Linear Regression Part 2

Heather Gaya

This tutorial aims to expand what we talked about in Part 1 about linear regression. Today we'll add in categorical variables and model selection. As before, I'll show you how to run it in both JAGS and NIMBLE and I'll go over how to graph the results with credible intervals!

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. For this tutorial I am using NIMBLE version 0.10.1

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

Contents

A Fake Scenario

As we saw in Part 1, Joe has gone out in the world and collected 50 frogs. He recorded each frog's age (in days), weight (in g), left back leg length (cm), the distance the animal was from the road (m) and what species the frog is (A or B). His data looks like this (but with 50 rows):

Frog	Age	Weight	Leg	Dist	Species
1	191	15.44	4.90	57.61	A
2	184	21.45	3.44	92.08	A
3	243	21.39	4.37	54.87	A
4	770	114.28	3.42	189.96	A
5	614	64.52	3.73	38.64	A
6	771	79.07	3.18	13.33	A

Previously, Joe was only interested in how weight was related to age, leg length and distance from the road, but now he wants to model the two species separately.

When we only used continuous variables, we could just multiply betas (β) by the variable itself to model the relationship. But "Species" isn't informative numerically - there's no order to Species A vs Species B - so we have to think about this a different way.

There are many options here for the actual equation, depending on what we think the relationship is with species - do we think the intercept is different? Maybe one frog grows faster than the other, so we would expect a difference in the relationship with age?

Before, we had:

$$E(weight) = \beta_0 + \beta_1 A + \beta_2 L + \beta_3 D$$
$$Actualweight \sim Normal(\mu = E(weight), \sigma = sd)$$

where A was age, L was leg length and D was distance to road.

Let's imagine a model where just the intercepts are different. Maybe this means we think the average weight of species B is always going to be heavier, but it will still have the same relationships with the other variables.

Conceptually, we might think of our expected weight equation as: E(weight) <- interceptA *(1 if species A, 0 if species B) + interceptB*(1 if species B, 0 if species A) + age X something + leg X something2 + distance X something3

In more mathy format,

$$E(weight) = \begin{cases} \text{species = A} & \beta_0^A + \beta_1 A + \beta_2 L + \beta_3 D \\ \text{species = B} & \beta_0^B + \beta_1 A + \beta_2 L + \beta_3 D \end{cases}$$
$$Actualweight \sim Normal(\mu = E(weight), \sigma = sd)$$

However in JAGS (or NIMBLE) we can't use this "ifelse" type statement. So we have to resort to a trick called "nested indexing".

A Quick Note on Nested Indexing

Nested indexing can be a little confusing at first glance, but it saves a ton of time and headache once you get used to the idea! We're used to thinking of variables as either one value (think, β_1) or a vector of lots of values (e.g. leg values of all frogs in our dataset) but variables can be all sorts of dimensions.

Let's say we have a variable s, a vector of length 50 that represents if the frog is species A or species B. 1's are species A and 2's are species B. Let's also say we have two intercept values, one for species A and the second for species B.

```
(s <- as.numeric(Frogs$Species))
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2
## [39] 2 2 2 2 2 2 2 2 2 2 2 2
fake.intercepts <- c(2, 5)
```

Normally in R, if we want the 5th individual's species, we would type s[5] and get out "1". And if we want the intercept for species A, we would type fake.intercepts[1] and get out "2". So we can conveniently put these together to say "hey, what's the intercept value for individual 5?". This is the joy of nested indexing.

```
s[5]
## [1] 1
fake.intercepts[1]
## [1] 2
fake.intercepts[s[5]]  # = fake.intercept[1]
## [1] 2
fake.intercepts[s[40]]  # = fake.intercept[2]
## [1] 5
```

Writing Model 1 JAGS

As before, we can save the model as a textstring in R to send to JAGS later. First, we start with the meat of the model - the equation we had up above (but now with nested indexing!). Also remember that JAGS requires the normal distribution be defined with mean and precision instead of standard deviation.

```
for (i in 1:n.frogs) {
    meanweight[i] <- beta0[s[i]] + age[i] * beta1 + leg[i] * beta2 + distance[i] *
        beta3
    weight[i] ~ dnorm(meanweight[i], prec)
}
```

Now let's get some priors in there and convert between precision and standard deviation

```
beta0[1] ~ dunif(-50, 50)
beta0[2] ~ dunif(-50, 50)
beta1 ~ dunif(-50, 50)
beta2 ~ dunif(-50, 50)
beta3 ~ dunif(-50, 50)

prec <- 1/(sd * sd)
sd ~ dunif(1e-04, 100)
```

And finally we stick it all together into one model!

```
modelstring.Frogs1 = "
  model
{
for (i in 1:n.frogs){
  meanweight[i] <- beta0[s[i]] + age[i]*beta1 + leg[i]*beta2 + distance[i]*beta3
    weight[i] ~ dnorm(meanweight[i], prec)
}

beta0[1] ~ dunif(-50,50)
beta0[2] ~ dunif(-50,50)
```

```
beta1 ~ dunif(-50,50)
beta2 ~ dunif(-50,50)
beta3 ~ dunif(-50,50)

prec <- 1/(sd *sd)
sd ~ dunif(0.0001, 100)
}
"
```

Time to send the model to JAGS. First we give the model parameters to monitor:

```
params <- c("beta0", "beta1", "beta2", "beta3", "sd")
```

And give JAGS the data we have on the frogs. We also need to add the new variable "s" that we created up above.

```
data <- list(weight = Frogs$Weight, distance = Frogs$Dist, age = Frogs$Age,
    leg = Frogs$Leg, n.frogs = 50, s = s)
```

Next initial values. For really simple models, you don't really have to provide these, since JAGS will try to pick its own. The big caveat is that sometimes JAGS picks terrible initial values and then throws weird errors, so that's something to be prepared for. Once you get into any kind of mixture models, I highly recommend using initial values. For now, we will skip them.

Finally we run the model and check the summary statistics.

```
library(runjags)
Frog.mod1 <- run.jags(model = modelstring.Frogs1, monitor = params, data = data,
    n.chains = 3, sample = 5000, method = "parallel")
## Warning: No initial values were provided - JAGS will use the same initial values
## for all chains
## Warning: You attempted to start parallel chains without setting different PRNG
## for each chain, which is not recommended. Different .RNG.name values have been
## added to each set of initial values.
## Calling 3 simulations using the parallel method...
## Following the progress of chain 1 (the program will wait for all chains
## to finish before continuing):
## Welcome to JAGS 4.3.0 on Tue Apr  6 10:46:04 2021
## JAGS is free software and comes with ABSOLUTELY NO WARRANTY
## Loading module: basemod: ok
## Loading module: bugs: ok
## . . Reading data file data.txt
## . Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 50
##    Unobserved stochastic nodes: 6
##    Total graph size: 463
## . Reading parameter file inits1.txt
## . Initializing model
## . Adapting 1000
## -------------------------------------------------| 1000
## +++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## Adaptation successful
## . Updating 4000
```

