Choosing Initial Values

Heather Gaya

2022-11-28

Time to talk about Initial Values! A lot of frustrating hours can be spent trying to even get your MCMC model to start running. For some reason, initial values are often glossed over during lectures about building Bayesian models and it seems about time to explain (1) possible process to finding good initials.

For now, this tutorial will focus on initial values for JAGS, but the process doesn't change for NIMBLE. Maybe in a future update, I'll add NIMBLE into this tutorial.

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

@doofgradstudent@ecoevo.social

Contents

Let's say I'm given the following model, along with the corresponding data files. Perhaps I am reviewing a manuscript, or maybe I'm a TA and someone asked me for help. Or maybe I'm just running my own code from awhile ago that I annotated particularly badly.

Model Example 1

Here's our beautiful model file:

```
writeLines("model {

 beta1 ~ dnorm(0, 0.5)
 beta2 ~ dnorm(0, 0.5)
 beta3 ~ dnorm(0, 0.5)
 beta4 ~ dnorm(0, 0.5)


 alpha1 ~ dnorm(0, 0.5)
 alpha2 ~ dnorm(0, 0.5)

 for (i in 1:nSites){
   eps1[i] ~ dnorm(0, tau.alpha)    # Random site effects
 }

 for (i in 1:nSites){
   eps2[i] ~ dnorm(0, tau.alpha)    # Random site effects on detection
   }
tau.alpha <- 1/ (sd.alpha * sd.alpha)
sd.alpha ~ dunif(0, 5)

for(i in 1:nSites) {
   log(lambda[i]) <- eps1[i] + beta1*Elvation[i] + beta2*observedAirTemp[i] + beta4*RH[i]
   N[i] ~ dpois(lambda[i])         # Latent local abundance
   log(sigma[i]) <- alpha1*cloud[i] + alpha2*wind[i] + eps2[i] ## detection model
   tau[i] <- 1/sigma[i]^2
   for(j in 1:nBins) {
     ## Trick to do integration for *point-transects*
     pbar[i,j] <- (sigma[i]^2 * (1-exp(-b[j+1]^2/(2*sigma[i]^2))) -
                   sigma[i]^2 * (1-exp(-b[j]^2/(2*sigma[i]^2)))) *
                  2*3.141593/area[j]
     pi[i,j] <- psi[j]*pbar[i,j]
   }
      n[i] ~ dbin(sum(pi[i,]), N[i])
   y[i,] ~ dmulti(pi[i,1:nBins]/sum(pi[i,]), n[i])
}
totalAbundance <- sum(N[1:nSites])

}",
    "PC_RE.txt")
```

And here's the data that someone provided us to run the model:

```
jags.pars.pc <- c("beta1", "beta2", "beta3", "beta4", "eps2", "alpha1",
    "alpha2", "totalAbundance")
jagsdat <- dget("jags.data_pc.txt")
str(jagsdat)
```

```
## List of 12
##  $ y              : int [1:9, 1:3] 1 2 4 0 1 1 0 1 0 2 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:3] "X.0.20_sum" "X.20.40_sum" "X.40.60_sum"
##  $ n              : num [1:9] 4 12 13 8 8 8 12 5 5
##  $ area           : num [1:3] 3491 10472 17453
##  $ b              : num [1:4] 0 33.3 66.7 100
##  $ psi            : num [1:3] 0.111 0.333 0.556
##  $ Elvation       : num [1:9] -0.188 1.52 0.599 0.972 0.653 ...
##  $ observedAirTemp: num [1:9] -0.0836 -0.9573 1.6881 -0.5204 1.2512 ...
##  $ RH             : num [1:9] 0.0564 -0.6902 0.1161 0.7731 0.3849 ...
##  $ cloud          : num [1:9] -1.8798 1.4562 0.0265 -0.9267 0.0265 ...
##  $ wind           : num [1:9] 0.836 0.393 0.836 -0.934 -0.491 ...
##  $ nSites         : int 9
##  $ nBins          : num 3
```

We dutifully can send this information to JAGS and see what it tells us as the problem. We don't even need to "understand" the model to provide reasonable initial values. First we can try running it without initial values, because sometimes we can get away with that.

