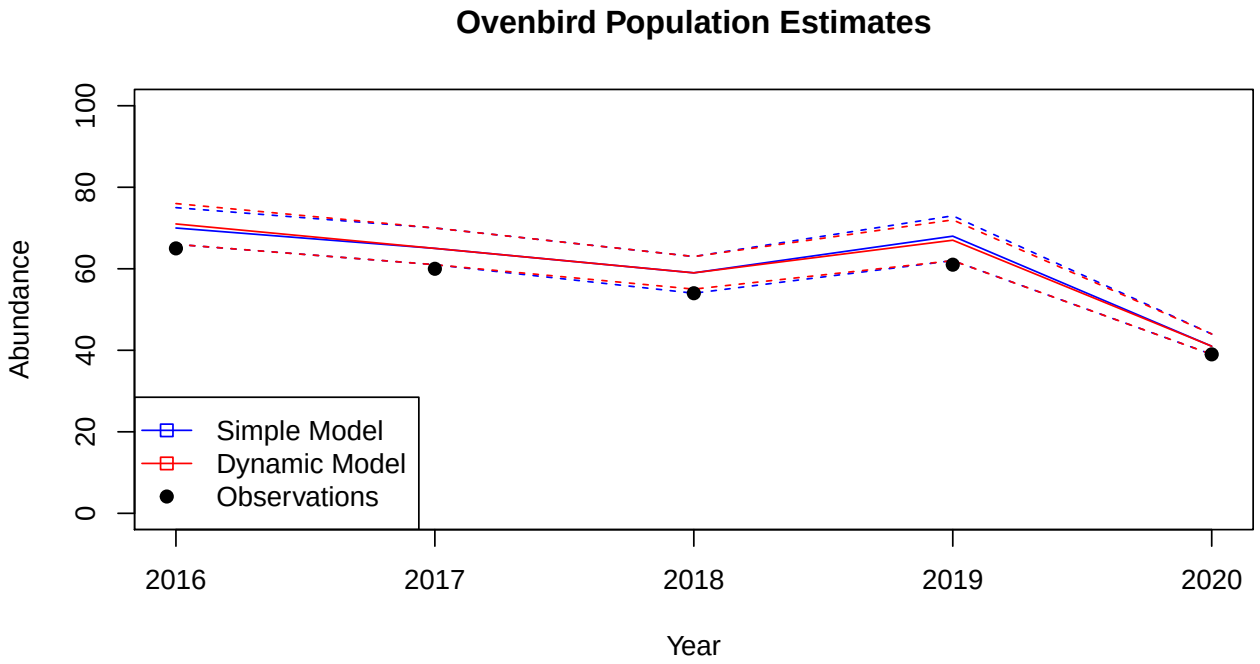
The expected abundance has a slight negative relationship with both elevation and temperature. Estimated apparent survival also has a slight negative relationship with temperature. Estimated apparent recruitment has a negative relationship with elevation and a positive relationship with temperature.

Remember this is real data so there's a ton of noise and I don't know the "real answer" to the population dynamics. There's also some underlying dynamics in this population that aren't being addressed with this model - namely competition with some very ecologically similar species in the area - but for the most part this matches what we already knew about ovenbirds. Cool!

Graphing the Results

Let's graph our results to get a better idea of what we found out. First, let's compare the population estimates from the two models with the observation data.

```
plot(2016:2020, simple.mod[6:10,2], type = "l",
     main = "Ovenbird Population Estimates", xlab = "Year",
     ylab = "Abundance", ylim = c(0,100), col = "blue")
lines(2016:2020, simple.mod[6:10,1], lty =2, col = "blue")
lines(2016:2020, simple.mod[6:10,3], lty =2, col = "blue")
lines(2016:2020, mod.fancy[11:15,2], lty =1, col = "red")
lines(2016:2020, mod.fancy[11:15,1], lty =2, col = "red")
lines(2016:2020, mod.fancy[11:15,3], lty =2, col = "red")
points(2016:2020, colSums(apply(jd.birds$y, c(1,3), max)),
       col = "black", pch = 19)
legend("bottomleft",
       c("Simple Model", "Dynamic Model", "Observations"),
       col = c("blue", "red", "black"),
       lty = c(1,1,0), pch = c(0,0,19))
```



The two models produce very similar results!

