Intro to Bayesian N-Mixture Models (Closed Populations)

Heather Gaya

11/4/2020

N-mixture models are a flexible way to get abundance estimates from point counts, electrofishing, double observer data, etc. In this tutorial I will explain some basic N-mixture models and show you how to code and run the models in both JAGS and NIMBLE. I will discuss binomial, double observer, and removal sampling detection models.

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

Fake Scenarios

There's a bunch of ways you can collect data that yield counts of unmarked individuals at different sites. Maybe you're electrofishing and want to estimate the abundance of fish in a stream (removal sampling). Maybe you're doing point counts for birds. Maybe you're counting butterflies in different plots (double observer).

All of these types of data have different underlying methods of detection, but they all have very similar data. The data is focused on the species rather than on the individuals themselves.

Let's walk through these three different examples. For these examples we're only going to discuss *closed* populations.

Simple Counts - Bird Point Counts

There are a lot of ways to do point counts, but one possible method is counting the number of animals seen or heard in a given amount of time. Let's say we go out and look for white-throated sparrows at 20 sites. We visit each site 4 times and take some measurements of the vegetation at the sites we visit and how windy it was (which might affect detection). None of the birds are marked, so we don't know if the ones we see at one visit are the same birds that we see at a different visit.

Here's what our data might look like:

Site	Visit1	Visit2	Visit3	Visit4	Wind1	Wind2	Wind3	Wind4	PercentCover
1	0	0	2	2	6.22	1.62	14.16	0.73	0.88
2	1	2	1	1	14.10	0.59	12.99	8.85	0.77
3	2	3	2	1	2.04	8.60	0.47	6.07	0.28
4	0	1	0	0	2.71	8.58	13.40	5.80	0.53
5	2	2	2	2	8.03	7.40	9.96	4.06	0.96
6	2	1	2	2	3.10	13.44	0.92	14.12	0.98
7	0	1	0	1	10.66	6.67	5.34	7.57	0.09
8	2	2	3	1	2.21	0.77	7.61	4.43	0.07
9	1	0	2	0	12.17	14.56	12.17	13.06	0.33
10	0	0	1	0	12.71	11.36	5.27	0.52	0.37
11	1	3	1	1	12.96	14.25	7.32	14.95	0.72
12	0	0	0	0	8.63	10.06	13.64	11.17	0.76
13	1	0	1	1	11.51	13.95	7.84	3.00	0.00
14	2	2	2	2	13.62	14.76	5.12	12.44	0.74
15	3	3	0	2	3.10	0.93	9.60	6.50	0.19
16	1	1	2	2	7.85	4.34	7.08	9.28	0.45
17	1	1	1	1	5.25	0.35	2.86	5.08	0.32
18	1	2	1	2	13.26	1.45	0.73	7.18	0.11
19	0	1	1	1	13.17	13.39	2.14	14.69	0.29
20	1	1	3	3	12.27	2.27	1.41	3.51	0.82
21	0	0	0	0	11.16	12.93	3.82	1.74	0.49
22	1	1	1	1	12.59	13.48	3.71	13.16	0.03
23	1	1	0	1	13.03	0.05	9.77	7.50	0.44
24	4	4	1	2	6.60	7.82	4.81	0.95	0.08
25	0	0	0	0	1.30	2.42	7.32	12.14	0.26
26	2	0	2	2	6.07	8.90	14.59	12.15	0.07
27	0	0	0	0	8.08	1.88	8.05	14.93	0.91
28	3	2	2	2	8.62	11.50	3.93	5.35	0.99
29	1	1	0	0	7.29	7.01	8.17	3.02	0.06
30	0	0	0	0	13.31	6.24	13.88	10.53	0.68

Removal Sampling - Electrofishing

For our second example, let's imagine we're electrofishing. (I got to try electrofishing twice in Kansas (Konza Prairie, whoo!) and I was terrible at it. But anyway.) When you're electrofishing, you close off a section of a stream/river and shock the water to catch fish, going from one barrier to the other and placing all your captured fish in a bucket. After each pass, the fish *stay* in the bucket - they do not go back into the river. At the end of the survey you count how many fish are in each bucket. We might expect more fish in deeper parts of the stream, but it's also hard to detect fish in deeper water, so we might measure average stream depth at each site to give us an idea of how stream reaches differ. Here's what this type of data might look like:

Site	pass1	pass2	pass3	pass4	StreamDepth
1	5	1	2	1	2.51
2	2	0	0	0	1.08
3	1	3	1	0	2.45
4	2	2	0	0	1.87
5	5	3	2	0	3.46
6	5	0	3	0	3.50
7	8	5	7	1	4.68
8	5	4	5	3	3.95
9	2	1	1	3	2.61
10	0	0	1	0	1.26
11	5	4	1	1	3.13
12	5	5	0	0	3.39
13	4	4	1	0	3.34
14	1	2	1	0	3.63
15	4	7	5	1	4.89
16	6	1	1	0	3.60
17	3	0	1	2	2.27
18	8	2	1	2	4.47
19	5	2	1	0	2.40
20	7	2	1	2	4.10
21	11	2	4	1	4.43
22	2	3	1	2	3.37
23	2	1	0	0	1.78
24	7	2	5	2	3.94
25	4	3	7	3	4.17
26	4	3	2	3	3.82
27	2	0	1	0	1.21
28	7	0	0	0	2.90
29	2	3	1	1	4.50
30	4	2	2	1	3.34

Dependent Double Observer

Double observer sampling can be used when you don't have the resources to do multiple passes of a survey or it would be difficult to use some other method of detection. This is a common method for aerial surveys (deer, waterfowl, etc) where multiple passes with a plane can be expensive. I recently read a study where someone used this for butterflies, so that's the example I'll use here.

You can have dependent or independent observers, but in this case we have a dependent double observer example. The first person walks along, looks for butterflies, and records how many they see. The second person writes down all the animals the first person saw as well as any others they see. The final result is a count of butterflies seen by person 1 and 2 or butterflies seen only by person 2. Butterflies are more likely to be flying around when the temperature is warm and the ground cover is mostly forbs, so we'll also look at

those two covariates.

This type of data looks like this:

Site	BothDetect	Obs2Only	Flowers	Temperature
1	1	0	0.18	-1.29
2	3	2	0.76	-0.89
3	4	0	0.57	0.36
4	0	1	0.05	-0.57
5	3	0	0.14	1.06
6	3	0	0.71	1.37
7	0	1	0.06	2.13
8	3	1	0.92	-2.73
9	3	0	0.21	-1.47
10	2	0	0.60	-0.84
11	1	2	0.37	-0.92
12	2	2	0.99	0.18
13	6	2	0.93	0.06
14	1	1	0.21	0.86
15	2	2	0.58	-0.07
16	2	4	0.84	1.16
17	4	4	0.92	0.45
18	2	2	0.51	0.23
19	5	3	0.77	0.20
20	4	1	0.55	0.12
21	0	1	0.06	0.72
22	0	1	0.50	1.57
23	0	3	0.44	-0.35
24	1	0	0.62	-1.07
25	4	3	0.86	-0.65
26	5	4	0.60	-0.29
27	2	0	0.31	0.11
28	3	2	0.48	0.02
29	3	1	0.15	0.40
30	1	4	0.84	0.15

The Meat of a Closed N-Mixture Model

State Process Model

At the core of an N-mixture model is the idea that there is some expected number of individual at a site. We call this expected number λ. This expected number 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. In a perfect world, the expected and the actual numbers would be equal ($\lambda = N$), but the real world has more variation than that.

Note that this is very similar to an occupancy model! In occupancy model our linear model helps predict the probability of occupancy rather than the mean abundance, but the idea is essentially the same.

In general, our 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.

So if we look at our sparrow point count example, our state model looks like this:

$$\log(\lambda_i) = \beta_0 + \beta_1 tree_i$$
$$N_i \sim \text{Poisson}(\lambda_i)$$

Detection Models

The state process describes what's actually out there. How many birds are really at those sites we counted or how many fish were actually in the stream reach, etc. But obviously we don't know *exactly* how many there are or we wouldn't need fancy models! The detection model will depend on what type of sampling you're doing.

Bird Point Counts - Binomial Detection

For our bird point counts, we can think of our detection probability as just being a proportion. There are some true number of birds out there N_i and each time we only see y_i, where $y_i = N_i{*}p$. We also know that it can be harder to detect birds on windy days, so we expect that our detection at each visit is related to wind and that each time we visit. Of course if we have more covariates or have reason to believe that probability of detection changes between sites we could model our system that way instead, but for now we'll keep it simple.

Note that we will use a logit transform to ensure that p stays between 0 and 1.

$$p_{it} = \frac{\exp(\alpha_0 + \alpha_1 Wind_{it})}{\exp(\alpha_0 + \alpha_1 Wind_{it}) + 1}$$
$$y_{it} \sim \text{Bin}(N_i, p_{it})$$

where p_{it} is the probability of detection at site i during visit t, N_i is the realized value of abundance at site i, y_{it} is the number of individuals observed at site i at visit t and $Wind_{it}$ is the wind speed during visit t at site i.

Electrofishing - Removal Sampling

Detection gets a little more interesting for our removal sampling example. Since we take fish out of the stream each pass, they aren't available to be captured in subsequent passes. So in the first pass, we might capture fish with probability p. But in the second pass, in order to capture a fish, it has to have not been captured previously. So our probability of capture in pass 2 is the probability of missing it the first time $1-p$ times the probability of detecting it the second time p. Likewise, to see it the third time we have to miss it twice $(1-p)*(1-p)$ and then detect it p. Finally, we have a probability of $(1-p)^4$ of never detecting a fish. However, we never get this data specifically, since if we don't detect it, we don't know it was there.

Again we will use a logit transform in our equation for p to make sure detection probability is bound between 0 and 1.

$$p_i = \frac{exp(\alpha_0 + \alpha_1 StreamDepth_i)}{exp(\alpha_0 + \alpha_1 StreamDepth_i) + 1}$$
$$\pi_{i1} = p_i$$
$$\pi_{i2} = (1 - p_i)(p_i)$$
$$\pi_{i3} = (1 - p_i)^2(p_i)$$
$$\pi_{i4} = (1 - p_i)^3(p_i)$$
$$\pi_{i5} = (1 - p_i)^4$$

where p_i is the probability of detecting a fish at site i, π_{i1} to π_{i4} are the probability of detecting a fish at site i in pass 1 to 4 and π_{i5} is the probability of never detecting a fish.

To model this type of probability, we can use a mulitnomial distribution. This is essentially a categorical distribution where the "categories" are the passes- pass 1, pass 2, pass 3, pass 4 or never detected.

But there's a catch. We don't know how many are in the "never detected" bin. We only have data on the fish we actually capture. So what do we do?

The secret is *conditional probability*. Instead of thinking about the probability of detecting or not detecting animals, we can think about the probability of detecting an animal in any particular pass *given* that the animal was caught during the survey. This eliminates the need for thinking about the "never detected" animals and lets us just focus on probability - which is much easier to work with!

The probability of an individual being detected during the survey is just the sum of all the probabilities of catching an animal in each pass.

For any given site:

$$\pi_{detect} = \pi_1 + \pi_2 + \pi_3 + \pi_4$$

Alternatively, it is 1- the probability of never being detected.

$$\pi_{detect} = 1 - \pi_5$$

Importantly, because we are working with conditional probability, we aren't placing N_i individuals into each bin. Instead we're placing n_i - the number that were captured during the survey. How do we get n_i? With a binomial draw, with probability π_{detect}.

$$n_i \sim \text{Bin}(N_i, \pi_{detect})$$

So now we can rethink our categories probabilities. The probability of being caught in pass 1 given that the individual was ever caught is just $\frac{\pi_1}{\pi_{detect}}$. Similarly, the probability for pass 2 becomes $\frac{\pi_2}{\pi_{detect}}$ and so on.