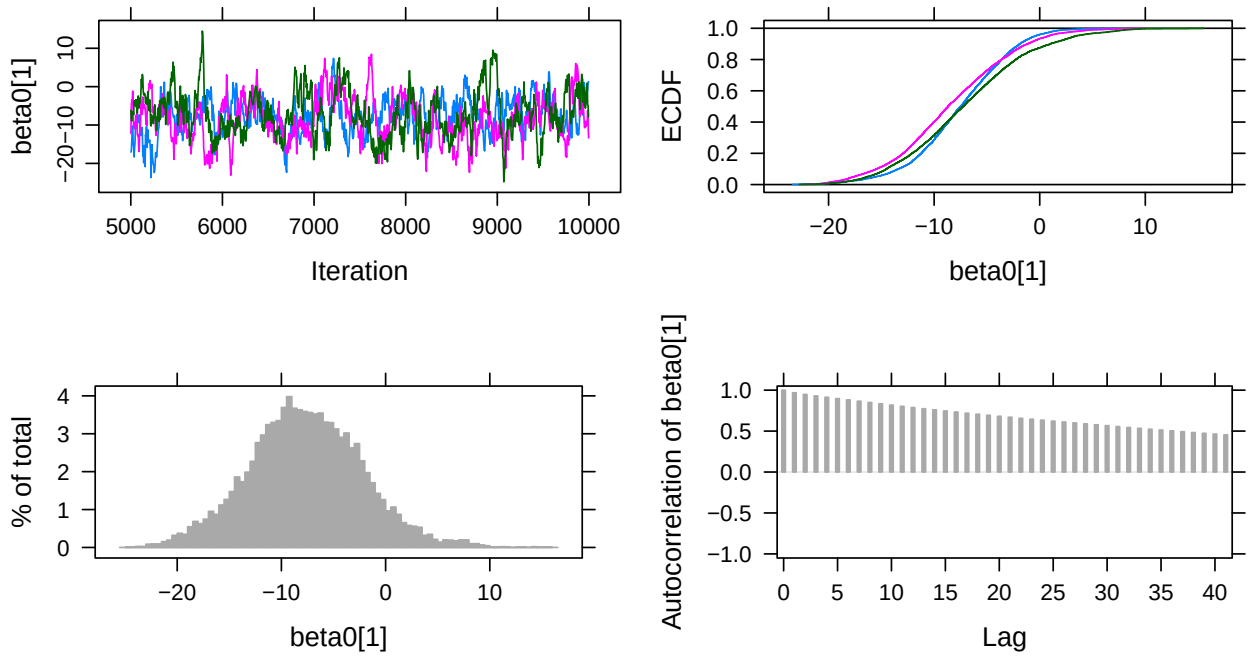
```
## ----------------------------------------------------| 4000
## ++++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## . . . . . Updating 5000
## ----------------------------------------------------| 5000
## ++++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## . . . . Updating 0
## . Deleting model
## .
## All chains have finished
## Simulation complete.  Reading coda files...
## Coda files loaded successfully
## Calculating summary statistics...
## Calculating the Gelman-Rubin statistic for 6 variables....
## Finished running the simulation
```

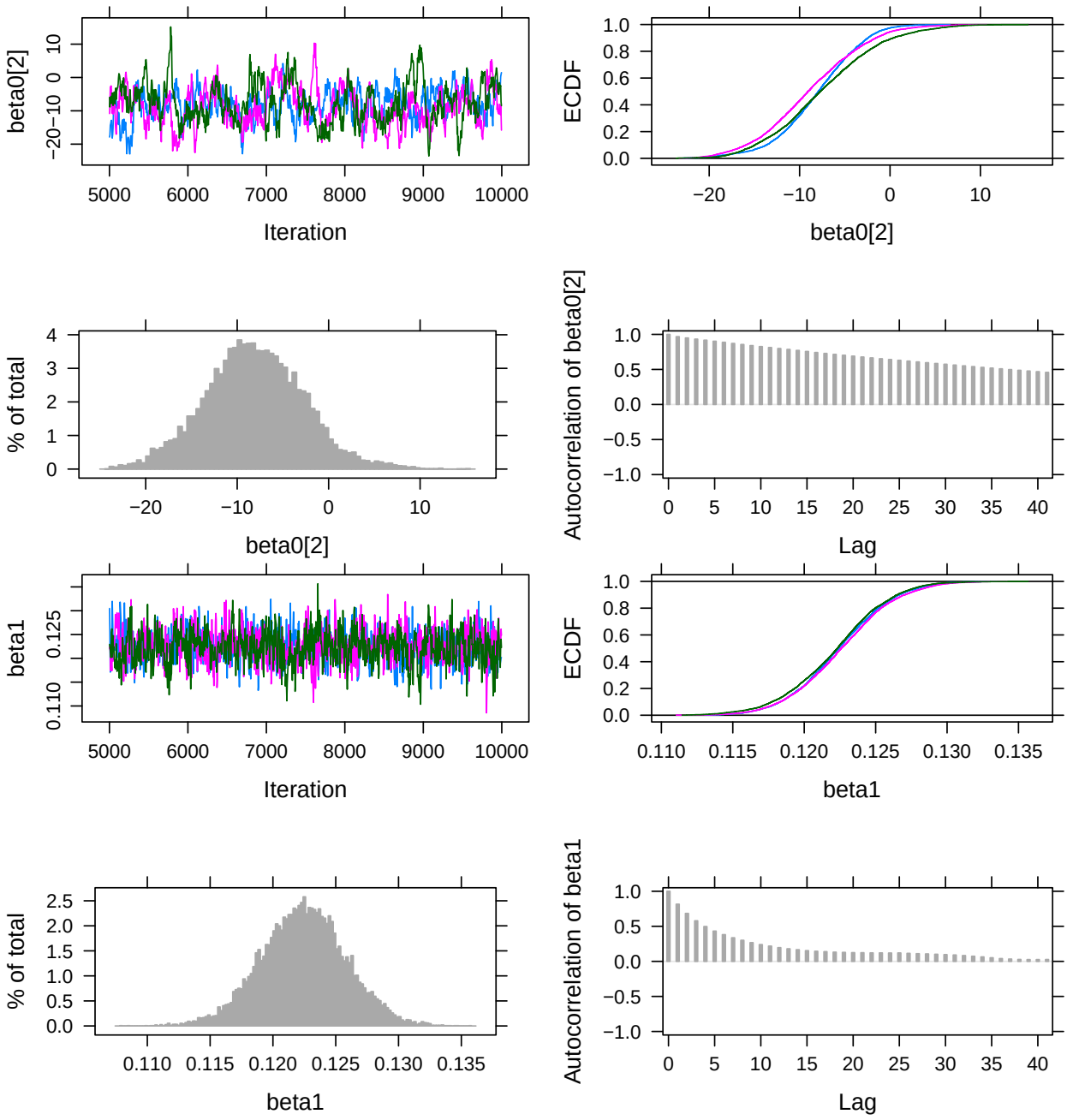
```
summary(Frog.mod1)
```

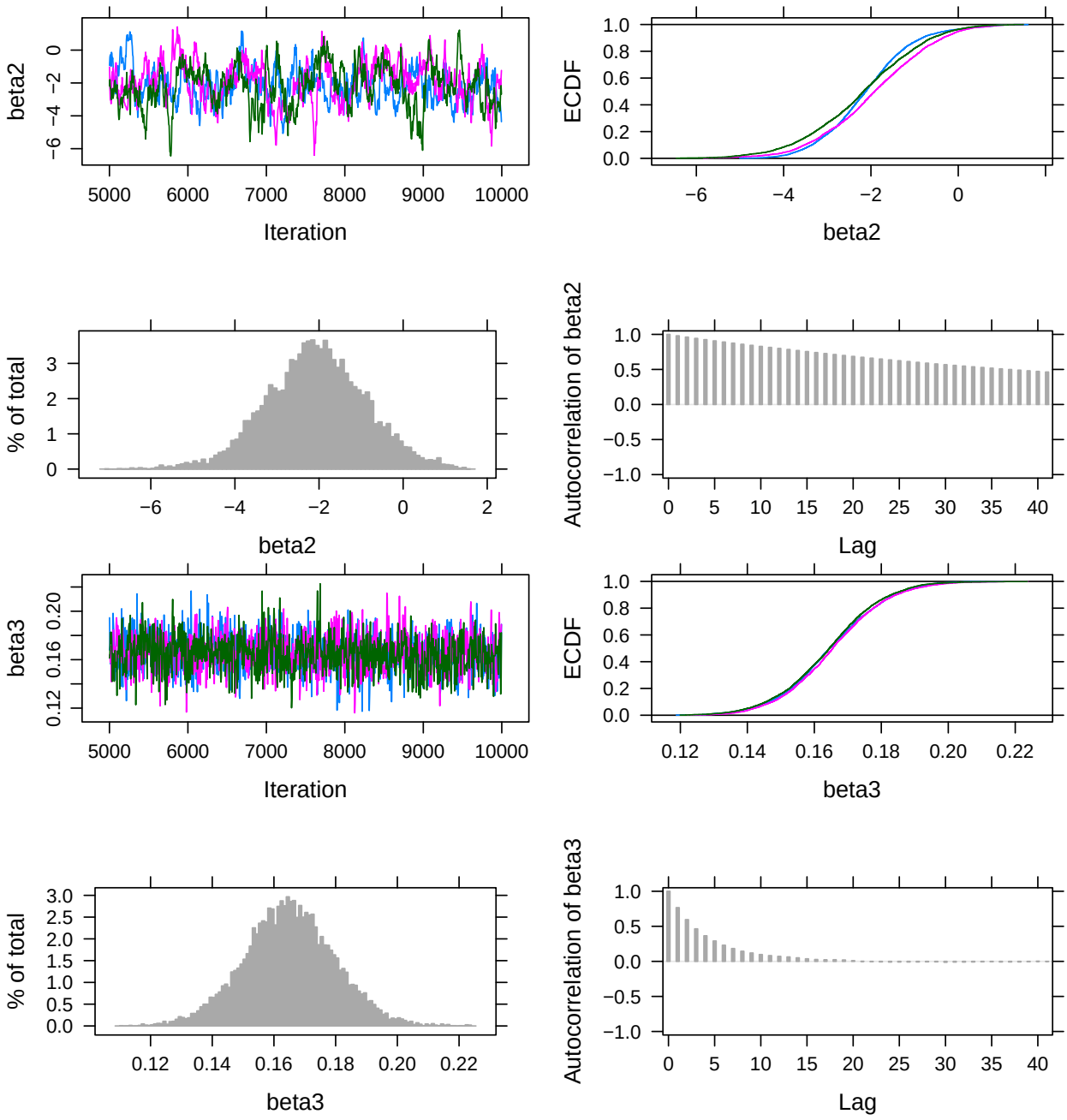
	Lower95	Median	Upper95	Mean	SD	Mode	MCerr	MC%ofSD	SSeff	AC.10	psrf
beta0[1]	-18.66	-7.81	3.01	-7.66	5.44	NA	0.44	8.2	150	0.81	1.02
beta0[2]	-19.41	-8.21	2.06	-8.10	5.43	NA	0.45	8.2	149	0.82	1.02
beta1	0.12	0.12	0.13	0.12	0.00	NA	0.00	3.0	1087	0.24	1.00
beta2	-4.37	-2.07	0.33	-2.07	1.19	NA	0.10	8.0	155	0.82	1.02
beta3	0.14	0.16	0.19	0.17	0.01	NA	0.00	2.3	1863	0.10	1.00
sd	4.39	5.45	6.75	5.51	0.61	NA	0.01	1.4	5035	0.02	1.00