How about a fun spatial plot? First we'll want to grab the estimates of N for each site and make a dataframe with the locations of each point count.

```
N.ests <- extend.jags(jags.birds.simple,
              add.monitor = "N", drop.monitor = params.birds.simple,
              burnin = 1000, sample = 5000)
## Calling 3 simulations using the parallel method...
## All chains have finished
## Simulation complete.  Reading coda files...
## Coda files loaded successfully
## Note: Summary statistics were not produced as there are >50 monitored
## variables
## [To override this behaviour see ?add.summary and ?runjags.options]
```

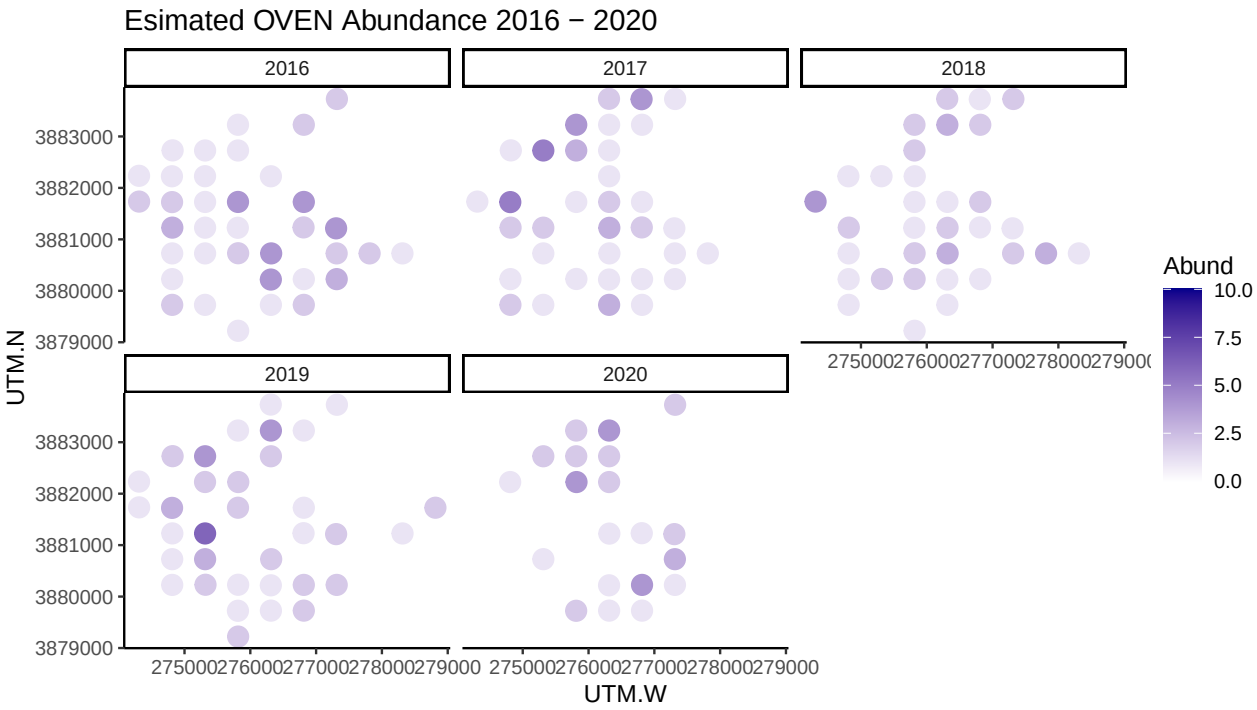
```
## FALSEFinished running the simulation
summary.N <- summary(N.ests)
## Calculating summary statistics...
## Calculating the Gelman-Rubin statistic for 240 variables....

points <- distinct(oven.detects[,c("PointName", "UTM.N", "UTM.W")])

abund <- data.frame(Abund = summary.N[,2],  #technically the median
                            upperAbund =  summary.N[,3],
                            lowerAbund =  summary.N[,1],
                            Name = rep(points$PointName,n.years),
                            UTM.N = rep(points$UTM.N, n.years),
                            UTM.W = rep(points$UTM.W, n.years),
                            year = rep(2016:2020, each = n.sites))
```

Now we just send to ggplot. Obviously you could adjust this graph to make it prettier or display other information as you saw fit.

```
library(ggplot2)
## Warning: package 'ggplot2' was built under R version 3.6.2
ggplot(abund, aes(x = UTM.W, y = UTM.N))+
  geom_point(aes(col = Abund), size = 4)+
  facet_wrap(~year)+
  scale_color_gradient(low = 'white', high = 'blue4',limits = c(0, 10))+
  theme_classic()+
  ggtitle("Esimated OVEN Abundance 2016 - 2020 ")
```



Pretty neat! This model suggests that not only is abundance changing, the spatial density of ovenbirds is also changing over time. Neat!