So our full detection model can now be written as:

$$p_i = \frac{exp(\alpha_0 + \alpha_1 StreamDepth_i)}{exp(\alpha_0 + \alpha_1 StreamDepth_i) + 1}$$
$$\pi_{i1} = p_i$$
$$\pi_{i2} = (1 - p_i)(p_i)$$
$$\pi_{i3} = (1 - p_i)^2(p_i)$$
$$\pi_{i4} = (1 - p_i)^3(p_i)$$
$$\pi_{i5} = (1 - p_i)^4$$
$$n_i \sim \text{Bin}(N_i, (1 - \pi_{i5}))y_{i1:4} \sim \text{Multinomial}(n_i, \{\frac{\pi_{i1}}{1 - \pi_{i5}}, \frac{\pi_{i2}}{1 - \pi_{i5}}, \frac{\pi_{i3}}{1 - \pi_{i5}}, \frac{\pi_{i4}}{1 - \pi_{i5}}\})$$

where p_i is the probability of detecting a fish at site i, π_{i1} to π_{i4} are the probability of detecting a fish at site i in pass 1 to 4, π_{i5} is the probability of never detecting a fish, y_{it} is the number of fish detected at site i in pass t, n_i is total number of fish detected at site i and N_i represents the realized abundance of fish at site i.

This can take a little bit of time to wrap your head around!

Butterflies - Dependent Double Observer

Double observer sampling uses some of the same logic as removal sampling but has fewer "categories" or multinomial bins. Let's say the first observer has detection probability p_{i1} at site i and the second observer has detection p_{i2}. Since the second observer sees everything the first person does (in practice this is done with verbal cues or physical cues), there are only three detection options.

If the first observer sees the individual:

$$\pi_{i1} = p_{i1}$$

Note that it doesn't matter for the second observer saw it or not - if the first person saw it, they will tell the second observer.

If only the second observer sees an individual then we know that observer one also has to *not* see it.

$$\pi_{i2} = (1 - p_{i1}) * p_{i2}$$

And if the individual was never detected then both observers had to not see it:

$$\pi_{i3} = (1 - p_{i1}) * (1 - p_{i2})$$

However just like in removal sampling, we don't know how many individuals we didn't see! So we must return to conditional probability and using n instead of N in our multinomial. Once again I'm setting up the two detection probabilites as linear models on the logit scale to constrain them between 0 and 1.

$$p_{i1} = \frac{exp(\alpha_{01} + \alpha_{11} Tempi)}{exp(\alpha_{01} + \alpha_{11} Tempi) + 1}$$
$$p_{i2} = \frac{exp(\alpha_{02} + \alpha_{12} Tempi)}{exp(\alpha_{02} + \alpha_{12} Tempi) + 1}$$
$$\pi_{i1} = p_{i1}$$
$$\pi_{i2} = (1 - p_{i1}) * p_{i2}$$
$$\pi_{i3} = (1 - p_{i1}) * (1 - p_{i2})$$
$$n_i \sim \text{Bin}(N_i, (1 - \pi_{i3}))y_{i1:2} \sim \text{Multinomial}(n_i, \{\frac{\pi_{i1}}{1 - \pi_{i3}}, \frac{\pi_{i2}}{1 - \pi_{i3}}\})$$

Note that the transition from this dependent observer model to the independent model is very simple - we would simply split the first π into two based on if the second observer also saw the individual or not ($p_{i1} * (1 - p_{i2})$ and $p_{i1} * p_{i2}$).

Coding a Closed N-Mixture Model in JAGS

So now that we have the formulas for our models, coding is easy right? Ha ha ha. Over time I've found it actually *does* get more intuitive but when I first started out the math to code relationship made very little sense. So let's walk through it!

Bird Point Counts

As always, let's start with our easiest example - sparrows. First, let's code the state process part.

As a reminder, this was the relationship we described above:

$$\log(\lambda_i) = \beta_0 + \beta_1 tree_i$$
$$N_i \sim \text{Poisson}(\lambda_i)$$

We know λ and N are going to vary with site, so immediately we know we need a for loop. Then we can stick the equation above into our code. Remember that "dpois" just mean "distribution - poisson"

```
modelstring.birds = "
  model
{
for (i in 1:n.sites){
  log(lambda[i]) <- beta0 + beta1*tree[i]
  N[i] ~ dpois(lambda[i])
}
```

Okay, awesome! We will supply the tree information as data to our model. But what about beta0 and beta1? We'll give them some nice vague priors.

```
modelstring.birds = "
  model
{
for (i in 1:n.sites){
  log(lambda[i]) <- beta0 + beta1*tree[i]
  N[i] ~ dpois(lambda[i])
}

beta0 ~ dunif(-3,3)
beta1 ~ dunif(-2,2)
```

Side note: How do we choose priors? There's no real rule, but one way to consider is to think about what the numbers do in the equation. For instance, if tree cover is 0 then we're saying that $log(lambda) = beta0$. This means $.05 \leq lambda \leq 20.09$. And if tree cover is 1 (100%), $log(lambda) = beta0 + beta1 * 1$, which means $.01 \leq lambda \leq 148.4$. This is a pretty large range and quite likely overkill - we saw a maximum of 6 sparrows at any given plot!

Now we can add in the detection information, which we said followed a binomial draw. Since p varies by site and visit, we know we'll need to put it inside two loops.

$$p_{it} = \frac{\exp(\alpha_0 + \alpha_1 Wind_{it})}{\exp(\alpha_0 + \alpha_1 Wind_{it}) + 1}$$
$$y_{it} \sim \text{Bin}(N_i, p_{it})$$

```
modelstring.birds = "
  model
{
for (i in 1:n.sites){
  log(lambda[i]) <- beta0 + beta1*tree[i]
  N[i] ~ dpois(lambda[i])

  for (t in 1:n.visit){
  logit(p[i,t]) <- alpha0 + alpha1*wind[i,t]
  y[i,t] ~ dbin(p[i,t], N[i])
  }
}

beta0 ~ dunif(-3,3)
beta1 ~ dunif(-2,2)
```

And now we just need to put into some priors for alpha0 and alpha1 and we can start running our model!

```
modelstring.birds = "
  model
{
for (i in 1:n.sites){
  log(lambda[i]) <- beta0 + beta1*tree[i]
  N[i] ~ dpois(lambda[i])

  for (t in 1:n.visit){
  logit(p[i,t]) <- alpha0 + alpha1*wind[i,t]
  y[i,t] ~ dbin(p[i,t], N[i])
  }
}

beta0 ~ dunif(-3,3)
beta1 ~ dunif(-2,2)
alpha0 ~ dunif(-2,2)
alpha1 ~ dunif(-2,2)
}
"
```

We have to give jags a list of parameters we care about, data, initial values and all that good stuff. You can scroll back up to the examples section to see what the data look like if you need a refresher.

```
params.birds <- c("alpha0", "alpha1", "beta0", "beta1", "N")
jd.birds <- list(y = as.matrix(birds[,c("Visit1", "Visit2", "Visit3", "Visit4")]),
                 wind = as.matrix(birds[,c("Wind1", "Wind2", "Wind3", "Wind4")]),
                 tree = birds[,"PercentCover"],
                 n.sites = 30,
                 n.visit = 4)
ji.birbs <- function(){list(
  alpha0 = runif(1, 0, 1),
                            alpha1= runif(1),
```

```
                            beta0 = runif(1),
                            beta1 = runif(1, -1, 1),
  N = apply(jd.birds$y, 1, max)+2
  )}
#1, .05, .5, .6
jags.birds <- run.jags(model = modelstring.birds,
                       inits = ji.birbs,
                     monitor = params.birds,
                     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
```

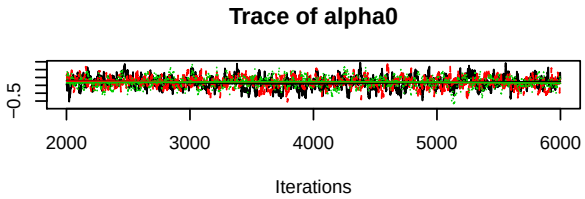
If you're struggling with initial values, providing different values of N is a good place to start. Above I gave N values as the maximum at each site plus 2, but you can provide any values you think might make sense.

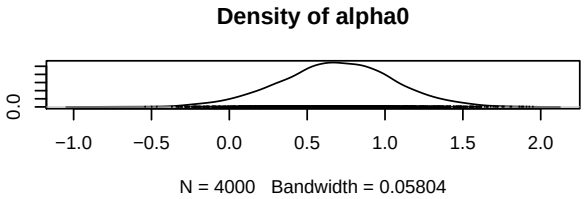
Let's check our results.

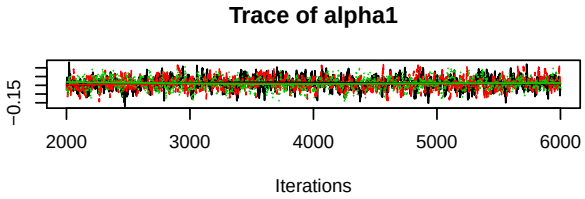
```
head(summary(jags.birds))
##           Lower95      Median   Upper95        Mean         SD Mode       MCerr
## alpha0 -0.0349409  0.69267100 1.4228700  0.68884498 0.36776971   NA 0.014993618
## alpha1 -0.1080080 -0.03873235 0.0279763 -0.03953752 0.03418012   NA 0.001175179
## beta0   0.0522222  0.59273050 1.0687900  0.58581471 0.26016571   NA 0.007933574
## beta1  -0.8342720  0.06165175 0.9152180  0.06029860 0.44629786   NA 0.012438965
## N[1]    2.0000000  2.00000000 3.0000000  2.15191667 0.39032126    2 0.006824039
## N[2]    2.0000000  2.00000000 3.0000000  2.21233333 0.46414708    2 0.008062679
##        MC%ofSD SSeff      AC.10      psrf
## alpha0     4.1   602 0.34797959 1.0015613
## alpha1     3.4   846 0.22631529 1.0005208
## beta0      3.0  1075 0.15445160 1.0012658
## beta1      2.8  1287 0.13778840 1.0015284
## N[1]       1.7  3272 0.04295645 1.0019120
## N[2]       1.7  3314 0.03322402 0.9998981
```

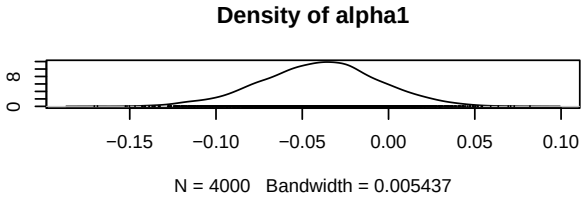
Next a visual inspection of the first few parameters.