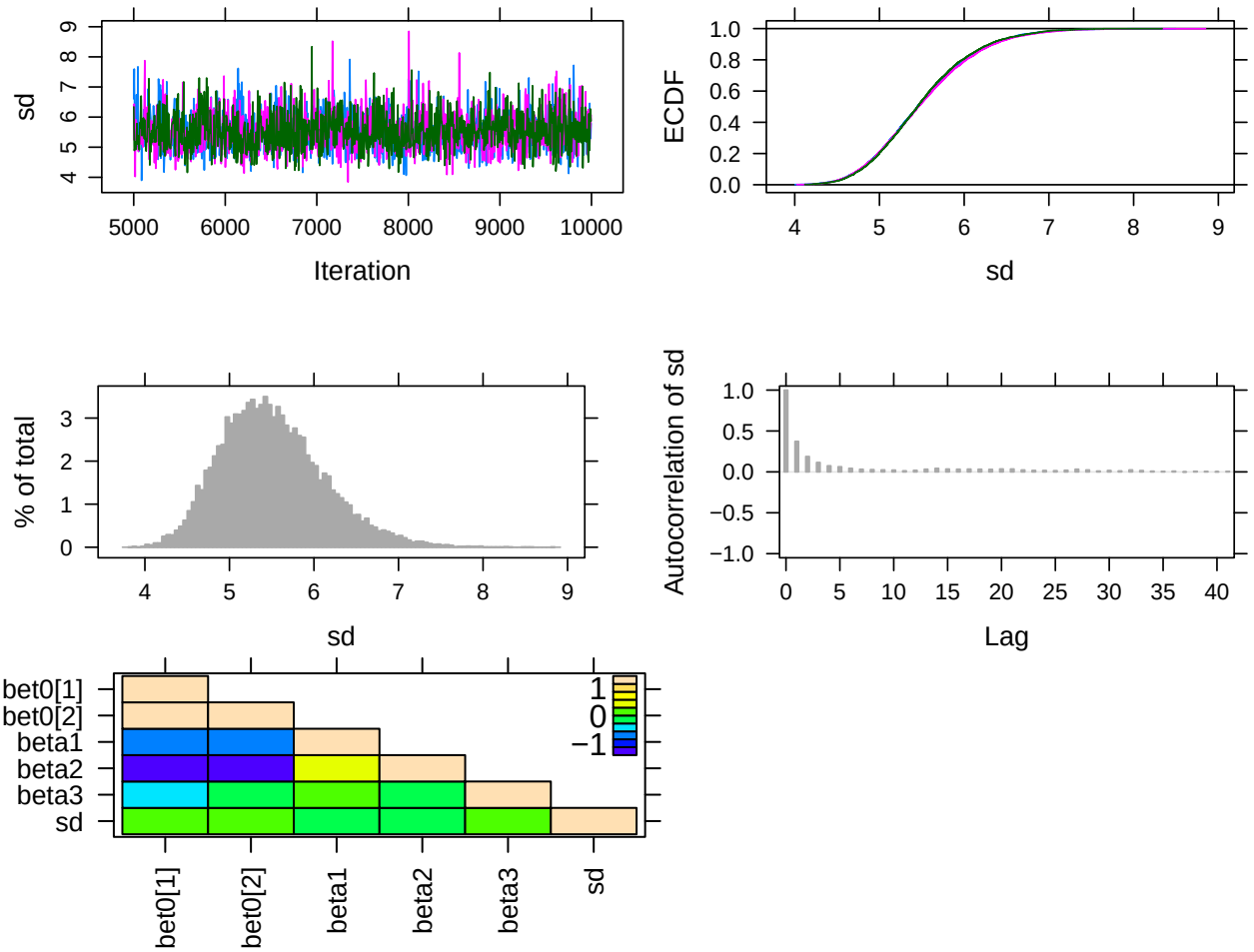
Looking good in terms of convergence, but let's do a visual inspection just for good measure.

```
plot(Frog.mod1)
## Generating plots...
```




beta0[2]
Iteration
ECDF
beta0[2]
% of total
beta0[2]
Autocorrelation of beta0[2]
Lag
beta1
Iteration
ECDF
beta1
% of total
beta1
Autocorrelation of beta1
Lag


beta2
Iteration
ECDF
beta2
% of total
beta2
Autocorrelation of beta2
Lag
beta3
Iteration
ECDF
beta3
% of total
beta3
Autocorrelation of beta3
Lag



Chains are mixing well and the autocorrelation drops off pretty quickly. Looks like our model has converged! Notice, however, that the CIs for beta0 are reallllly wide - not very useful if this were real data and a real model!

Model 1 in NIMBLE

Running the model in NIMBLE is almost the same as running it in JAGS, except we can skip the precision part.

```
library(nimble)
nimbleFrogs1 <- nimbleCode({
    # don't forget this part

    for (i in 1:n.frogs) {
        meanweight[i] <- beta0[s[i]] + age[i] * beta1 + leg[i] * beta2 +
            distance[i] * beta3
        weight[i] ~ dnorm(meanweight[i], sd = sd)
    }

    beta0[1] ~ dunif(-50, 50)
    beta0[2] ~ dunif(-50, 50)
    beta1 ~ dunif(-50, 50)
    beta2 ~ dunif(-50, 50)
```

```
    beta3 ~ dunif(-50, 50)

    sd ~ dunif(1e-04, 100)
})
```

As always, we need to define our params, data, constants and inits arguments. Don't forget the new variable "s" is now a new constant.

```
params <- c("beta0", "beta1", "beta2", "beta3", "sd")
data <- list(weight = Frogs$Weight, distance = Frogs$Dist, age = Frogs$Age,
    leg = Frogs$Leg)
constants <- list(n.frogs = 50, s = s)
inits <- list(beta0 = runif(2, -10, 10), beta1 = runif(1, -10, 10), beta2 = runif(1,
    -10, 10), beta3 = runif(1, -10, 10), sd = runif(1, 0.001, 100))
```