Coding the Model in NIMBLE

The switch to NIMBLE is pretty easy for these models. Let's take a look.

Simple Dynamic Model

Nothing in the meat of the model changes when we move over to NIMBLE.

```
library(nimble)
nimblebirds.simple <-
  nimbleCode({
for (t in 1:n.years){
  for (i in 1:n.sites){
    lambda[i,t] <- exp(psi.0 + psi.1*Elev[i]+psi.2*Temp[i,t])
    N[i,t] ~ dpois(lambda[i,t])

    logit(p[i,t]) <- alpha.0 + alpha.1*Time[i,t]
    for (j in 1:n.visit){
        y[i,j,t] ~ dbin(p[i,t], N[i,t])
      } #end j
    }#end i
    Total.N[t] <- sum(N[1:n.sites,t])
  } #end t

psi.0 ~dunif(-3,3)
psi.1 ~dunif(-3,3)
psi.2 ~dunif(-3,3)
alpha.0 ~dunif(-3,3)
alpha.1 ~dunif(-3,3)
})
```

Short and sweet.

Dail and Madsen (2011) Model

The only real change is turning sum(N[,t]) into sum(N[1:n.sites,t]). No biggie.

```
nimblebirds.fancy <-
  nimbleCode({
for (i in 1:n.sites){
  lambda[i,1] <- exp(psi.0 + psi.1*Elev[i]+psi.2*Temp[i,1])
  N[i,1] ~ dpois(lambda[i,1]) #year 1

  for (t in 2:n.years){
    lambda[i,t] <- exp(psi.0 + psi.1*Elev[i]+psi.2*Temp[i,t])
    logit(phi[i,t]) <- phi.0 + phi.1*Temp[i,t]
    S[i,t] ~ dbin(phi[i,t], N[i,t-1]) #probability then size

    log(gamma[i,t]) <- gamma.0 + gamma.1*Elev[i] + gamma.2*Temp[i,t]
    G[i,t] ~ dpois(lambda[i,t-1]*gamma[i,t])

    N[i,t] <- S[i,t] + G[i,t]
  } #end t

  for (t in 1:n.years){
    logit(p[i,t]) <- alpha.0 + alpha.1*Time[i,t]
```

```
    for (j in 1:n.visit){
      y[i,j,t] ~ dbin(p[i,t], N[i,t])
      } #end j
    }#end t again
  } #end i

psi.0 ~dunif(-3,3)
psi.1 ~dunif(-3,3)
psi.2 ~dunif(-3,3)
phi.0 ~ dunif(-4,3)
phi.1 ~ dunif(-3,3)
gamma.0 ~dunif(-3,3)
gamma.1 ~dunif(-3,3)
gamma.2 ~dunif(-3,3)
alpha.0 ~dunif(-1,3)
alpha.1 ~dunif(-3,3)

for (t in 1:n.years){
  Total.N[t] <- sum(N[1:n.sites,t])
  }
})
```

Runing the Simple Model

Just like with JAGS we'll want to get our data, initials, params, etc. in order.

```
nc.simple <- list(n.sites = n.sites,
                  n.visit = 4,
                  n.years = n.years)
nd.simple <- list(y = y,
                  Elev = st_Elev,
                  Temp = st_Temp,
                  Time = time)
np.simple <- c("psi.0", "psi.1", "psi.2", "alpha.0", "alpha.1", "Total.N")
ni.simple <- list(
  alpha.0 = runif(1, 0, 1),
  alpha.1= runif(1),
  N = apply(y, c(1,3), max)+2)
```

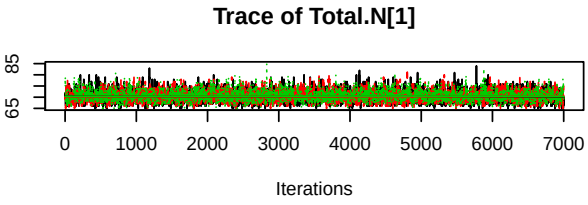
Send it to NIMBLE in parallel to save time