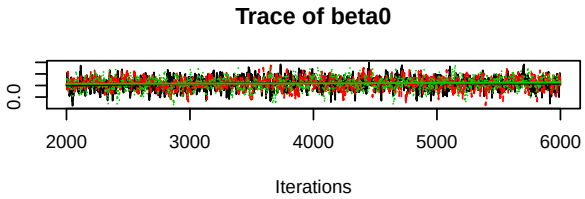
```
plot(jags.birds$mcmc[,c("alpha0", "alpha1", "beta0", "beta1", "N[1]"),])
```

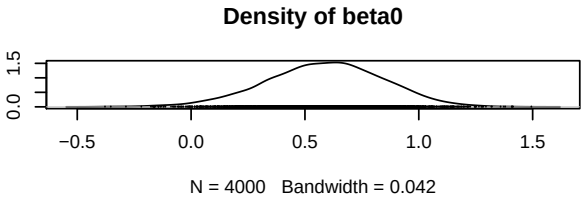


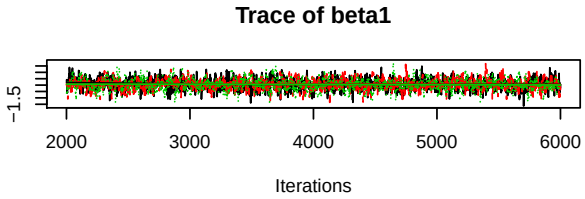


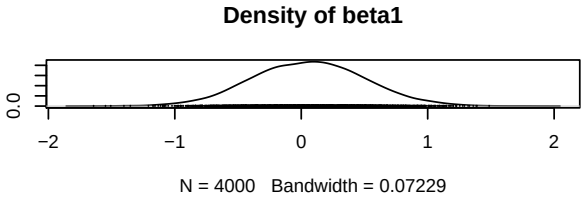


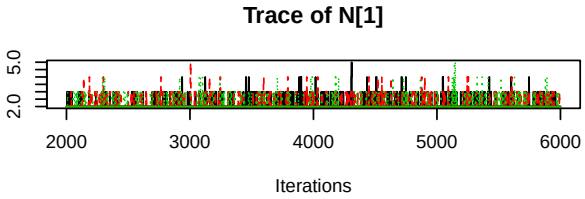


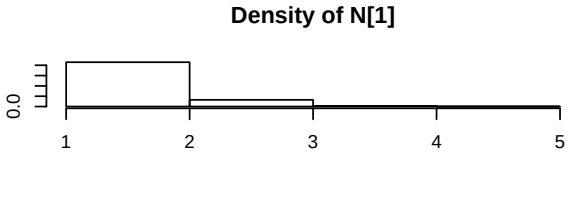












Both the visual inspection and the psrf suggest convergence! Since I simulated this data, we can also compare our estimates to the true values. Remember that N is the estimated abundance of the site, the beta values relate to covariates that influence abundance at a site and the alpha values tell you about detection.

	Simulation	Mean	LCI	UCI
alpha0	1.00	0.69	-0.03	1.42
alpha1	0.05	-0.04	-0.11	0.03
beta0	0.50	0.59	0.05	1.07
beta1	0.60	0.06	-0.83	0.92
N[1]	2.00	2.15	2.00	3.00
N[2]	2.00	2.21	2.00	3.00
N[3]	3.00	3.22	3.00	4.00
N[4]	1.00	1.06	1.00	2.00
N[5]	3.00	2.76	2.00	4.00
N[6]	3.00	2.53	2.00	4.00
N[7]	1.00	1.12	1.00	2.00
N[8]	4.00	3.20	3.00	4.00

	Simulation	Mean	LCI	UCI
N[9]	2.00	2.13	2.00	3.00
N[10]	1.00	1.06	1.00	2.00
N[11]	3.00	3.21	3.00	4.00
N[12]	0.00	0.07	0.00	1.00
N[13]	1.00	1.23	1.00	2.00
N[14]	2.00	2.93	2.00	4.00
N[15]	3.00	3.31	3.00	4.00
N[16]	2.00	2.30	2.00	3.00
N[17]	2.00	1.26	1.00	2.00
N[18]	3.00	2.27	2.00	3.00
N[19]	1.00	1.25	1.00	2.00
N[20]	3.00	3.30	3.00	4.00
N[21]	0.00	0.06	0.00	1.00
N[22]	1.00	1.43	1.00	3.00
N[23]	1.00	1.21	1.00	2.00
N[24]	4.00	4.38	4.00	6.00
N[25]	0.00	0.05	0.00	0.00
N[26]	2.00	2.47	2.00	4.00
N[27]	0.00	0.06	0.00	1.00
N[28]	3.00	3.37	3.00	5.00
N[29]	1.00	1.11	1.00	2.00
N[30]	0.00	0.08	0.00	1.00

In this case, our jags model did decently well but with all the variation in wind the model was a little with with our detection parameters (but still in the credible interval).

Electrofishing

In our electrofishing example we only had one covariate - stream depth - but we thought it influenced both detection (harder to catch fishies in deep pools) and abundance (more fishies are found in deep pools).

Our state process model was: As a reminder, this was the relationship we described above:

$$\log(\lambda_i) = \beta_0 + \beta_1 depth_i$$
$$N_i \sim \text{Poisson}(\lambda_i)$$

and our detection was modeled via:

$$p_i = \frac{exp(\alpha_0 + \alpha_1 StreamDepth_i)}{exp(\alpha_0 + \alpha_1 StreamDepth_i) + 1}$$
$$\pi_{i1} = p_i$$
$$\pi_{i2} = (1 - p_i)(p_i)$$
$$\pi_{i3} = (1 - p_i)^2(p_i)$$
$$\pi_{i4} = (1 - p_i)^3(p_i)$$
$$\pi_{i5} = (1 - p_i)^4$$
$$n_i \sim \text{Bin}(N_i, (1 - \pi_{i5}))y_{i1:4} \sim \text{Multinomial}(n_i, \{\frac{\pi_{i1}}{1 - \pi_{i5}}, \frac{\pi_{i2}}{1 - \pi_{i5}}, \frac{\pi_{i3}}{1 - \pi_{i5}}, \frac{\pi_{i4}}{1 - \pi_{i5}}\})$$

We know that λ and N will vary by site, as will p and π, so in this case we'll only need one for loop. In JAGS you can use the pow() function to raise values to the power of another value. So $pow((1 - p[i]), 4) = (1 - p[i])^4$.

```
modelstring.fish = "
  model
{
for (i in 1:n.sites){
  log(lambda[i]) <- beta0 + beta1*depth[i]
  N[i] ~ dpois(lambda[i])

  logit(p[i]) <- alpha0 + alpha1*depth[i]
  pi[i,1] <- p[i]
  pi[i,2] <- (1-p[i])*(p[i])
  pi[i,3] <- pow((1-p[i]), 2)*(p[i])
  pi[i,4] <- pow((1-p[i]), 3)*(p[i])
  pi[i,5] <- pow((1-p[i]), 4)
  n[i] ~ dbin((1-pi[i,5]), N[i])
  y[i,1:4] ~dmulti(pi[i,1:4]/(1-pi[i,5]), n[i]))
}
```

Now all that's left is to add priors and we can run the model!

```
modelstring.fish = "
  model
{
for (i in 1:n.sites){
  log(lambda[i]) <- beta0 + beta1*depth[i]
  N[i] ~ dpois(lambda[i])

  logit(p[i]) <- alpha0 + alpha1*depth[i]
  pi[i,1] <- p[i]
  pi[i,2] <- (1-p[i])*(p[i])
  pi[i,3] <- pow((1-p[i]), 2)*(p[i])
  pi[i,4] <- pow((1-p[i]), 3)*(p[i])
  pi[i,5] <- pow((1-p[i]), 4)
  n[i] ~ dbin((1-pi[i,5]), N[i])
  y[i,1:4] ~dmulti(pi[i,1:4]/(1-pi[i,5]), n[i])
}

beta0 ~ dunif(-3,3)
beta1 ~ dunif(-2,2)
alpha0 ~ dunif(-2,2)
alpha1 ~ dunif(-2,2)
}
"
```

Next we send the model to JAGS:

```
library(runjags)
params.fish <- c("alpha0", "alpha1", "beta0", "beta1", "N")
jd.fish <- list(y = as.matrix(fish[,c("pass1", "pass2", "pass3", "pass4")]),
                depth = fish[,"StreamDepth"],
                n.sites = 30,
               n = apply(fish[,c("pass1", "pass2", "pass3", "pass4")], 1, sum))
ji.fish <- function(){list(
  alpha0 = runif(1, -1, 1),
  alpha1= runif(1, -1, 0),
  beta0 = runif(1),
```

```
  beta1 = runif(1),
  N = jd.fish$n+2
)}

jags.fish <- run.jags(model = modelstring.fish,
                      inits = ji.fish,
                    monitor = params.fish,
                    data = jd.fish, n.chains = 3, burnin =  1000,
                    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
```

Notice that this time we also provided the model with n as data. Remember to give initial values of N if you're having trouble getting your model to initialize.

Now let's check our output.

```
head(summary(jags.fish))
##          Lower95     Median    Upper95       Mean         SD Mode       MCerr
## alpha0 -0.207684  0.7787015  1.6406700  0.7384958 0.48780424   NA 0.065525776
## alpha1 -0.623407 -0.3731845 -0.0852153 -0.3656177 0.13928648   NA 0.019808295
## beta0  -0.260926  0.3647290  0.9309860  0.3624105 0.29217122   NA 0.032394145
## beta1   0.434317  0.5962230  0.7811270  0.5988795 0.08473088   NA 0.010073096
## N[1]    9.000000  9.0000000 11.0000000  9.5850000 0.80304575    9 0.015699524
## N[2]    2.000000  2.0000000  3.0000000  2.1086667 0.34236105    2 0.009719443
##        MC%ofSD SSeff      AC.10     psrf
## alpha0    13.4    55 0.83151396 1.053652
## alpha1    14.2    49 0.84945458 1.069692
## beta0     11.1    81 0.77000412 1.063250
## beta1     11.9    71 0.79966881 1.100065
## N[1]       2.0  2616 0.04914484 1.003477
## N[2]       2.8  1241 0.10464121 1.002554
```

Hmm, it doesn't seem to have finished converging. Let's run it a little longer.

```
jags.fish <- extend.jags(jags.fish, sample = 5000, silent.jags = T)
## Calling 3 simulations using the parallel method...
## All chains have finished
## Simulation complete.  Reading coda files...
## Coda files loaded successfully
## Finished running the simulation
```

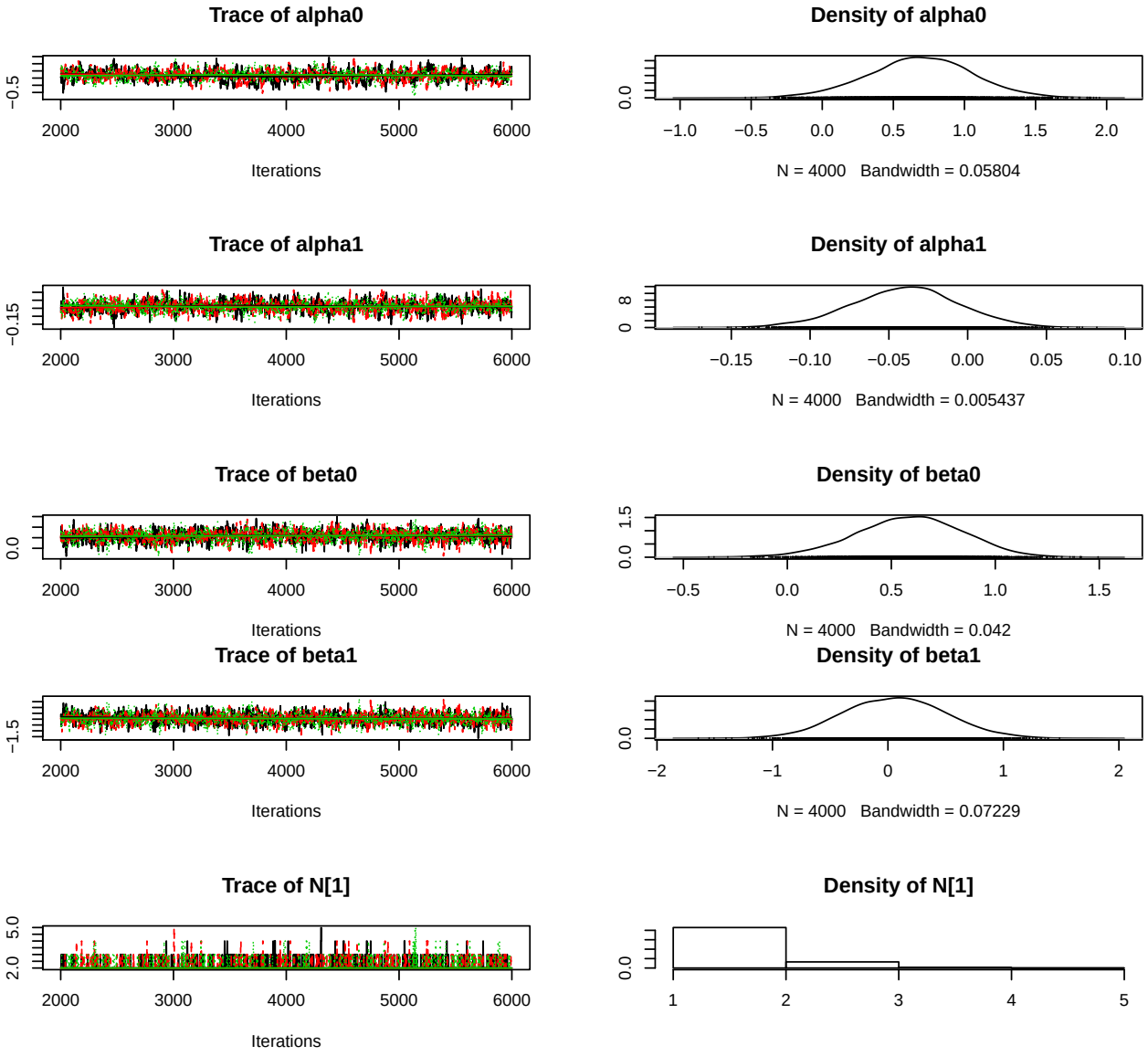
Check convergence again.

```
head(summary(jags.fish))
##          Lower95     Median     Upper95       Mean         SD Mode       MCerr
## alpha0 -0.545128  0.6072075  1.65394000  0.5826742 0.56866772   NA 0.048656865
## alpha1 -0.637011 -0.3252920 -0.00905285 -0.3192927 0.16001540   NA 0.014553286
## beta0  -0.133305  0.4409190  1.04023000  0.4374302 0.29669748   NA 0.019560729
```

```
## beta1   0.406026  0.5753695  0.73918100  0.5767262 0.08463301   NA 0.005915543
## N[1]    9.000000  9.0000000 11.00000000  9.6653810 0.87402724    9 0.016164194
## N[2]    2.000000  2.0000000  3.00000000  2.1627143 0.44235307    2 0.014069341
##         MC%ofSD SSeff      AC.10     psrf
## alpha0      8.6   137 0.88174367 1.006879
## alpha1      9.1   121 0.89536978 1.009086
## beta0       6.6   230 0.79984236 1.009111
## beta1       7.0   205 0.82548074 1.010504
## N[1]        1.8  2924 0.08390745 1.000196
## N[2]        3.2   989 0.15096578 1.000225
```