Time for our 7 step process to run the model. Notice that this time we'll prepare for using model selection by enabling WAIC. We'll also want to check convergence as normal. I'll explain what WAIC does in the next section.

```
prepfrogs <- nimbleModel(code = nimbleFrogs1, constants = constants, data = data,
    inits = inits)
## 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.
prepfrogs$initializeInfo()  #everything is good to go!
## All model variables are initialized.
mcmcfrogs <- configureMCMC(prepfrogs, monitors = params, print = T)
## ===== Monitors =====
## thin = 1: beta0, beta1, beta2, beta3, sd
## ===== Samplers =====
## RW sampler (6)
##   - beta0[]  (2 elements)
##   - beta1
##   - beta2
##   - beta3
##   - sd
frogsMCMC <- buildMCMC(mcmcfrogs, enableWAIC = TRUE)  # a change here to allow us to use WAIC
## Monitored nodes are valid for WAIC.
Cmodel <- compileNimble(prepfrogs)  #compiling the model itself in C++;
## compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ compilation details.
## compilation finished.
Compfrogs <- compileNimble(frogsMCMC, project = prepfrogs)  # compile the samplers next
## compiling... this may take a minute. Use 'showCompilerOutput = TRUE' to see C++ compilation details.
## compilation finished.
Frog.mod.nimble <- runMCMC(Compfrogs, niter = 30000, thin = 1, nchains = 3,
    nburnin = 10000, samplesAsCodaMCMC = TRUE, WAIC = TRUE)
## running chain 1...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## running chain 2...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
```

```
## running chain 3...
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
```

```
library(coda)
summary(Frog.mod.nimble$samples)
gelman.diag(Frog.mod.nimble$samples)
```

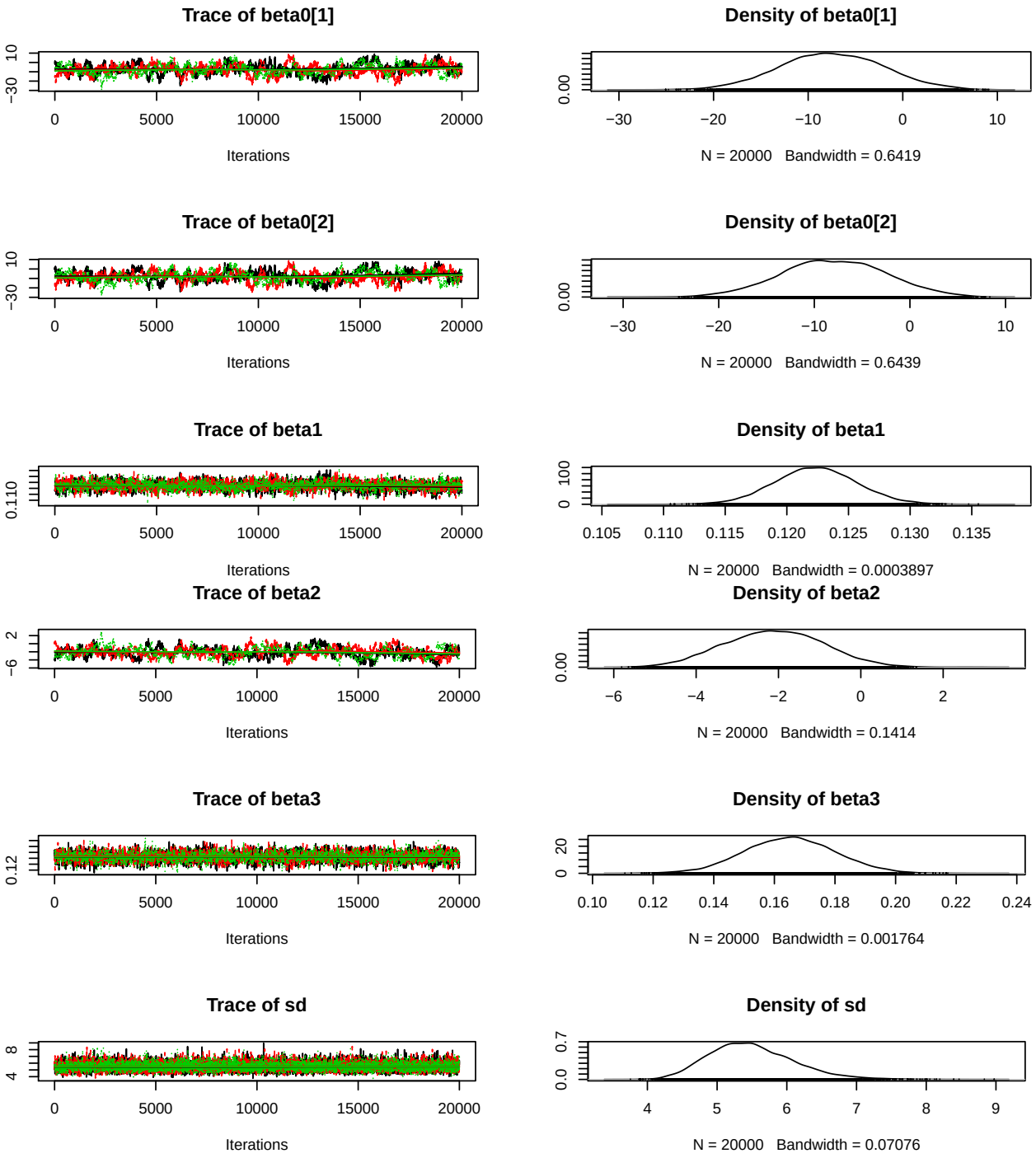
	Mean	SD	Naive SE	Time-series SE
beta0[1]	-7.51	5.47	0.02	0.38
beta0[2]	-7.97	5.48	0.02	0.40
beta1	0.12	0.00	0.00	0.00
beta2	-2.10	1.20	0.00	0.08
beta3	0.17	0.02	0.00	0.00
sd	5.50	0.61	0.00	0.01

	2.5%	25%	50%	75%	97.5%
beta0[1]	-18.22	-11.26	-7.54	-3.68	3.26
beta0[2]	-18.78	-11.73	-7.98	-4.16	2.73
beta1	0.12	0.12	0.12	0.12	0.13
beta2	-4.43	-2.94	-2.10	-1.27	0.24
beta3	0.14	0.15	0.17	0.18	0.19
sd	4.46	5.07	5.45	5.87	6.84

```
## Potential scale reduction factors:
##
##          Point est. Upper C.I.
## beta0[1]       1.02       1.06
## beta0[2]       1.02       1.06
## beta1          1.01       1.04
## beta2          1.01       1.05
## beta3          1.00       1.01
## sd             1.00       1.00
##
## Multivariate psrf
##
## 1.02
```

Happy convergence! But per always, we should plot our chains just in case.

```
plot(mcmc.list(Frog.mod.nimble$samples))
```



The results are essentially the same as we got from our JAGS output. Whooo!

Model Selection: DIC

So now we have our new model... but is it really any better than any other model we might try? This is where model selection comes in. There's no one way that's agreed upon for Bayesian model selection. DIC is convenient and already built into JAGS but there's not theoretical basis for why it works. WAIC is a more robust option than DIC, but requires slightly more calculation (unless you run NIMBLE, which has WAIC built in). Both options give fairly similar answers and are appropriate when the model is not a mixture

model, so we're good to use them for our simple linear regressions.

But what are they? Gelman, Hwang, and Vehtari 2013 give a great explanation of these methods but I will paraphrase in a hopefully helpful way. First, let's talk about DIC since many people (currently) seem to use this if they do model selection for linear regression.

DIC is a "somewhat Bayesian version of AIC". The goal of DIC is similar to that of AIC - estimate the likelihood of that model and correct for the number of parameters. The model with the lowest DIC score is considered the "best" model OF THE MODELS WE TESTED. This is not a test for the "true model" only a test for the "best of the options available". DIC is calculated via:

$$DIC = -2log(p(y|\theta_{bayes})) + 2p_{DIC}$$

If we break this apart, we can see the similarities with AIC. Firstly, $p(y|\theta_{bayes})$ simply refers to the posterior estimate of θ, AKA the fit of the model. How well does the model fit the data we gave it? Intuitively we have a sense that a better fit to the data makes a better model, so this part of the formula is logical. We then take the log of it and multiply it by -2 to make it similar to AIC. The second half of the equation tries to correct for potential overfitting of data. In general, the more parameters we add, the better the model will seem to "fit" the data, but that doesn't mean it's going to be any good at predicting anything. We want to balance the fit with the data and the ability to predict points not in our dataset. Similar to the k in AIC, $2p_{DIC}$ refers to the effective number of parameters in the model. The more parameters we have, the higher the value for this half of the equation and the higher the value of DIC.