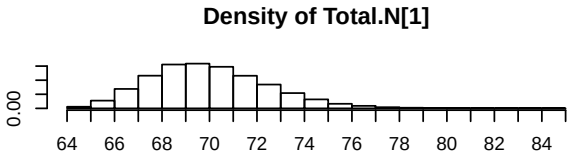
```
library(coda)
library(parallel)
cl <- makeCluster(3)
clusterExport(cl = cl, varlist = c("nc.simple",
              "nd.simple", "ni.simple", "np.simple", "nimblebirds.simple"))
birds.out <- clusterEvalQ(cl = cl,{
  library(nimble)
  library(coda)
prepbirds <- nimbleModel(code = nimblebirds.simple, constants = nc.simple,
                         data = nd.simple, inits = ni.simple)
mcmcbirds<- configureMCMC(prepbirds, monitors = np.simple, 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
```

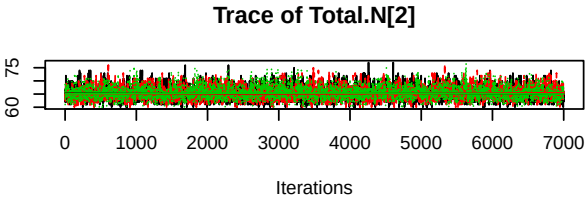
```
Compbirds$run(nburnin = 1000, niter = 8000) #if you run this in your console it will say "null".
return(as.mcmc(as.matrix(Compbirds$mvSamples)))
})
birds.nimble.simple <- mcmc.list(birds.out)
stopCluster(cl)
```

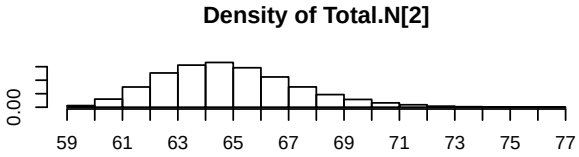
Let's make sure our model is converged.

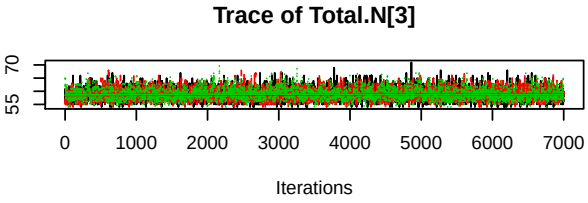
```
gelman.diag(birds.nimble.simple)$psrf
##            Point est. Upper C.I.
## Total.N[1]   1.001772   1.006859
## Total.N[2]   1.000074   1.000188
## Total.N[3]   1.000713   1.001661
## Total.N[4]   1.000502   1.001485
## Total.N[5]   1.000283   1.001397
## alpha.0      1.001274   1.003321
## alpha.1      1.005732   1.012902
## psi.0        1.002631   1.007893
## psi.1        1.000101   1.000296
## psi.2        1.005424   1.016441
plot(birds.nimble.simple)
```