Looks like convergence has been reached according to the r-hat diagnostic (psrf), but let's do a visual inspection too.

```
plot(jags.birds$mcmc[,c("alpha0", "alpha1", "beta0", "beta1", "N[1]"),])
```



Looks pretty good!

As before, we can compare with the simulated data set to see how well we estimated the parameters.

Remember that N is the estimated abundance of the site, the beta values relate to covariates that influence abundance at a site and the alpha values tell you about detection.

	Simulation	Mean	LCI	UCI
alpha0	0.0	0.58	-0.55	1.65
alpha1	0.3	-0.32	-0.64	-0.01
beta0	0.5	0.44	-0.13	1.04
beta1	0.6	0.58	0.41	0.74
N[1]	9.0	9.67	9.00	11.00
N[2]	2.0	2.16	2.00	3.00
N[3]	6.0	5.63	5.00	7.00
N[4]	6.0	4.35	4.00	6.00
N[5]	12.0	11.81	10.00	14.00
N[6]	14.0	9.91	8.00	13.00
N[7]	35.0	27.26	21.00	34.00
N[8]	21.0	20.03	17.00	24.00
N[9]	9.0	7.74	7.00	9.00
N[10]	2.0	1.19	1.00	2.00
N[11]	15.0	12.29	11.00	15.00
N[12]	13.0	11.70	10.00	14.00
N[13]	15.0	10.61	9.00	13.00
N[14]	7.0	6.17	4.00	9.00
N[15]	31.0	24.72	17.00	33.00
N[16]	13.0	10.11	8.00	13.00
N[17]	7.0	6.51	6.00	8.00
N[18]	25.0	18.10	13.00	24.00
N[19]	8.0	8.60	8.00	10.00
N[20]	17.0	15.52	12.00	20.00
N[21]	26.0	22.94	18.00	28.00
N[22]	10.0	9.66	8.00	12.00
N[23]	5.0	3.32	3.00	5.00
N[24]	30.0	19.00	16.00	23.00
N[25]	22.0	20.79	17.00	25.00
N[26]	16.0	14.66	12.00	18.00
N[27]	4.0	3.18	3.00	4.00
N[28]	10.0	8.00	7.00	10.00
N[29]	17.0	12.30	7.00	18.00
N[30]	12.0	10.61	9.00	13.00

Alternative Removal Model Specification

There are actually two other ways you can write the removal model that will also run in JAGS.

The first way is to think of each pass as a binomial draw of detection, with the "trial size" decreasing each time.

We could think of the four passes as:

$$y_{i1} \sim \text{Bin}(N_i, p_i) y_{i2} \sim \text{Bin}(N_i - y_{i1}, p_i) y_{i3} \sim \text{Bin}(N_i - y_{i1} - y_{i2}, p_i) y_{i4} \sim \text{Bin}(N_i - y_{i1} - y_{i2} - y_{i3}, p_i)$$

This method works, but it doesn't mix as well in JAGS and I think it's less fun.

But should you want some code for it, the model looks like this:

```
  modelstring.fish2 = "
  model
{
for (i in 1:n.sites){
  log(lambda[i]) <- beta0 + beta1*depth[i]
  N[i] ~ dpois(lambda[i])
  logit(p[i]) <- alpha0 + alpha1*depth[i]

  y[i,1] ~ dbin(p[i], N[i])
  y[i,2] ~ dbin(p[i], N[i]-y[i,1])
  y[i,3] ~ dbin(p[i], N[i]-y[i,1]-y[i,2])
}

beta0 ~ dunif(-3,3)
beta1 ~ dunif(-2,2)
alpha0 ~ dunif(-2,2)
alpha1 ~ dunif(-2,2)
}
"
```

The other option only works if you draw N_i from a poisson as we have in this tutorial. This method will mix a lot faster in JAGS, so I think it's kind of nifty.

It turns out that through some fun math, you can transform this:

$$
\begin{aligned}
N_i &\sim \text{Poisson}(\lambda_i) \\
y_{i1} &\sim \text{Bin}(N_i, \pi_{i1}) \\
\pi_{i1} &= p_i
\end{aligned}
$$

Into:

$$
\begin{aligned}
&\text{Poisson}(\lambda_i \pi_{i1}) \\
&\pi_{i1} = p_i
\end{aligned}
$$

I tried to think of a fun, non-confusing way to explain this, but it ended up being more confusing. It doesn't really matter *why* this works, but the point is we could alternatively write our JAGS model as:

```
modelstring.fish3 = "
  model
{
for (i in 1:n.sites){
  log(lambda[i]) <- beta0 + beta1*depth[i]
  logit(p[i]) <- alpha0 + alpha1*depth[i]

  pi[i,1] <- p[i]
  pi[i,2] <- (1-p[i])*p[i]
  pi[i,3] <- (1-p[i])^2*p[i]
  y[i,1] ~ dpois(lambda[i]*pi[i,1])
  y[i,2] ~ dpois(lambda[i]*pi[i,2])
  y[i,3] ~ dpois(lambda[i]*pi[i,3])
}

beta0 ~ dunif(-3,3)
```

```
beta1 ~ dunif(-2,2)
alpha0 ~ dunif(-2,2)
alpha1 ~ dunif(-2,2)
}
"
```

Anyway, just some options to consider!

Doubled Observed Butterflies

Time to code our butterfly model. As a refresher, we had modeled our butterflies with:

$$\log(\lambda_i) = \beta_0 + \beta_1 flower_i$$
$$N_i \sim \text{Poisson}(\lambda_i)$$
$$p_{i1} = \frac{exp(\alpha_{01} + \alpha_{11} Tempi)}{exp(\alpha_{01} + \alpha_{11} Tempi) + 1}$$
$$p_{i2} = \frac{exp(\alpha_{02} + \alpha_{12} Tempi)}{exp(\alpha_{02} + \alpha_{12} Tempi) + 1}$$
$$\pi_{i1} = p_{i1}$$
$$\pi_{i2} = (1 - p_{i1}) * p_{i2}$$
$$\pi_{i3} = (1 - p_{i1}) * (1 - p_{i2})$$
$$n_i \sim \text{Bin}(N_i, (1 - \pi_{i3}))$$
$$y_{i1:2} \sim \text{Multinomial}(n_i, \{\frac{\pi_{i1}}{1 - \pi_{i3}}, \frac{\pi_{i2}}{1 - \pi_{i3}}\})$$

This code is almost identical to the electrofishing example, though the calculations for the π values are a little different. Note that we have no reason to believe that the relationship between detection and temperature is different between the observers, so the two linear equations for observer-specific detection will share a slope parameter.

```
modelstring.butterfly = "
  model
{
for (i in 1:n.sites){
  log(lambda[i]) <- beta0 + beta1*flowers[i]
  N[i] ~ dpois(lambda[i])

  logit(p1[i]) <- alpha0.a + alpha1*Temp[i]
  logit(p2[i]) <- alpha0.b + alpha1*Temp[i]
  pi[i,1] <- p1[i]
  pi[i,2] <- (1-p1[i])*(p2[i])
  pi[i,3] <- (1-p1[i])*(1-p2[i])
  n[i] ~ dbin((1-pi[i,3]), N[i])
  y[i,1:2] ~dmulti(pi[i,1:2]/(1-pi[i,3]), n[i])
}

beta0 ~ dunif(-3,3)
beta1 ~ dunif(-4,4)
alpha0.a ~ dunif(-2,2)
alpha0.b ~ dunif(-2,2)
alpha1 ~ dunif(-2,2)
```

```
}
"
```

Time to send the info to JAGS.