To extract DIC from our JAGS run, we simply run:

```
Dic_1 <- extract(Frog.mod1, what = "dic")
## Loading required namespace: rjags
## Compiling rjags model and adapting for 1000 iterations...
## Obtaining DIC samples from 5000 iterations...
Dic_1
## Mean deviance:  310.8
## penalty 6.5
## Penalized deviance: 317.3
```

We find that the mean deviance (the value of interest) is 310.7907421. Cool! But useless unless we run more models and compare values.

I'm not going to show you how to extract DIC from NIMBLE, since it automatically calculates WAIC, and WAIC is a generally more robust method of model selection. If you want my biased, unexpert opinion, WAIC is the better method overall.

Model Selection: WAIC

WAIC is (shocker) also fairly similar but attempts to approximate cross-validation to help determine model fit. WAIC uses a similar measure of goodness-of-fit as DIC, (log of the average posterior likelihood for each data point), but uses the posterior variance in log-likelihood, with larger variances resulting in harsher penalties.

If we want to calculate WAIC from NIMBLE, we simply have to enable it during the run commands and then extract it via:

```
Frog.mod.nimble$WAIC
## [1] 326.973
```

If we want to calculate WAIC in JAGS, we have to do a little bit more work. We're going to calculate the deviance of our points using, in this case, the log-normal PDF equation.

Wow that sounds scary! But it's really simple. In our model, we currently pull weight from a normal, with mean of "meanweight" and precision "prec". Now we're going to ask JAGS to give us the probability that our

weight value (our data) really came from that same normal distribution we just pulled from. If we have a really good model, then the probability that we pulled from that distribution should be high, because, you know, *it came from* that distribution. But if our model is really terrible, then the deviance will be really high and the probability really low - in other words, the difference between our data and the "expected" value from the model will be very different.

Here's what that all looks like in JAGS. Note that if we were modeling weight from something other than a normal, we'd have to use the logdensity.(whatever the distribution was) function instead of logdensity.norm.

```
modelstring.Frogs1 = "
  model
{
for (i in 1:n.frogs){
  meanweight[i] <- beta0[s[i]] + age[i]*beta1 + leg[i]*beta2 + distance[i]*beta3
    weight[i] ~ dnorm(meanweight[i], prec)
    loglik[i] <- logdensity.norm(weight[i], meanweight[i], prec) #for WAIC calculation
}

beta0[1] ~ dunif(-50,50)
beta0[2] ~ dunif(-50,50)
beta1 ~ dunif(-50,50)
beta2 ~ dunif(-50,50)
beta3 ~ dunif(-50,50)

prec <- 1/(sd *sd)
sd ~ dunif(0.0001, 100)
}
"
```

```
params <- c("beta0", "beta1", "beta2", "beta3", "sd", "loglik")
data <- list(weight = Frogs$Weight, distance = Frogs$Dist, age = Frogs$Age,
    leg = Frogs$Leg, n.frogs = 50, s = s)
Frog.mod1 <- run.jags(model = modelstring.Frogs1, monitor = params, data = data,
    n.chains = 3, sample = 5000, method = "parallel")
## Warning: No initial values were provided - JAGS will use the same initial values
## for all chains
## Warning: You attempted to start parallel chains without setting different PRNG
## for each chain, which is not recommended. Different .RNG.name values have been
## added to each set of initial values.
## Calling 3 simulations using the parallel method...
## Following the progress of chain 1 (the program will wait for all chains
## to finish before continuing):
## Welcome to JAGS 4.3.0 on Tue Apr  6 10:46:55 2021
## JAGS is free software and comes with ABSOLUTELY NO WARRANTY
## Loading module: basemod: ok
## Loading module: bugs: ok
## . . Reading data file data.txt
## . Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 50
##    Unobserved stochastic nodes: 6
##    Total graph size: 513
## . Reading parameter file inits1.txt
```

```
## . Initializing model
## . Adapting 1000
## -------------------------------------------------| 1000
## +++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## Adaptation successful
## . Updating 4000
## -------------------------------------------------| 4000
## ************************************************* 100%
## . . . . . . . Updating 5000
## -------------------------------------------------| 5000
## ************************************************* 100%
## . . . . Updating 0
## . Deleting model
## .
## 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
```

We can then calculate WAIC by hand using the code below. First we find all the output that relates to "loglik" (in this case, there are 50). Then we put those together as a matrix, find the mean and variance log-likelihood for each element (each data point) and then stick that into the WAIC equation.

```
vars <- grep("loglik", colnames(Frog.mod1$mcmc[[1]]))
# find the output that relates to loglik
like <- as.matrix(Frog.mod1$mcmc[, vars, ])
fbar <- colMeans(exp(like))
# mean likelihood (note - not on log scale)
Pw <- sum(apply(like, 2, var))
# mean variance in log-likelihood
WAIC <- -2 * sum(log(fbar)) + 2 * Pw
WAIC
## [1] 327.4315
```

We aren't going to get the same value from our hand calculation as from NIMBLE (326.9729746) because of a slight difference in how NIMBLE calculates WAIC, but the difference is not very important. The main thing is that we compare WAIC values that have been calculated the same way - either by hand or via NIMBLE.

Of course, the output of DIC or WAIC is not particularly meaningful on its own. It is only meaningful in comparison to other DIC or WAIC values for models run with the same data. So let's run some other models!

Models 2 and 3 in JAGS

Returning to our models, let's test a model where one frog species grows faster than the other. Let's say we expect a difference in the relationship with age, but no difference in the intercept.

```
modelstring.Frogs2 = "
  model
{
for (i in 1:n.frogs){
  meanweight[i] <- beta0 + age[i]*beta1[s[i]] + leg[i]*beta2 + distance[i]
  #note that the only thing that changes is the indexing of beta0 and beta1
```

```
  weight[i] ~ dnorm(meanweight[i], prec)
  loglik[i] <- logdensity.norm(weight[i], meanweight[i], prec)
}

#but now beta0 is only one value and  beta1 can be two different values
beta0 ~ dunif(-50,50)
beta1[1] ~ dunif(-50,50)
beta1[2] ~ dunif(-50,50)
beta2 ~ dunif(-50,50)
beta3 ~ dunif(-50,50)



#I like to convert precision to standard deviation because it confuses me otherwise.
prec <- 1/(sd *sd)
sd ~ dunif(0.0001, 100)
}
"
```

Now let's run the model through JAGS and see what we get.

```
Frog.mod2 <- run.jags(model = modelstring.Frogs2, monitor = params, data = data,
    n.chains = 3, sample = 5000, method = "parallel")
## Warning: No initial values were provided - JAGS will use the same initial values
## for all chains
## Warning: You attempted to start parallel chains without setting different PRNG
## for each chain, which is not recommended. Different .RNG.name values have been
## added to each set of initial values.
## Calling 3 simulations using the parallel method...
## Following the progress of chain 1 (the program will wait for all chains
## to finish before continuing):
## Welcome to JAGS 4.3.0 on Tue Apr  6 10:47:03 2021
## JAGS is free software and comes with ABSOLUTELY NO WARRANTY
## Loading module: basemod: ok
## Loading module: bugs: ok
## . . Reading data file data.txt
## . Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 50
##    Unobserved stochastic nodes: 6
##    Total graph size: 463
## . Reading parameter file inits1.txt
## . Initializing model
## . Adapting 1000
## -------------------------------------------------| 1000
## +++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## Adaptation successful
## . Updating 4000
## -------------------------------------------------| 4000
## ************************************************* 100%
## . . . . . . . Updating 5000
## -------------------------------------------------| 5000
```

```
## ************************************************** 100%
## . . . . Updating 0
## . Deleting model
## .
## 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
```

```
head(summary(Frog.mod2), n = 5)
```

```
## Calculating summary statistics...
## Calculating the Gelman-Rubin statistic for 56 variables....
```

	Lower95	Median	Upper95	Mean	SD	Mode	MCerr	MC%ofSD	SSeff	AC.10	psrf
beta0	-50.00	-23.39	25.01	-18.91	22.78	NA	1.12	4.9	412	0.60	1
beta1[1]	0.09	0.14	0.20	0.14	0.03	NA	0.00	2.0	2596	0.04	1
beta1[2]	0.10	0.16	0.22	0.16	0.03	NA	0.00	2.0	2545	0.05	1
beta2	-30.04	-17.27	-7.83	-17.92	5.81	NA	0.27	4.7	450	0.56	1
beta3	-48.73	0.36	46.00	0.12	28.72	NA	0.23	0.8	15000	0.01	1

Looks like our model converged. We would also want to graph it like always to double check (not shown).