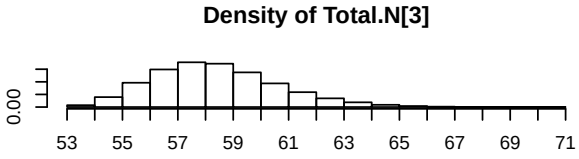


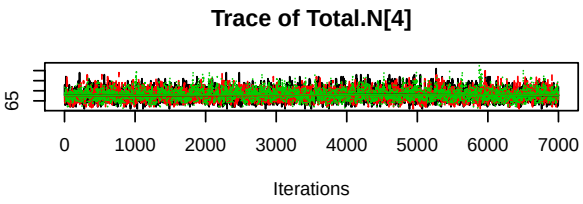


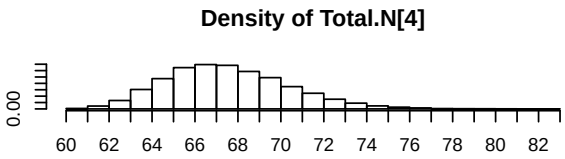


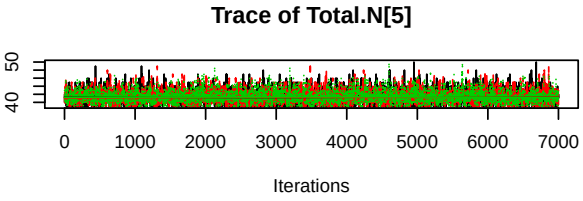


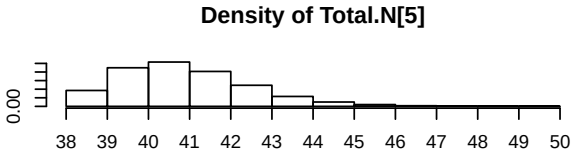


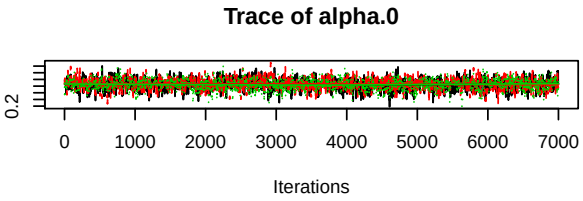


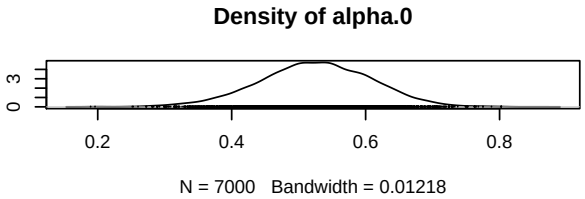


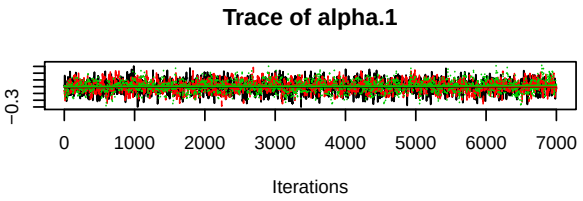


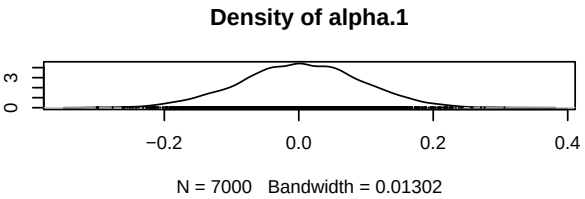


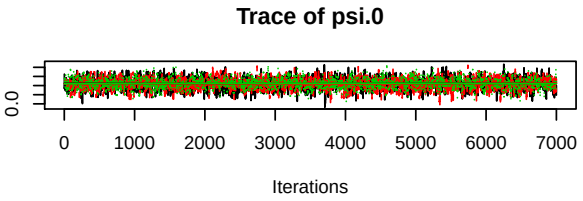


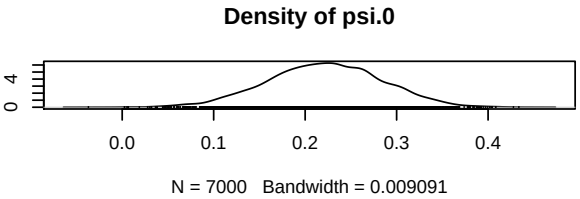


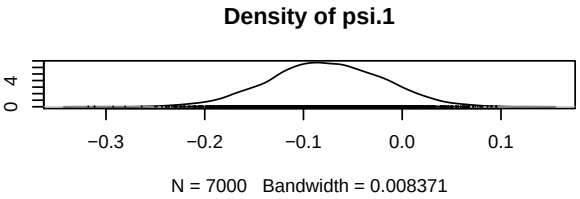


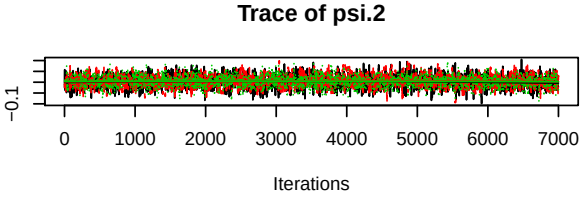


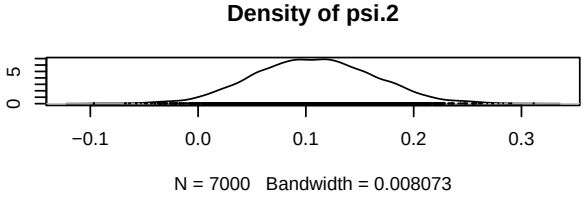












Our results should be the same as with JAGS.

```
summary(birds.nimble.simple)
##
```

```
## Iterations = 1:7000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 7000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##                 Mean      SD  Naive SE Time-series SE
## Total.N[1] 70.507619 2.50469 0.0172840      0.0398869
## Total.N[2] 65.359000 2.46984 0.0170435      0.0393205
## Total.N[3] 58.873429 2.27446 0.0156953      0.0343110
## Total.N[4] 68.021143 2.86749 0.0197876      0.0481594
## Total.N[5] 41.399190 1.59235 0.0109883      0.0207931
## alpha.0     0.525079 0.08410 0.0005803      0.0019536
## alpha.1     0.003399 0.09020 0.0006224      0.0020338
## psi.0       0.221160 0.06277 0.0004331      0.0010401
## psi.1      -0.077200 0.05780 0.0003988      0.0009045
## psi.2       0.107771 0.05574 0.0003847      0.0008405
##
## 2. Quantiles for each variable:
##
##                2.5%      25%       50%      75%   97.5%
## Total.N[1] 66.000000 69.00000 70.000000 72.00000 76.0000
## Total.N[2] 61.000000 64.00000 65.000000 67.00000 71.0000
## Total.N[3] 55.000000 57.00000 59.000000 60.00000 64.0000
## Total.N[4] 63.000000 66.00000 68.000000 70.00000 74.0000
## Total.N[5] 39.000000 40.00000 41.000000 42.00000 45.0000
## alpha.0     0.354155  0.47077  0.526100  0.58347  0.6855
## alpha.1    -0.175685 -0.05611  0.003789  0.06434  0.1771
## psi.0       0.096625  0.17927  0.221785  0.26393  0.3412
## psi.1      -0.191146 -0.11609 -0.077530 -0.03740  0.0336
## psi.2       0.001431  0.06984  0.107081  0.14559  0.2161
```

Runing the Dail Madsen Model

As with JAGS, the Dail Madsen model becomes a little trickier to run. Before we send the model to parallel computing, the only changes in our commands are in the initial and parameter objects. I'm using the same initials I used for the JAGS model. See the JAGS section for an explanation of how I got them.