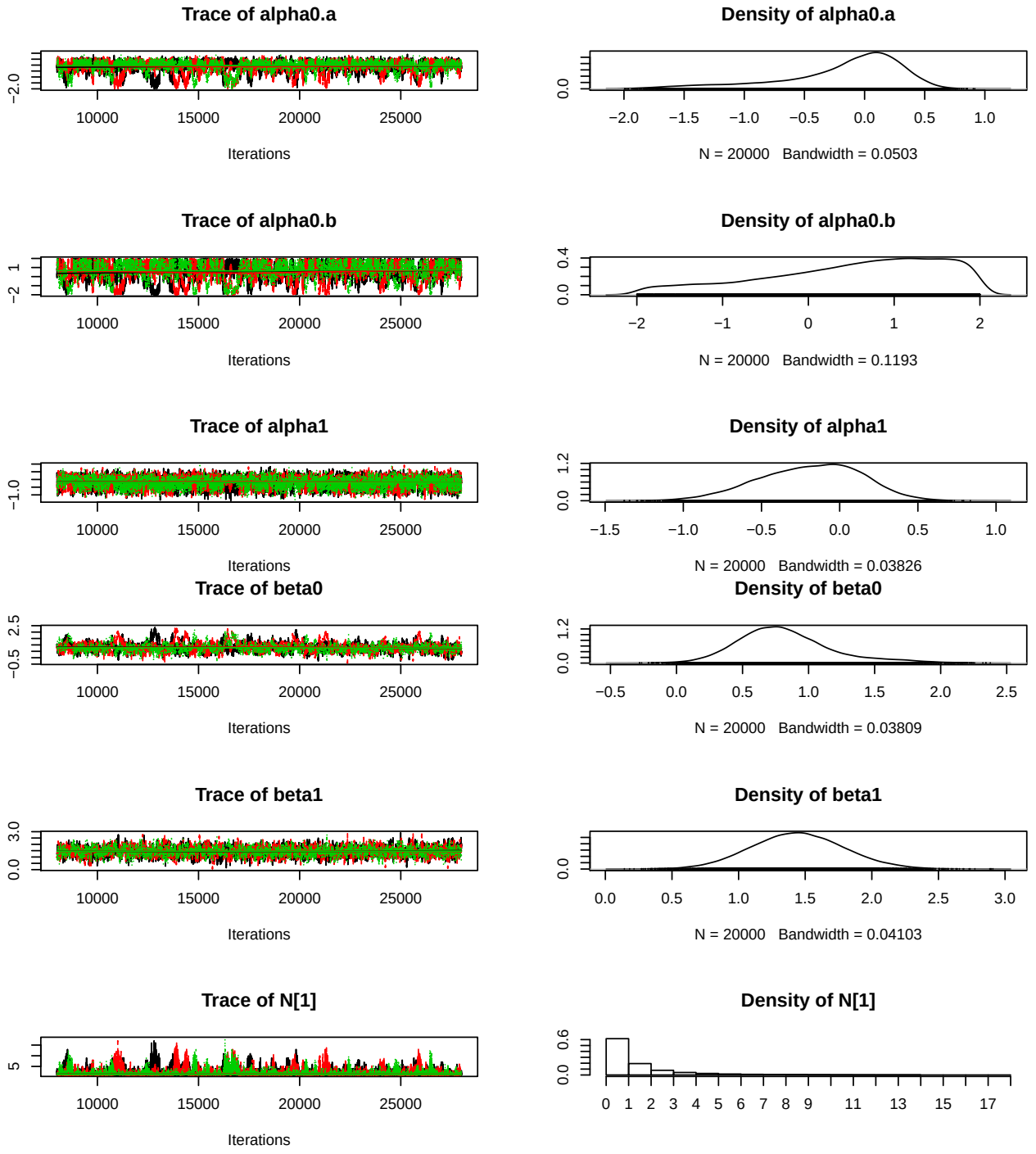
```
params.butterfly <- c("alpha0.a","alpha0.b", "alpha1", "beta0", "beta1", "N")
jd.butterfly <- list(y = as.matrix(butterflies[,c("BothDetect", "Obs2Only")]),
                 flowers = butterflies$Flowers,
                 Temp = butterflies$Temperature,
                 n.sites = 30,
                n = apply(butterflies[,c("BothDetect", "Obs2Only")], 1, sum))
ji.butterfly <- function(){list(
  alpha0.a = runif(1),
  alpha0.b = runif(1),
  alpha1= runif(1),
  beta0 = runif(1),
  beta1 = runif(1, 1, 2),
  N = abund.b+1
)}


jags.butterfly<- run.jags(model = modelstring.butterfly,
                      inits = ji.butterfly,
                    monitor = params.butterfly,
                    data = jd.butterfly, n.chains = 3, burnin =  7000,
                    sample = 20000, 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
```

Check our output like always

```
head(summary(jags.butterfly))
##            Lower95      Median  Upper95       Mean         SD Mode       MCerr
## alpha0.a -1.375750 -0.04865905 0.569000 -0.1766287 0.5067266   NA 0.026136379
## alpha0.b -1.465420  0.70061900 1.999950  0.5178891 1.0163835   NA 0.049047497
## alpha1   -0.846697 -0.16883600 0.400222 -0.1882582 0.3256519   NA 0.006386971
## beta0     0.142632  0.77879650 1.625940  0.8197025 0.3647564   NA 0.016807060
## beta1     0.767020  1.44246000 2.137800  1.4448935 0.3499433   NA 0.007346721
## N[1]      1.000000  1.00000000 5.000000  1.8582333 1.5681255    1 0.075711871
##          MC%ofSD SSeff     AC.10     psrf
## alpha0.a     5.2   376 0.7990589 1.007090
## alpha0.b     4.8   429 0.8016248 1.004717
## alpha1       2.0  2600 0.2334969 1.001257
## beta0        4.6   471 0.7494993 1.005077
## beta1        2.1  2269 0.4666792 1.000394
## N[1]         4.8   429 0.5870001 1.003910
```

And trace plots.

```
plot(jags.butterfly$mcmc[,c("alpha0.a","alpha0.b", "alpha1", "beta0", "beta1", "N[1]"),])
```



I find that this specific model really doesn't mix well unless you have a lot of sites, repeat site visits, or very few covariates on detection. But even without those things this model does a decent job. Let's compare to simulation. As with the other two models, N is the estimated abundance of the site, the beta values relate to covariates that influence abundance at a site and the alpha values tell you about detection (alpha0.a is observer 1, alpha0.b is observer 2).

	Simulation	Mean	LCI	UCI

	Simulation	Mean	LCI	UCI
alpha0.a	0.0	-0.18	-1.38	0.57
alpha0.b	0.2	0.52	-1.47	2.00
alpha1	0.1	-0.19	-0.85	0.40
beta0	0.5	0.82	0.14	1.63
beta1	2.0	1.44	0.77	2.14
N[1]	2.0	1.86	1.00	5.00
N[2]	7.0	7.04	5.00	14.00
N[3]	7.0	5.66	4.00	11.00
N[4]	1.0	1.73	1.00	4.00
N[5]	3.0	3.96	3.00	7.00
N[6]	4.0	5.27	3.00	11.00
N[7]	1.0	1.98	1.00	5.00
N[8]	5.0	6.55	4.00	16.00
N[9]	3.0	3.89	3.00	7.00
N[10]	3.0	3.61	2.00	9.00
N[11]	4.0	4.13	3.00	8.00
N[12]	8.0	7.04	4.00	17.00
N[13]	10.0	10.78	8.00	20.00
N[14]	3.0	3.04	2.00	6.00
N[15]	6.0	5.62	4.00	11.00
N[16]	7.0	8.68	6.00	16.00
N[17]	11.0	10.82	8.00	19.00
N[18]	5.0	5.49	4.00	10.00
N[19]	9.0	10.21	8.00	17.00
N[20]	6.0	6.58	5.00	12.00
N[21]	1.0	1.84	1.00	5.00
N[22]	1.0	2.71	1.00	7.00
N[23]	3.0	4.32	3.00	9.00
N[24]	3.0	2.65	1.00	9.00
N[25]	10.0	9.43	7.00	18.00
N[26]	9.0	10.65	9.00	16.00
N[27]	2.0	3.11	2.00	7.00
N[28]	8.0	6.42	5.00	11.00
N[29]	5.0	4.92	4.00	8.00
N[30]	6.0	7.43	5.00	15.00

Coding the Models in NIMBLE

Lucky for us, wwitching these models over to NIMBLE is pretty easy!

Bird Point Counts in NIMBLE

For our sparrow model, we really only have to alter 3 lines of our code to make it run in JAGS.

```
nimblebirds <- #no more modelstring
  nimbleCode({  #this line is v. important

for (i in 1:n.sites){
  log(lambda[i]) <- beta0 + beta1*tree[i]
```

```
  N[i] ~ dpois(lambda[i])

  for (t in 1:n.visit){
  logit(p[i,t]) <- alpha0 + alpha1*wind[i,t]
  y[i,t] ~ dbin(p[i,t], N[i])
  }
}

beta0 ~ dunif(-3,3)
beta1 ~ dunif(-2,2)
alpha0 ~ dunif(-2,2)
alpha1 ~ dunif(-2,2)
}) #close the brackets
```

Now we want to give the data, initials and parameters of interest to NIMBLE. NIMBLE likes you to split your data into constants (things that don't change) and data (things that *could* or do change). An easy rule of thumb is - if it comes from a distribution, it's data, otherwise put it in as a constant. Note that if it has NA's in it, it's also data.

The reason you have to make this split in NIMBLE but not in JAGS is that you could setup an entire model in NIMBLE with one set of data and then run it again with a different set of data without having to worry about recompiling! This isn't something I do frequently, but if you're interested, here's the NIMBLE help page on this topic: https://r-nimble.org/html_manual/cha-building-models.html

```
np.birds <- c("alpha0", "alpha1", "beta0", "beta1", "N")
nd.birds <- list(y = as.matrix(birds[,c("Visit1", "Visit2", "Visit3", "Visit4")]),
                 wind = as.matrix(birds[,c("Wind1", "Wind2", "Wind3", "Wind4")]),
                 tree = birds[,"PercentCover"])
nc.birds <- list(n.sites = 30,
                 n.visit = 4)
ni.birds <- list(
  alpha0 = runif(1, 0, 1),
  alpha1= runif(1),
  beta0 = runif(1),
  beta1 = runif(1, -1, 1),
  N = apply(jd.birds$y, 1, max)+2
  )
```

Alright, time to run! You can run code in NIMBLE multiple ways, so let's start with the "less customizable but easier" method.

```
library(nimble)
library(coda)
Birds.mod.nimble <- nimbleMCMC(code = nimblebirds,
                  constants = nc.birds,
                  data = nd.birds, inits = ni.birds, monitors = np.birds,
                  niter = 10000, thin = 1, nchains =3,
                  nburnin = 1000, samplesAsCodaMCMC = TRUE)
## 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...
## checking model calculations...
## model building finished.
```

```
## compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ compilation details.
## compilation finished.
## running chain 1...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## running chain 2...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## running chain 3...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
```

You could alternatively run NIMBLE in a lot of separate steps - useful for error checking, adjusting the MCMC, running the same model with different data, etc.

```
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...
## model building finished.
  prepbirds$initializeInfo()
## All model variables are initialized.
  mcmcbirds <- configureMCMC(prepbirds, monitors = np.birds, print = T )
## ===== Monitors =====
## thin = 1: alpha0, alpha1, beta0, beta1, N
## ===== Samplers =====
## slice sampler (30)
##   - N[]  (30 elements)
## RW sampler (4)
##   - beta0
##   - beta1
##   - alpha0
##   - alpha1
  birdsMCMC <- buildMCMC(mcmcbirds) #actually build the code for those samplers
  Cmodel <- compileNimble(prepbirds) #compiling the model itself in C++;
## compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ compilation details.
## compilation finished.
  Compbirds <- compileNimble(birdsMCMC, project = prepbirds) # compile the samplers next
## compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ compilation details.
## compilation finished.
  Birds.mod.nimble <- runMCMC(Compbirds, niter = 10000,
                              thin = 1, nburnin = 1000,
                              nchains = 3,
                              samplesAsCodaMCMC = TRUE)
## running chain 1...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## running chain 2...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## running chain 3...
```

```
## |-------------|-------------|-------------|-------------|
## |-----------------------------------------------------|
```

Awesome, let's check the results. The results in NIMBLE can be processed through the coda package, which gives you two lists of results - mean and sd in the first and quantiles in the second list.