Let's extract our WAIC and DIC values and see how it compares to model 1. In real life, we'd only use one of these metrics, not both

```
Dic_2 <- extract(Frog.mod2, what = "dic")
## Compiling rjags model and adapting for 1000 iterations...
## Obtaining DIC samples from 5000 iterations...
Dic_2
## Mean deviance:  527.7
## penalty 4.392
## Penalized deviance: 532.1
vars2 <- grep("loglik", colnames(Frog.mod2$mcmc[[1]]))  #find the output that relates to loglik
like2 <- as.matrix(Frog.mod2$mcmc[, vars2, ])
fbar2 <- colMeans(exp(like2))  #mean likelihood
Pw2 <- sum(apply(like2, 2, var))  #mean variance in log-likelihood
WAIC2 <- -2 * sum(log(fbar2)) + 2 * Pw2
WAIC2
## [1] 531.8519
```

To compare with our previous model:

```
Dic_1
## Mean deviance:  310.8
## penalty 6.5
## Penalized deviance: 317.3
Dic_2  #bigger aka worse
## Mean deviance:  527.7
## penalty 4.392
## Penalized deviance: 532.1
```

```
WAIC
## [1] 327.4315
WAIC2  #bigger, aka not better
## [1] 531.8519
```

Seems that this model is slightly worse than model 1. Let's test one more model for funsies. Maybe Joe thinks about his frogs and again and thinks that maybe weight is only related to age, but he still suspects there's a different equation (both intercept and slope) for both species.

```
modelstring.Frogs3 = "
  model
{
for (i in 1:n.frogs){
  meanweight[i] <- beta0[s[i]] + age[i]*beta1[s[i]]
  #got to reindex everything
  weight[i] ~ dnorm(meanweight[i], prec)
  loglik[i] <- logdensity.norm(weight[i], meanweight[i], prec)
}

#adjust priors to match
beta0[1] ~ dunif(-50,50)
beta0[2] ~ dunif(-50,50)
beta1[1] ~ dunif(-50,50)
beta1[2] ~ dunif(-50,50)

prec <- 1/(sd *sd)
sd ~ dunif(0.0001, 100)
}
"
```

Per usual, we adjust the parameters to monitor, data and initial values, then send it all to JAGS.

```
params3 <- c("beta0", "beta1", "sd", "loglik")
data3 <- list(weight = Frogs$Weight, age = Frogs$Age, n.frogs = 50, s = s)
inits3 <- function() {
    list(beta0 = runif(2, -10, 10), beta1 = runif(2, -10, 10))
}
Frog.mod3 <- run.jags(model = modelstring.Frogs3, monitor = params3, inits = inits3,
    data = data3, n.chains = 3, sample = 10000, method = "parallel")
## Warning: You attempted to start parallel chains without setting different PRNG
## for each chain, which is not recommended. Different .RNG.name values have been
## added to each set of initial values.
## Calling 3 simulations using the parallel method...
## Following the progress of chain 1 (the program will wait for all chains
## to finish before continuing):
## Welcome to JAGS 4.3.0 on Tue Apr  6 10:47:12 2021
## JAGS is free software and comes with ABSOLUTELY NO WARRANTY
## Loading module: basemod: ok
## Loading module: bugs: ok
## . . Reading data file data.txt
## . Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 50
```

```
##    Unobserved stochastic nodes: 5
##    Total graph size: 311
## . Reading parameter file inits1.txt
## . Initializing model
## . Adapting 1000
## -------------------------------------------------| 1000
## +++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## Adaptation successful
## . Updating 4000
## -------------------------------------------------| 4000
## ************************************************* 100%
## . . . . . Updating 10000
## -------------------------------------------------| 10000
## ************************************************* 100%
## . . . . Updating 0
## . Deleting model
## .
## All chains have finished
## Simulation complete.  Reading coda files...
## Coda files loaded successfully
## 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
```

```
head(summary(Frog.mod3), n = 4)
```

```
## Calculating summary statistics...
## Calculating the Gelman-Rubin statistic for 55 variables....
```

	Lower95	Median	Upper95	Mean	SD	Mode	MCerr	MC%ofSD	SSeff	AC.10	psrf
beta0[1]	-4.26	3.34	10.92	3.38	3.86	NA	0.07	1.7	3440	0.11	1
beta0[2]	-13.80	-5.46	3.32	-5.45	4.35	NA	0.09	2.0	2611	0.18	1
beta1[1]	0.09	0.11	0.13	0.11	0.01	NA	0.00	1.7	3374	0.11	1
beta1[2]	0.10	0.12	0.14	0.12	0.01	NA	0.00	2.0	2575	0.18	1

Next we can grab our DIC and WAIC values:

```
Dic_3 <- extract(Frog.mod3, what = "dic")
## Compiling rjags model and adapting for 1000 iterations...
## Obtaining DIC samples from 10000 iterations...

vars3 <- grep("loglik", colnames(Frog.mod3$mcmc[[1]]))
# find the output that relates to loglik
like3 <- as.matrix(Frog.mod3$mcmc[, vars3, ])
fbar3 <- colMeans(exp(like3))  #mean likelihood
Pw3 <- sum(apply(like3, 2, var))
# mean variance in log-likelihood
WAIC3 <- -2 * sum(log(fbar3)) + 2 * Pw3
```

Let's make a table to compare all the results:

```
data.frame(Mods = 1:3, DICS = c(sum(Dic_1$deviance), sum(Dic_2$deviance),
    sum(Dic_3$deviance)), WAICS = c(WAIC, WAIC2, WAIC3))
```

Mods	DICS	WAICS
1	310.79	327.43
2	527.68	531.85
3	377.02	382.03

Okay, that model was a little worse than model 1! But remember that DIC and WAIC only tell you about models you've tested - in fact, the true model I used to create this data didn't use any of these models. So take all your model outputs with a grain of salt.

Models 2 and 3 in NIMBLE

Running the output from NIMBLE would be redundant, but here's those same models as above but in NIMBLE. Notice that basically nothing changes - the main differences are just in the way the models are sent to their various programs and the use of sd instead of precision.

```
nimbleFrogs2 <- nimbleCode({
    # don't forget this part
    for (i in 1:n.frogs) {
        meanweight[i] <- beta0 + age[i] * beta1 + leg[i] * beta2 + distance[i] *
            beta3[s[i]]
        # note that the only thing that changes is the indexing of beta0 and
        # beta3
        weight[i] ~ dnorm(meanweight[i], sd = sd)
    }

    # but now beta0 is only one value and beta3 can be two different values
    beta0 ~ dunif(-50, 50)
    beta1 ~ dunif(-50, 50)
    beta2 ~ dunif(-50, 50)
    beta3[1] ~ dunif(-50, 50)
    beta3[2] ~ dunif(-50, 50)

    sd ~ dunif(1e-04, 100)
})
```

```
nimble.Frogs3 <- nimbleCode({
    for (i in 1:n.frogs) {
        meanweight[i] <- beta0[s[i]] + age[i] * beta1[s[i]]
        # got to reindex everything
        weight[i] ~ dnorm(meanweight[i], sd = sd)
    }

    # adjust priors to match
    beta0[1] ~ dunif(-50, 50)
    beta0[2] ~ dunif(-50, 50)
    beta1[1] ~ dunif(-50, 50)
    beta1[2] ~ dunif(-50, 50)

    sd ~ dunif(1e-04, 100)
})
```

Graphing the Output - Option 1

I, for one, am fairly lazy. Especially when it comes to math. If a program can do the work for me, I'm happy to let it do it! So for graphing output where multiple variables have credible intervals, I like to have JAGS (or NIMBLE) do the work for me.

We can add a few lines to our code to get some nice point estimates at equally spaced points along the line we want to graph. Let's make a graph showing the relationship between expected weight and distance to road.