```
np.fancy <- c("psi.0", "psi.1", "psi.2",
              "phi.0", "phi.1", "gamma.0", "gamma.1",
              "gamma.2", "alpha.0", "alpha.1", "Total.N")
nc.fancy <- list(n.sites = n.sites,
                 n.visit = 4,
                 n.years = n.years) #no change from simple
nd.fancy  <- list(y = y,
                 Elev = st_Elev,
                 Temp = st_Temp,
                 Time = time) #no change from simple

ni.fancy <- list(psi.0 = coefs[1],psi.1 = coefs[2], psi.2 = coefs[3],
    phi.0 = phi0.sim[use], phi.1 = phi1.sim[use],
    gamma.0 = gamma0.sim[use], gamma.1 = gamma1.sim[use], gamma.2 = gamma2.sim[use])
```

Time to send it to NIMBLE! However, we have to do one more very important thing - we have to initialize all the S and G values for each year.

```
library(coda)
library(parallel)
cl <- makeCluster(3)
clusterExport(cl = cl,
    varlist = c("nc.fancy", "nd.fancy", "ni.fancy",
                "np.fancy", "nimblebirds.fancy"))
birds.out <- clusterEvalQ(cl = cl,{
  library(nimble)
  library(coda)
prepbirds <- nimbleModel(code = nimblebirds.fancy, constants = nc.fancy,
                         data = nd.fancy, inits = ni.fancy)
prepbirds$simulate("N[,1]") #calculate starting N from psi values
prepbirds$calculate("lifted_lambda_oBi_comma_t_minus_1_cB_times_gamma_oBi_comma_t_cB_L9")
#calculate the lambda[i,t-1]*gamma[i,t] term for all i and t from the inits provided
prepbirds$simulate("G") #simulate all the G terms from the inits provided
prepbirds$simulate("S[,2]") #simulate second year S from newly calculated N[,1]
prepbirds$calculate("N[,2]") #calculate S + G for year 2
prepbirds$simulate("S[,3]") #simulate 3rd year S from newly calculated N[,2]
prepbirds$calculate("N[,3]") #year 3
prepbirds$simulate("S[,4]") # year 4
prepbirds$calculate("N[,4]") #year 4
prepbirds$simulate("S[,5]") #year 5
prepbirds$calculate("N[,5]") # year 5
mcmcbirds<- configureMCMC(prepbirds, monitors = np.fancy, print = T )
birdsMCMC <- buildMCMC(mcmcbirds) #actually build the code for those samplers
Cmodel <- compileNimble(prepbirds) #compiling the model itself in C++;
Compbirds <- compileNimble(birdsMCMC, project = prepbirds) # compile the samplers next
Compbirds$run(nburnin = 20000, niter = 30000) #if you run this in your console it will say "null".
return(as.mcmc(as.matrix(Compbirds$mvSamples)))
})
birds.nimble.fancy <- mcmc.list(birds.out)
stopCluster(cl)
```