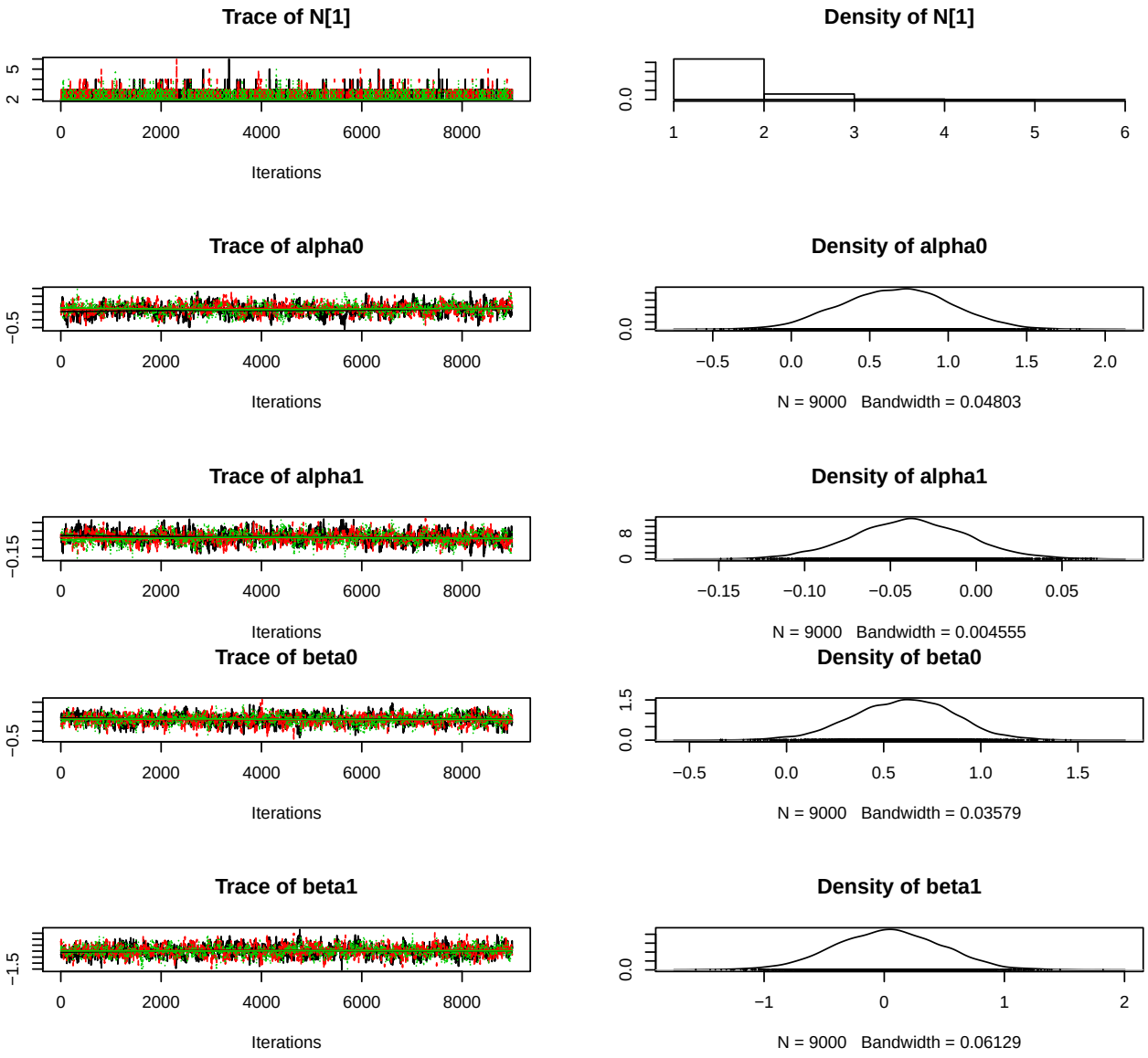
```
summary(Birds.mod.nimble)$statistics[c("N[1]", "alpha0", "alpha1", "beta0", "beta1"),]
##               Mean         SD     Naive SE Time-series SE
## N[1]    2.13744444 0.37570106 0.0022864439    0.003939013
## alpha0  0.66824433 0.34875721 0.0021224688    0.013355436
## alpha1 -0.03836108 0.03322234 0.0002021847    0.001138764
## beta0   0.60239362 0.25988147 0.0015815883    0.008553815
## beta1   0.03391417 0.44502764 0.0027083520    0.013579718
summary(Birds.mod.nimble)$quantiles[c("N[1]", "alpha0", "alpha1", "beta0", "beta1"),]
##                2.5%         25%         50%         75%      97.5%
## N[1]    2.000000000  2.00000000  2.00000000  2.00000000 3.00000000
## alpha0 -0.004240041  0.43017222  0.67443795  0.90553273 1.34632597
## alpha1 -0.103658502 -0.06044559 -0.03845821 -0.01613301 0.02801048
## beta0   0.080446775  0.42915353  0.61141503  0.78273025 1.10370334
## beta1  -0.844274807 -0.26873146  0.03588560  0.33745246 0.88433987
```

To evaluate convergence, we have to ask the coda package for the gelman-rubin diagnostic.

```
tail(gelman.diag(Birds.mod.nimble)$psrf)
##        Point est. Upper C.I.
## N[29]    1.003425   1.004709
## N[30]    1.009145   1.016420
## alpha0   1.013350   1.039437
## alpha1   1.004926   1.011795
## beta0    1.004227   1.012975
## beta1    1.002324   1.006245
```

And let's check a few plots as always.

```
plot(Birds.mod.nimble[,c("N[1]", "alpha0", "alpha1", "beta0", "beta1"),])
```



Looks like convergence is reached!

We can compare the results to the simulation and JAGS output, just for fun. Obviously we aren't expecting any difference between NIMBLE and JAGS, but just as proof:

	Simulation	JAGSMean	JAGSLCI	JAGSUCI	NIMBLEMean	NIMBLELCI	NIMBLEUCI
alpha0	1.00	0.69	-0.03	1.42	0.67	0.00	1.35
alpha1	0.05	-0.04	-0.11	0.03	-0.04	-0.10	0.03
beta0	0.50	0.59	0.05	1.07	0.60	0.08	1.10
beta1	0.60	0.06	-0.83	0.92	0.03	-0.84	0.88
N[1]	2.00	2.15	2.00	3.00	2.14	2.00	3.00
N[2]	2.00	2.21	2.00	3.00	2.21	2.00	3.00
N[3]	3.00	3.22	3.00	4.00	3.22	3.00	4.00
N[4]	1.00	1.06	1.00	2.00	1.06	1.00	2.00
N[5]	3.00	2.76	2.00	4.00	2.77	2.00	4.00
N[6]	3.00	2.53	2.00	4.00	2.55	2.00	4.00
N[7]	1.00	1.12	1.00	2.00	1.12	1.00	2.00

	Simulation	JAGSMean	JAGSLCI	JAGSUCI	NIMBLEMean	NIMBLELCI	NIMBLEUCI
N[8]	4.00	3.20	3.00	4.00	3.21	3.00	4.00
N[9]	2.00	2.13	2.00	3.00	2.14	2.00	3.00
N[10]	1.00	1.06	1.00	2.00	1.06	1.00	2.00
N[11]	3.00	3.21	3.00	4.00	3.21	3.00	4.00
N[12]	0.00	0.07	0.00	1.00	0.09	0.00	1.00
N[13]	1.00	1.23	1.00	2.00	1.24	1.00	2.00
N[14]	2.00	2.93	2.00	4.00	2.94	2.00	5.00
N[15]	3.00	3.31	3.00	4.00	3.31	3.00	5.00
N[16]	2.00	2.30	2.00	3.00	2.32	2.00	4.00
N[17]	2.00	1.26	1.00	2.00	1.28	1.00	2.00
N[18]	3.00	2.27	2.00	3.00	2.29	2.00	4.00
N[19]	1.00	1.25	1.00	2.00	1.27	1.00	3.00
N[20]	3.00	3.30	3.00	4.00	3.29	3.00	5.00
N[21]	0.00	0.06	0.00	1.00	0.06	0.00	1.00
N[22]	1.00	1.43	1.00	3.00	1.44	1.00	3.00
N[23]	1.00	1.21	1.00	2.00	1.21	1.00	2.00
N[24]	4.00	4.38	4.00	6.00	4.39	4.00	6.00
N[25]	0.00	0.05	0.00	0.00	0.05	0.00	1.00
N[26]	2.00	2.47	2.00	4.00	2.48	2.00	4.00
N[27]	0.00	0.06	0.00	1.00	0.07	0.00	1.00
N[28]	3.00	3.37	3.00	5.00	3.39	3.00	5.00
N[29]	1.00	1.11	1.00	2.00	1.10	1.00	2.00
N[30]	0.00	0.08	0.00	1.00	0.09	0.00	1.00

Electofishing Removal Sampling in NIMBLE

As before, not much changes between JAGS and NIMBLE when we run our removal sampling model. The big difference is that in NIMBLE you can't ask it to calculate the conditional probabilities inside the "dmulti" function. Instead we have to make this calculation in a separate line and use the new variable ($pi.cond[i, 1:4]$) inside the multinomial.

```
nimblefish <- #no more modelstring
  nimbleCode({  #this line is important
for (i in 1:n.sites){
  log(lambda[i]) <- beta0 + beta1*depth[i]
  N[i] ~ dpois(lambda[i])

  logit(p[i]) <- alpha0 + alpha1*depth[i]
  pi[i,1] <- p[i]
  pi[i,2] <- (1-p[i])*(p[i])
  pi[i,3] <- pow((1-p[i]), 2)*(p[i])
  pi[i,4] <- pow((1-p[i]), 3)*(p[i])
  pi[i,5] <- pow((1-p[i]), 4)
  n[i] ~ dbin((1-pi[i,5]), N[i])
  pi.cond[i,1:4] <- pi[i,1:4]/(1-pi[i,5]) #this line has to be added
  y[i,1:4] ~ dmulti(pi.cond[i,1:4], n[i])
}

beta0 ~ dunif(-3,3)
beta1 ~ dunif(-2,2)
alpha0 ~ dunif(-2,2)
alpha1 ~ dunif(-2,2)
```

```
})
```

Let's get our data in order to send to NIMBLE.

```
np.fish <- c("alpha0", "alpha1", "beta0", "beta1", "N")
nd.fish <- list(y = as.matrix(fish[,c("pass1", "pass2", "pass3", "pass4")]),
                depth = fish[,"StreamDepth"],
               n = apply(fish[,c("pass1", "pass2", "pass3", "pass4")], 1, sum))
nc.fish <- list(n.sites = 30)
ni.fish <- list(
  alpha0 = runif(1, -1, 1),
  alpha1= runif(1, -1, 0),
  beta0 = runif(1),
  beta1 = runif(1),
  N = nd.fish$n+2
)
```

Alright, time to run!

```
Fish.mod.nimble <- nimbleMCMC(code = nimblefish,
                  constants = nc.fish,
                  data = nd.fish, inits = ni.fish, monitors = np.fish,
                  niter = 8000, thin = 1, nchains =3,
                  nburnin = 1000, samplesAsCodaMCMC = TRUE)
## 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...
## checking model calculations...
## model building finished.
## compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ compilation details.
## compilation finished.
## running chain 1...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## running chain 2...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## running chain 3...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
```

Check results.

```
summary(Fish.mod.nimble)$statistics[c("N[1]", "alpha0", "alpha1", "beta0", "beta1"),]
##              Mean         SD    Naive SE Time-series SE
## N[1]    9.7139524 0.93277829 0.006436782     0.02597525
## alpha0  0.5225280 0.72086874 0.004974467     0.11966391
## alpha1 -0.3043279 0.20381952 0.001406488     0.03447023
## beta0   0.4533248 0.32887872 0.002269479     0.03681102
## beta1   0.5728347 0.09454667 0.000652434     0.01127431
summary(Fish.mod.nimble)$quantiles[c("N[1]", "alpha0", "alpha1", "beta0", "beta1"),]
##              2.5%         25%        50%        75%      97.5%
## N[1]    9.0000000  9.00000000  9.0000000 10.0000000 12.0000000
## alpha0 -0.9013636  0.03284259  0.6131684  1.0546205  1.7643627
```

```
## alpha1 -0.6813325 -0.44644543 -0.3288286 -0.1624458  0.0803947
## beta0  -0.1856661  0.23437797  0.4518066  0.6675802  1.0965483
## beta1   0.3868732  0.50914661  0.5706297  0.6345936  0.7577890
```

Got to check for convergence as always.

```
tail(gelman.diag(Fish.mod.nimble)$psrf)
##        Point est. Upper C.I.
## N[29]    1.033978   1.101389
## N[30]    1.001675   1.005655
## alpha0   1.043413   1.112833
## alpha1   1.053060   1.143323
## beta0    1.070397   1.215859
## beta1    1.081348   1.252406
```

Oh no, it didn't converge! Unlike in JAGS, you can't just "extend" a model that was run using the "nimbleMCMC" function. But luckily if we run our model the "long" way, we have that option!