First, we add some new variables into our model that represent the predicted weights for species A and species B at different distances along the road. We'll hold the other variables constant at their mean values.

```
modelstring.Frogs2graph = "
model
{
for (i in 1:n.frogs){
  meanweight[i] <- beta0[s[i]] + age[i]*beta1 + leg[i]*beta2 + distance[i]*beta3
    weight[i] ~ dnorm(meanweight[i], prec)
}

beta0[1] ~ dunif(-50,50)
beta0[2] ~ dunif(-50,50)
beta1 ~ dunif(-50,50)
beta2 ~ dunif(-50,50)
beta3 ~ dunif(-50,50)

prec <- 1/(sd *sd)
sd ~ dunif(0.0001, 100)

#Let's graph distances from 0 to 200 m from the road
#(reasonable, given our data ranges from 1.7 to 198)

for (k in 1:201 ){ #can't iterate starting at 0,
#so we'll just use k-1 in our equation and start at 1 instead

  graph_w1[k] <- beta0[1] + 392*beta1 + 4*beta2 + (k-1)*beta3 #species 1
  graph_w2[k] <- beta0[2] + 392*beta1 + 4*beta2 + (k-1)*beta3 #species 2
}
}
"
```

Now we'll want to monitor these new variables graphw1 and graphw2.

```
params2 <- c("beta0", "beta1", "beta2", "beta3", "sd", "graph_w1", "graph_w2")
Frog.mod2graph <- run.jags(model = modelstring.Frogs2graph, monitor = params2,
    data = data, n.chains = 3, sample = 5000, method = "parallel")
## Warning: No initial values were provided - JAGS will use the same initial values
## for all chains
## Warning: You attempted to start parallel chains without setting different PRNG
## for each chain, which is not recommended. Different .RNG.name values have been
## added to each set of initial values.
## Calling 3 simulations using the parallel method...
## Following the progress of chain 1 (the program will wait for all chains
## to finish before continuing):
## Welcome to JAGS 4.3.0 on Tue Apr  6 10:47:26 2021
## JAGS is free software and comes with ABSOLUTELY NO WARRANTY
```

```
## Loading module: basemod: ok
## Loading module: bugs: ok
## . . Reading data file data.txt
## . Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 50
##    Unobserved stochastic nodes: 6
##    Total graph size: 1468
## . Reading parameter file inits1.txt
## . Initializing model
## . Adapting 1000
## -------------------------------------------------| 1000
## +++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## Adaptation successful
## . Updating 4000
## -------------------------------------------------| 4000
## ************************************************* 100%
## . . . . . . . Updating 5000
## -------------------------------------------------| 5000
## ************************************************* 100%
## . . . Updating 0
## . Deleting model
## .
## 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
```

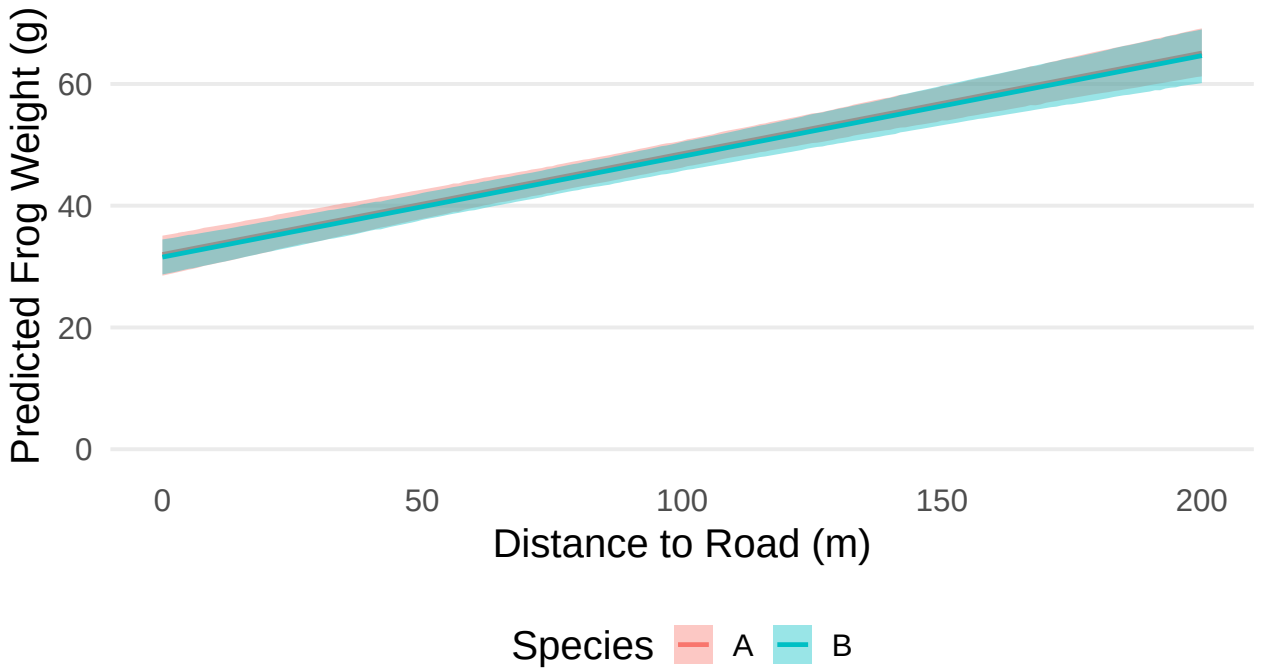
This will produce 402 more nodes, so I'm not going to look at my output this time. We already know the model should converge at this point since we haven't added any new stochastic information to the model, so we don't have to worry about convergence as long as we run the same number of chains as before. Instead, we extract the means and CI's from the output without looking at them and throw them into a nice dataframe for ggplot to deal with. (If you want the base R graph, it's available in the R code, but I'll skip it for now)

```
res <- summary(Frog.mod2graph)
## Calculating summary statistics...
## Calculating the Gelman-Rubin statistic for 408 variables....
nodes <- c(grep("graph_w1", rownames(res)), grep("graph_w2", rownames(res)))
graphme <- data.frame(low = res[nodes, 1], upper = res[nodes, 3], mean = res[nodes,
    4], Species = rep(c("A", "B"), each = 201), dist = c(0:200, 0:200))
```

And then we throw it into ggplot! Obviously you can customize your graph however, but here's the basic idea.

```
library(ggplot2)
## Warning: package 'ggplot2' was built under R version 3.6.2
ggplot(data = graphme, aes(x = dist, group = Species, col = Species, fill = Species)) +
    geom_smooth(aes(y = mean, ymin = low, ymax = upper), stat = "Identity") +
    xlab("Distance to Road (m)") + ylab("Predicted Frog Weight (g)") +
    ylim(0, 70) + theme_minimal(base_size = 18) + theme(legend.position = "bottom",
    panel.grid.minor = element_blank(), panel.grid.major.x = element_blank(),
```

```
    strip.background = element_blank())
```



Tragically not the most interesting of graphs. But yay! Now you can graph output, perform model selection and write linear regression with categorical variables (hopefully). The same exact procedure can be done in NIMBLE - just extract the values from the mcmc chains and send them to ggplot!

Graphing the Output - Option 2

If you don't feel like re-running your model, here's an alternative option to getting that same beautiful output.

Let's graph model 3, even though it wasn't the best model, just for some variety. As a reminder, here's the relationship we modeled:

```
meanweight[i] <- beta0[s[i]] + age[i] * beta1[s[i]]
```

First, we're going to grab the values of beta0 and beta1 from each iteration of our model run. We'll grab the last 1001 iterations so that we know our results are reliable and the chains will have mixed well. We're just grabbing from the first chain here, but you can grab from whatever combo of chains you want.

```
coef.post <- as.matrix(Frog.mod3$mcmc[[1]][9000:10000, c("beta0[1]", "beta1[1]",
    "beta0[2]", "beta1[2]")])
```

Next we grab a few model iterations (in this case 100, but you can take however many you want), and make a matrix of the predicted weight values for each species. Note that this is using the betas from each iteration instead of them mean betas from all the iterations. We'll also need to predict over a range of ages we're interested in. Our data spans frogs ages 24 to 773 days, so we'll also graph that range as well.