Check convergence (you would want to do this before you stop the cluster in case you needed to extend the run)

```
gelman.diag(birds.nimble.fancy)$psrf
##            Point est. Upper C.I.
## Total.N[1]   1.001087   1.003831
## Total.N[2]   1.000000   1.000047
## Total.N[3]   1.000436   1.001486
## Total.N[4]   1.001033   1.003509
## Total.N[5]   1.001353   1.004190
## alpha.0      1.001264   1.003555
## alpha.1      1.004384   1.015716
## gamma.0      1.004529   1.014342
## gamma.1      1.008902   1.031192
## gamma.2      1.026804   1.090921
## phi.0        1.026293   1.078991
## phi.1        1.065733   1.211399
## psi.0        1.003747   1.009915
## psi.1        1.007849   1.028002
```

```
## psi.2        1.002194   1.004809
```

And finally, check out the results. Our results should be the same as with JAGS.

```
summary(birds.nimble.fancy)
##
## Iterations = 1:10000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 10000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##                 Mean      SD  Naive SE Time-series SE
## Total.N[1] 71.381733 2.78666 0.0160888       0.043811
## Total.N[2] 65.298700 2.47055 0.0142637       0.035367
## Total.N[3] 58.854000 2.29837 0.0132697       0.029645
## Total.N[4] 67.274400 2.71661 0.0156843       0.037755
## Total.N[5] 41.159500 1.52020 0.0087769       0.017842
## alpha.0     0.527499 0.08478 0.0004895       0.001692
## alpha.1     0.002398 0.08946 0.0005165       0.001729
## gamma.0    -0.220960 0.16841 0.0009723       0.006329
## gamma.1    -0.028919 0.13683 0.0007900       0.004444
## gamma.2     0.120136 0.07306 0.0004218       0.002500
## phi.0      -3.263595 0.52964 0.0030579       0.032184
## phi.1      -0.018275 0.60411 0.0034878       0.045006
## psi.0       0.374569 0.12846 0.0007417       0.004471
## psi.1      -0.071922 0.11841 0.0006837       0.003894
## psi.2      -0.097293 0.12458 0.0007193       0.002705
##
## 2. Quantiles for each variable:
##
##                2.5%      25%       50%      75%   97.5%
## Total.N[1] 67.00000 69.00000 71.000000 73.00000 77.0000
## Total.N[2] 61.00000 64.00000 65.000000 67.00000 71.0000
## Total.N[3] 55.00000 57.00000 59.000000 60.00000 64.0000
## Total.N[4] 63.00000 65.00000 67.000000 69.00000 73.0000
## Total.N[5] 39.00000 40.00000 41.000000 42.00000 45.0000
## alpha.0     0.35781  0.47114  0.528869  0.58492  0.6904
## alpha.1    -0.17090 -0.05901  0.002744  0.06292  0.1804
## gamma.0    -0.55905 -0.33340 -0.217933 -0.10581  0.1067
## gamma.1    -0.28758 -0.12493 -0.030642  0.06234  0.2440
## gamma.2    -0.01976  0.07093  0.119277  0.16767  0.2707
## phi.0      -3.97179 -3.70804 -3.350296 -2.91810 -2.0659
## phi.1      -1.06641 -0.45032 -0.079334  0.37700  1.2743
## psi.0       0.11464  0.28936  0.376841  0.46171  0.6215
## psi.1      -0.31584 -0.14793 -0.071103  0.01080  0.1472
## psi.2      -0.33783 -0.18408 -0.096401 -0.01440  0.1520
```

Final Notes

Dynamic N-mixture models are a really neat way to take simple data (counts) and learn a lot about population dynamics, but there's always way to improve your models! For instance, in this example I used a binomial detection function, which really doesn't do a great job of capturing the true detection probability in our point counts. A way better method would be removal sampling (time-to-detection models) or distance sampling. Additionally, we didn't do any model selection in this example but if we were publishing this data we definitely would want to do that. So just remember, this is a cool model framework but take the results (especially the OVEN results shown here) with a grain of salt!