First let's run the exact same thing that we did above, but the "long" way. We'll run it in parallel to allow for 3 chains to run at the same time (much faster).

```
library(parallel)
cl <- makeCluster(3)
clusterExport(cl = cl, varlist = c("nc.fish", "nd.fish", "ni.fish", "np.fish", "nimblefish"))
Fish.out <- clusterEvalQ(cl = cl,{
  library(nimble)
  library(coda)
  #you're now in a totally different environment so have to load packages again
  prepfish <- nimbleModel(code = nimblefish, constants = nc.fish,
                          data = nd.fish, inits = ni.fish)
  prepfish$initializeInfo()
  mcmcfish <- configureMCMC(prepfish, monitors = np.fish, print = T )
  fishMCMC <- buildMCMC(mcmcfish) #actually build the code for those samplers
  Cmodel <- compileNimble(prepfish) #compiling the model itself in C++;
  Compfish <- compileNimble(fishMCMC, project = prepfish) # compile the samplers next
  Compfish$run(nburnin = 1000, niter = 8000) #if you run this in your console it will say "null". Don't
  return(as.mcmc(as.matrix(Compfish$mvSamples)))
}) #this will take awhile and not produce any noticeable output.
Fish.mod.nimble <- mcmc.list(Fish.out)
```

Check our diagnostic as before:

```
tail(gelman.diag(Fish.mod.nimble)$psrf)
##        Point est. Upper C.I.
## N[29]    1.006426   1.016825
## N[30]    1.001149   1.003058
## alpha0   1.020861   1.062200
## alpha1   1.016824   1.052320
## beta0    1.048543   1.150565
## beta1    1.045654   1.139677
```

So now let's add more samples. Note that in NIMBLE you'll likely see higher sample numbers, but they'll run much faster than they do in JAGS.

```
out2 <- clusterEvalQ(cl, {
   Compfish$run(40000, reset = FALSE)
  return(as.mcmc(as.matrix(Compfish$mvSamples)))
```

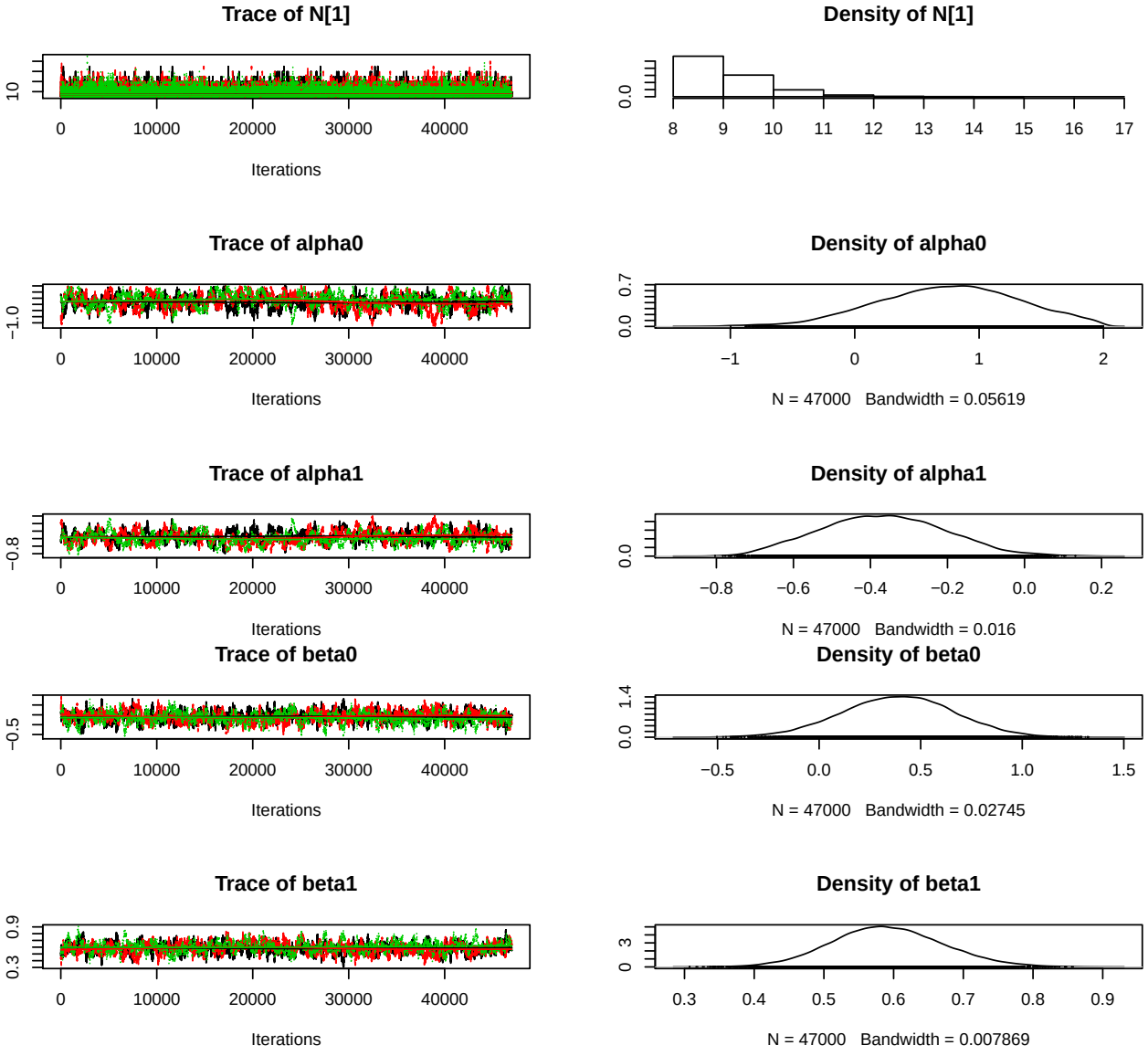
```
})
Fish.mod.nimble <- mcmc.list(out2)
```

Check diagnostic again:

```
tail(gelman.diag(Fish.mod.nimble)$psrf)
##        Point est. Upper C.I.
## N[29]    1.018582   1.058778
## N[30]    1.000417   1.000813
## alpha0   1.056740   1.177139
## alpha1   1.062124   1.194981
## beta0    1.059312   1.183934
## beta1    1.063508   1.196171
```

We could keep doing this as necessary until we are satisfied with convergence. Let's also look at the trace plots to see how the chains look.

```
plot(Fish.mod.nimble[,c("N[1]", "alpha0", "alpha1", "beta0", "beta1"),])
```



Once we're satisfied, we can look at the results of our model!

```
summary(Fish.mod.nimble)
##
## Iterations = 1:47000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 47000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean      SD  Naive SE Time-series SE
## N[1]    9.5950 0.81627 0.0021738       0.007033
## N[2]    2.1190 0.36952 0.0009841       0.004636
## N[3]    5.5599 0.79337 0.0021128       0.007052
## N[4]    4.2861 0.56876 0.0015147       0.006082
## N[5]   11.8054 1.46334 0.0038970       0.009289
## N[6]    9.9131 1.50720 0.0040139       0.009860
## N[7]   27.7775 3.82912 0.0101974       0.087675
## N[8]   20.1232 2.06156 0.0054902       0.021375
## N[9]    7.6759 0.86891 0.0023140       0.007038
## N[10]   1.1455 0.40686 0.0010835       0.004882
## N[11]  12.2437 1.18334 0.0031514       0.006914
## N[12]  11.6855 1.39956 0.0037272       0.008515
## N[13]  10.5945 1.35979 0.0036213       0.008175
## N[14]   6.1998 1.64288 0.0043752       0.011192
## N[15]  25.3986 4.66223 0.0124161       0.122214
## N[16]  10.1254 1.61152 0.0042917       0.010907
## N[17]   6.4560 0.71886 0.0019144       0.006960
## N[18]  18.4853 3.18889 0.0084924       0.058965
## N[19]   8.5283 0.76641 0.0020410       0.006765
## N[20]  15.6877 2.31872 0.0061750       0.029119
## N[21]  23.2876 3.09121 0.0082323       0.057131
## N[22]   9.6494 1.38565 0.0036901       0.008359
## N[23]   3.2623 0.54906 0.0014622       0.006270
## N[24]  19.1027 2.04947 0.0054580       0.021056
## N[25]  21.0090 2.47056 0.0065794       0.033292
## N[26]  14.7323 1.88388 0.0050170       0.016716
## N[27]   3.1367 0.39576 0.0010539       0.004915
## N[28]   7.9456 1.03316 0.0027514       0.006835
## N[29]  12.6865 3.29720 0.0087808       0.062626
## N[30]  10.5725 1.34569 0.0035837       0.008017
## alpha0  0.7511 0.56784 0.0015122       0.032018
## alpha1 -0.3670 0.16171 0.0004306       0.009736
## beta0   0.3873 0.27742 0.0007388       0.011100
## beta1   0.5919 0.07963 0.0002121       0.003390
##
## 2. Quantiles for each variable:
##
##           2.5%     25%     50%     75%    97.5%
## N[1]    9.0000  9.0000  9.0000 10.0000 12.00000
## N[2]    2.0000  2.0000  2.0000  2.0000  3.00000
## N[3]    5.0000  5.0000  5.0000  6.0000  8.00000
```

```
## N[4]    4.0000  4.0000  4.0000  4.0000  6.00000
## N[5]   10.0000 11.0000 12.0000 13.0000 15.00000
## N[6]    8.0000  9.0000 10.0000 11.0000 13.00000
## N[7]   22.0000 25.0000 27.0000 30.0000 37.00000
## N[8]   17.0000 19.0000 20.0000 21.0000 25.00000
## N[9]    7.0000  7.0000  7.0000  8.0000 10.00000
## N[10]   1.0000  1.0000  1.0000  1.0000  2.00000
## N[11]  11.0000 11.0000 12.0000 13.0000 15.00000
## N[12]  10.0000 11.0000 11.0000 12.0000 15.00000
## N[13]   9.0000 10.0000 10.0000 11.0000 14.00000
## N[14]   4.0000  5.0000  6.0000  7.0000 10.00000
## N[15]  19.0000 22.0000 25.0000 28.0000 36.00000
## N[16]   8.0000  9.0000 10.0000 11.0000 14.00000
## N[17]   6.0000  6.0000  6.0000  7.0000  8.00000
## N[18]  14.0000 16.0000 18.0000 20.0000 26.00000
## N[19]   8.0000  8.0000  8.0000  9.0000 10.00000
## N[20]  12.0000 14.0000 15.0000 17.0000 21.00000
## N[21]  19.0000 21.0000 23.0000 25.0000 31.00000
## N[22]   8.0000  9.0000  9.0000 10.0000 13.00000
## N[23]   3.0000  3.0000  3.0000  3.0000  5.00000
## N[24]  16.0000 18.0000 19.0000 20.0000 24.00000
## N[25]  17.0000 19.0000 21.0000 22.0000 27.00000
## N[26]  12.0000 13.0000 14.0000 16.0000 19.00000
## N[27]   3.0000  3.0000  3.0000  3.0000  4.00000
## N[28]   7.0000  7.0000  8.0000  8.0000 10.00000
## N[29]   8.0000 10.0000 12.0000 14.0000 20.00000
## N[30]   9.0000 10.0000 10.0000 11.0000 14.00000
## alpha0 -0.3908  0.3672  0.7698  1.1562  1.79139
## alpha1 -0.6676 -0.4819 -0.3693 -0.2558 -0.04537
## beta0  -0.1674  0.2011  0.3906  0.5766  0.92438
## beta1   0.4402  0.5376  0.5893  0.6442  0.75440
```