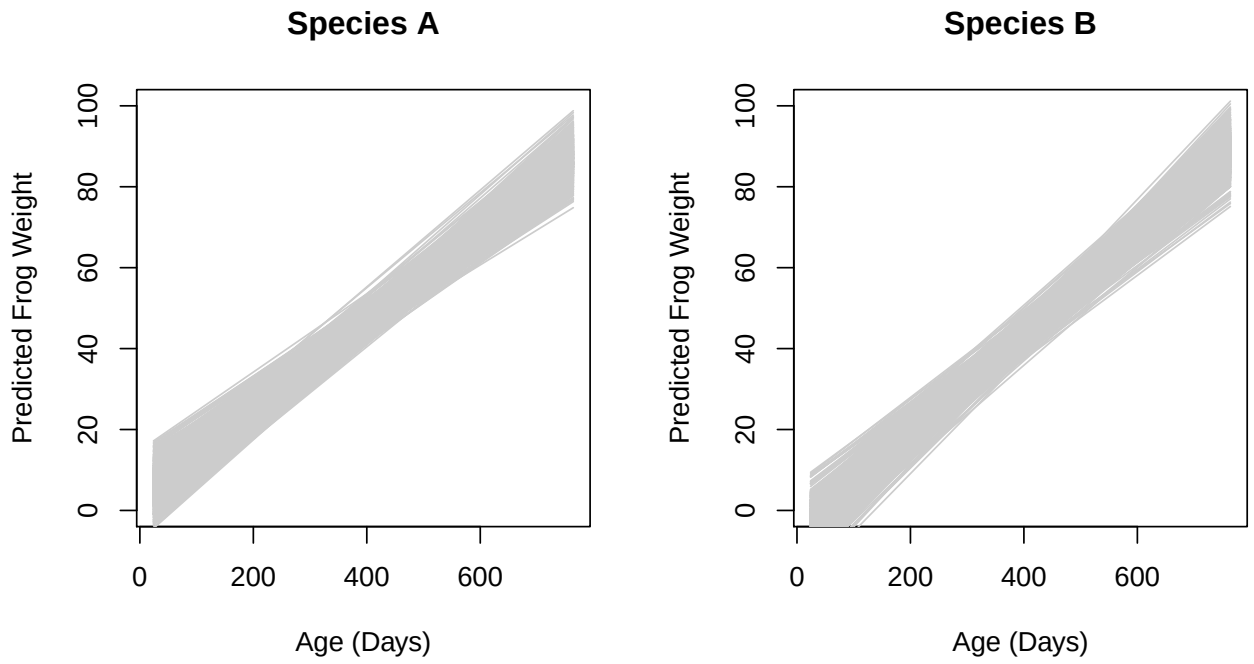
```
n.iter <- nrow(coef.post)
ages <- seq(24, 773, by = 10)
coef.pred1 <- matrix(NA, nrow = n.iter, ncol = length(ages))
coef.pred2 <- matrix(NA, nrow = n.iter, ncol = length(ages))
for (i in 1:n.iter) {
    coef.pred1[i, ] <- coef.post[i, "beta0[1]"] + ages * coef.post[i, "beta1[1]"]
    coef.pred2[i, ] <- coef.post[i, "beta0[2]"] + ages * coef.post[i, "beta1[2]"]
```

```
}
```

Each column is now the expected weight for a frog of that age, with coef.pred1 representing species 1 and coef.pred2 representing species 2.

We can now graph our results in ggplot or base R, whichever we prefer. Here's a way to do it in base R.

```
par(mfrow = c(1, 2))
plot(ages, coef.pred1[1, ], type = "l", xlab = "Age (Days)", ylab = "Predicted Frog Weight",
    ylim = c(0, 100), col = "white", main = "Species A")
for (i in 1:n.iter) {
    lines(ages, coef.pred1[i, ], col = grey(0.8))
}
plot(ages, coef.pred2[1, ], type = "l", xlab = "Age (Days)", ylab = "Predicted Frog Weight",
    ylim = c(0, 100), col = "white", main = "Species B")
for (i in 1:n.iter) {
    lines(ages, coef.pred2[i, ], col = grey(0.8))
}
```



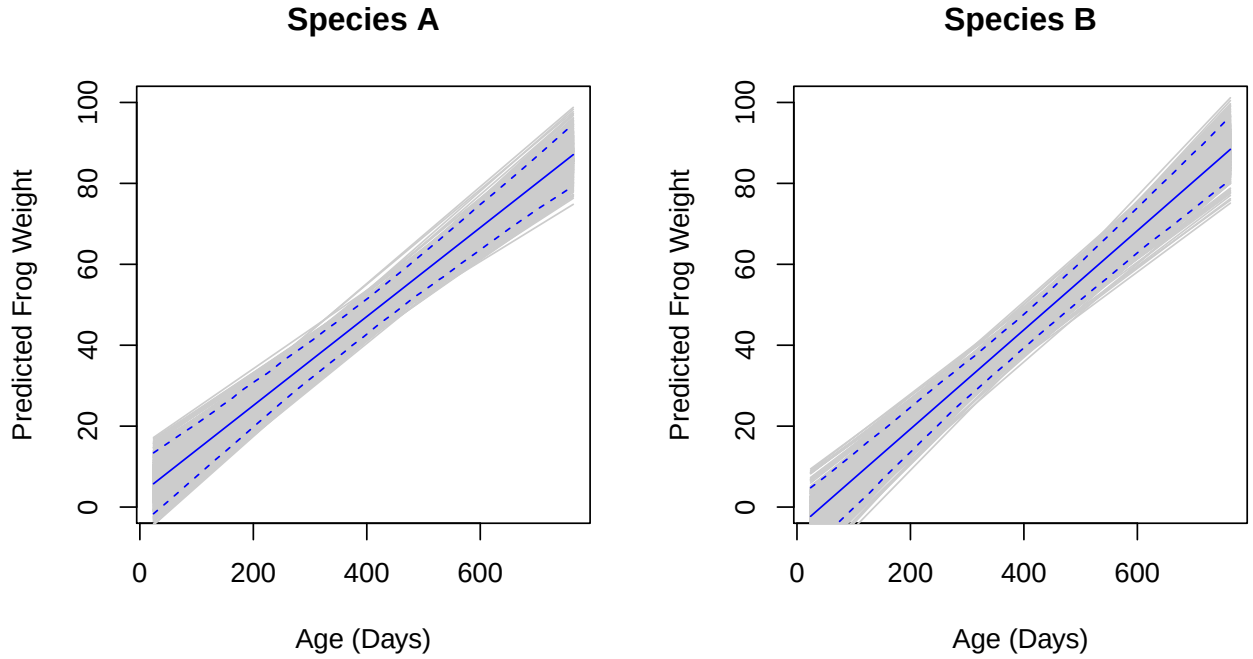
To get the CI lines, we can calculate them from our prediction objects, then plot:

```
post.mean.1 <- colMeans(coef.pred1)
post.lower1 <- apply(coef.pred1, 2, quantile, prob = 0.025)
post.upper1 <- apply(coef.pred1, 2, quantile, prob = 0.975)
post.mean.2 <- colMeans(coef.pred2)
post.lower2 <- apply(coef.pred2, 2, quantile, prob = 0.025)
post.upper2 <- apply(coef.pred2, 2, quantile, prob = 0.975)
```

```
par(mfrow = c(1, 2))  #splits the plot into two plots
plot(ages, coef.pred1[1, ], type = "l", xlab = "Age (Days)", ylab = "Predicted Frog Weight",
    ylim = c(0, 100), col = "white", main = "Species A")
for (i in 1:n.iter) {
    lines(ages, coef.pred1[i, ], col = grey(0.8))
}
lines(ages, post.mean.1, col = "blue", type = "l")
```

```r
lines(ages, post.lower1, col = "blue", type = "l", lty = 2)
lines(ages, post.upper1, col = "blue", type = "l", lty = 2)

plot(ages, coef.pred2[1, ], type = "l", xlab = "Age (Days)", ylab = "Predicted Frog Weight",
    ylim = c(0, 100), col = "white", main = "Species B")
for (i in 1:n.iter) {
    lines(ages, coef.pred2[i, ], col = grey(0.8))
}
lines(ages, post.mean.2, col = "blue", type = "l")
lines(ages, post.lower2, col = "blue", type = "l", lty = 2)
lines(ages, post.upper2, col = "blue", type = "l", lty = 2)
```



Whoo! Fancy graphs!