Butterfly Surveys (Double Observer) in NIMBLE

There really isn't a whole lot that has to change to run this in NIMBLE. Again, we will have to pre-calculate the multinomial probabilities, but that's about the only difference.

```
nimblebfly<- #no more modelstring
  nimbleCode({  #this line is new
for (i in 1:n.sites){
  log(lambda[i]) <- beta0 + beta1*flowers[i]
  N[i] ~ dpois(lambda[i])

  logit(p1[i]) <- alpha0.a + alpha1*Temp[i]
  logit(p2[i]) <- alpha0.b + alpha1*Temp[i]
  pi[i,1] <- p1[i]
  pi[i,2] <- (1-p1[i])*(p2[i])
  pi[i,3] <- (1-p1[i])*(1-p2[i])
  n[i] ~ dbin((1-pi[i,3]), N[i])
  pi.cond[i,1:2] <- pi[i,1:2]/(1-pi[i,3]) #new
  y[i,1:2] ~dmulti(pi.cond[i,1:2], n[i]) #this has to change
}

beta0 ~ dunif(-3,3)
```

```
beta1 ~ dunif(-4,4)
alpha0.a ~ dunif(-2,2)
alpha0.b ~ dunif(-2,2)
alpha1 ~ dunif(-2,2)
})
```

Time to send the info to NIMBLE.

```
np.butterfly <- c("alpha0.a","alpha0.b", "alpha1", "beta0", "beta1", "N")
nd.butterfly <- list(y = as.matrix(butterflies[,c("BothDetect", "Obs2Only")]),
                 flowers = butterflies$Flowers,
                 Temp = butterflies$Temperature,
                n = apply(butterflies[,c("BothDetect", "Obs2Only")], 1, sum))
nc.butterfly <- list(n.sites = 30)
ni.butterfly <- list(
  alpha0.a = runif(1),
  alpha0.b = runif(1),
  alpha1= runif(1),
  beta0 = runif(1),
  beta1 = runif(1, 1, 2),
  N = nd.butterfly$n+1
)
```

Alright, time to run!

```
Butterfly.mod.nimble <- nimbleMCMC(code = nimblebfly,
                  constants = nc.butterfly,
                  data = nd.butterfly, inits = ni.butterfly,
                  monitors = np.butterfly,
                  niter = 200000, thin = 1, nchains =3,
                  nburnin = 150000, samplesAsCodaMCMC = TRUE)
## 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...
## checking model calculations...
## model building finished.
## compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ compilation details.
## compilation finished.
## running chain 1...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## running chain 2...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## running chain 3...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
```

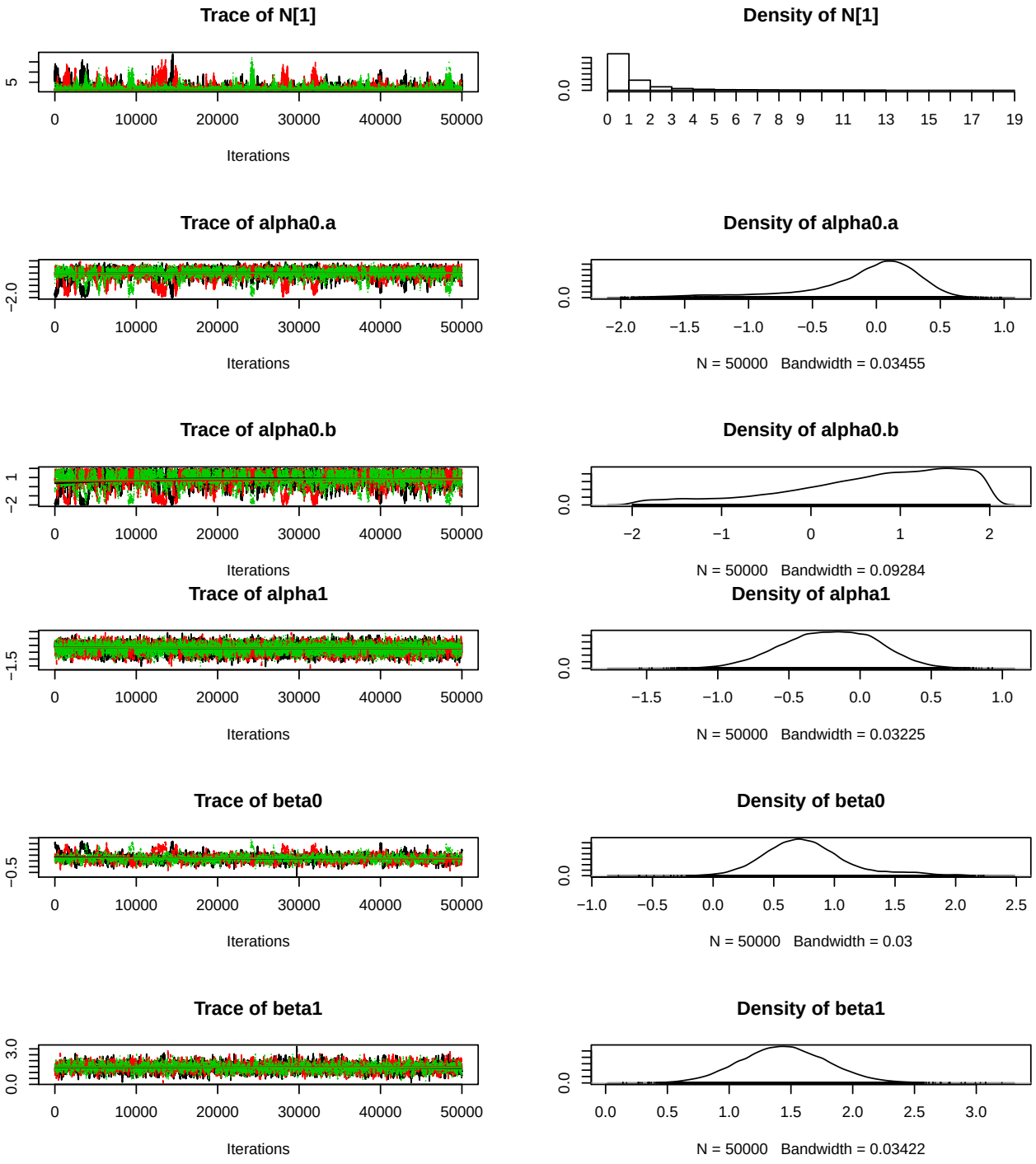
Got to check for convergence as always.

```
tail(gelman.diag(Butterfly.mod.nimble)$psrf)
##          Point est. Upper C.I.
## N[30]      1.027024   1.042256
## alpha0.a   1.011541   1.023173
```

```
## alpha0.b    1.006026    1.013552
## alpha1      1.000176    1.000602
## beta0       1.008139    1.014683
## beta1       1.004208    1.013707
```

And check our plots!

```
plot(Butterfly.mod.nimble[,c("N[1]", "alpha0.a","alpha0.b", "alpha1", "beta0", "beta1"),])
```



Similarly to our jags model, we can see it didn't really do the best job of mixing, but the estimates aren't too

far off the truth!

We can also look at our full results:

```
summary(Butterfly.mod.nimble)
##
## Iterations = 1:50000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 50000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##              Mean     SD  Naive SE Time-series SE
## N[1]      1.68507 1.3907 0.0035907       0.062957
## N[2]      6.60089 2.8078 0.0072498       0.138607
## N[3]      5.35760 2.1489 0.0055484       0.101549
## N[4]      1.59078 1.1877 0.0030667       0.049414
## N[5]      3.81501 1.3291 0.0034317       0.049054
## N[6]      4.90699 2.5474 0.0065774       0.107451
## N[7]      1.85341 1.2783 0.0033006       0.035053
## N[8]      5.97568 3.7431 0.0096646       0.168878
## N[9]      3.70039 1.4270 0.0036845       0.063469
## N[10]     3.27648 2.2729 0.0058685       0.110554
## N[11]     3.90784 1.7172 0.0044338       0.081638
## N[12]     6.44293 3.7275 0.0096245       0.183481
## N[13]    10.21889 3.4529 0.0089152       0.168686
## N[14]     2.86057 1.4159 0.0036558       0.054699
## N[15]     5.30193 2.1705 0.0056041       0.107499
## N[16]     8.20862 2.9758 0.0076834       0.134041
## N[17]    10.27173 3.3668 0.0086930       0.165472
## N[18]     5.22369 1.9988 0.0051608       0.095693
## N[19]     9.78219 2.7724 0.0071583       0.133788
## N[20]     6.28421 2.0988 0.0054192       0.100868
## N[21]     1.69237 1.2187 0.0031468       0.045299
## N[22]     2.43515 1.9767 0.0051039       0.074139
## N[23]     4.04849 1.8545 0.0047883       0.089911
## N[24]     2.29413 2.3433 0.0060504       0.114184
## N[25]     8.89725 3.2201 0.0083142       0.160282
## N[26]    10.31394 2.2319 0.0057627       0.110185
## N[27]     2.90733 1.5723 0.0040596       0.070674
## N[28]     6.14721 1.9236 0.0049668       0.093227
## N[29]     4.75084 1.3299 0.0034337       0.054523
## N[30]     6.95825 3.0409 0.0078516       0.148510
## alpha0.a -0.09399 0.4607 0.0011896       0.021346
## alpha0.b  0.70105 0.9499 0.0024525       0.037420
## alpha1   -0.20932 0.3300 0.0008520       0.004679
## beta0     0.76918 0.3520 0.0009089       0.012247
## beta1     1.44507 0.3501 0.0009040       0.007478
##
## 2. Quantiles for each variable:
##
##              2.5%     25%      50%      75%   97.5%
## N[1]      1.0000  1.0000  1.00000  2.00000  6.0000
```

```
## N[2]       5.0000  5.0000  6.00000  7.00000 15.0000
## N[3]       4.0000  4.0000  5.00000  6.00000 12.0000
## N[4]       1.0000  1.0000  1.00000  2.00000  5.0000
## N[5]       3.0000  3.0000  3.00000  4.00000  8.0000
## N[6]       3.0000  3.0000  4.00000  6.00000 12.0000
## N[7]       1.0000  1.0000  1.00000  2.00000  5.0000
## N[8]       4.0000  4.0000  4.00000  6.00000 18.0000
## N[9]       3.0000  3.0000  3.00000  4.00000  8.0000
## N[10]      2.0000  2.0000  2.00000  3.00000 10.0000
## N[11]      3.0000  3.0000  3.00000  4.00000  9.0000
## N[12]      4.0000  4.0000  5.00000  7.00000 18.0000
## N[13]      8.0000  8.0000  9.00000 11.00000 21.0000
## N[14]      2.0000  2.0000  2.00000  3.00000  7.0000
## N[15]      4.0000  4.0000  5.00000  6.00000 12.0000
## N[16]      6.0000  6.0000  7.00000  9.00000 17.0000
## N[17]      8.0000  8.0000  9.00000 11.00000 21.0000
## N[18]      4.0000  4.0000  5.00000  6.00000 11.0000
## N[19]      8.0000  8.0000  9.00000 10.00000 18.0000
## N[20]      5.0000  5.0000  6.00000  7.00000 13.0000
## N[21]      1.0000  1.0000  1.00000  2.00000  5.0000
## N[22]      1.0000  1.0000  2.00000  3.00000  8.0000
## N[23]      3.0000  3.0000  3.00000  4.00000 10.0000
## N[24]      1.0000  1.0000  1.00000  2.00000 10.0000
## N[25]      7.0000  7.0000  8.00000  9.00000 19.0000
## N[26]      9.0000  9.0000 10.00000 11.00000 17.0000
## N[27]      2.0000  2.0000  2.00000  3.00000  8.0000
## N[28]      5.0000  5.0000  5.00000  6.00000 12.0000
## N[29]      4.0000  4.0000  4.00000  5.00000  9.0000
## N[30]      5.0000  5.0000  6.00000  7.00000 16.0000
## alpha0.a -1.3959 -0.2650  0.01504  0.20872  0.5161
## alpha0.b -1.5995  0.1473  0.88511  1.46123  1.9464
## alpha1   -0.8629 -0.4377 -0.20045  0.03056  0.4033
## beta0     0.1738  0.5378  0.73486  0.94904  1.6576
## beta1     0.7725  1.2068  1.43975  1.67907  2.1466
```

And that's Closed Population N-Mixture Models! Things get a little more interesting with dynamic models, so stay tuned :)