First Error

```
library(jagsUI)
##
## Attaching package: 'jagsUI'
## The following object is masked from 'package:coda':
##
##     traceplot
jags.post.pt <- jags.basic(data=jagsdat, #no initial values
                           parameters.to.save=jags.pars.pc,
                           model.file="PC_RE.txt",
                           n.chains=3, n.adapt=100, n.burnin=0,
                           n.iter=5000, parallel=TRUE)
##
## Processing function input.......
##
## Done.
##
## Beginning parallel processing using 3 cores. Console output will be suppressed.
## Error in checkForRemoteErrors(val): 3 nodes produced errors; first error: Error in node n[1]
## Node inconsistent with parents
```

The error we receive is "Error in node n[1]. Node inconsistent with parents"

What does this mean? Well, let's see where n is "created" in the model:

```
n[i] ~ dbin(sum(pi[i, ]), N[i])
```

This leaves us with two courses of action - figure out what sum(pi[1,]) is and see if we can understand the issue or figure out what N[1] is. If n[1] doesn't make sense with it's "parent" value, then the problem is going to be either:

1. N[1] < n[1], which means that the results of the binomial with any probability will never give us n
2. sum(pi[i,]) is so low that the probability of that we saw n/N individuals is unreasonable

3. sum(pi[i,]) is so HIGH that the probability that we only saw n/N individuals is unreasonable.

The easiest place to start fixing this is by giving better initial values for N. We can start by giving values of N that are much larger than our values of n.

```
jags.inits <- function() {
    list(N = jagsdat$n + 10)
}
jags.inits()
## $N
## [1] 14 22 23 18 18 18 22 15 15
```

Second Error

```
jags.post.pt <- jags.basic(data = jagsdat, inits = jags.inits, parameters.to.save = jags.pars.pc,
    model.file = "PC_RE.txt", n.chains = 3, n.adapt = 100, n.burnin = 0,
    n.iter = 5000, parallel = TRUE)
##
## Processing function input.......
##
## Done.
##
## Beginning parallel processing using 3 cores. Console output will be suppressed.
## Error in checkForRemoteErrors(val): 3 nodes produced errors; first error: Error in node y[1,1:3]
## Node inconsistent with parents
```

Now that we've given JAGS some initial values, it has a new problem. "Error in node y[1,1:3]. Node inconsistent with parents"

Okay, back to the model to see where y[1,1:3] appears.

```
y[i, ] ~ dmulti(pi[i, 1:nBins]/sum(pi[i, ]), n[i])
```

So y[1,1:3] comes from a multinomial draw, where the probability of each bin is: pi[1,1:nBins]/sum(pi[1,]) and the total size of the multinomial is n[1]. If we look at our data, we can see that n[1] is data. We can't change data! And y[1,1:3] is also data!

That means we need to figure out where this pi stuff comes from. Back to the model:

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

pi comes from the big mess above. If we look at our jagsdata object, we can see we provide psi, b and area as data. So that leaves us with only one place to look: sigma!

Where does sigma get defined in the model?

```
log(sigma[i]) <- alpha1 * cloud[i] + alpha2 * wind[i] + eps2[i]
```

Great. So somehow we need to choose reasonable initial values for alpha1, alpha2 and eps2 so that sigma is a reasonable value so that the pi's become a reasonable value! Oof!

Here's a tip: Try setting anything that deal with a covariate to 0 as an initial value. If we do that, we only have to deal with finding a reasonable value for the intercept (in this case eps2).

Where does eps2 get defined?

```
for (i in 1:nSites) {
    eps2[i] ~ dnorm(0, tau.alpha)  # Random site effects on detection
}
tau.alpha <- 1/(sd.alpha * sd.alpha)
sd.alpha ~ dunif(0, 5)
```

Awesome. So basically as long as we choose a reasonable value for eps2 that isn't crazy far from 0, we should be good. Let's experiment with different values of eps2.

Lazy Method

First off, I'll show you the lazy way.

```
jags.inits <- function() {
    list(N = jagsdat$n + 10, alpha1 = 0, alpha2 = 0, eps2 = rep(1, jagsdat$nSites))
}
jags.inits()
## $N
## [1] 14 22 23 18 18 18 22 15 15
##
## $alpha1
## [1] 0
##
## $alpha2
## [1] 0
##
## $eps2
## [1] 1 1 1 1 1 1 1 1 1
```

```
jags.post.pt <- jags.basic(data = jagsdat, inits = jags.inits, parameters.to.save = jags.pars.pc,
    model.file = "PC_RE.txt", n.chains = 3, n.adapt = 100, n.burnin = 0,
    n.iter = 5000, parallel = TRUE)
##
## Processing function input.......
##
## Done.
##
## Beginning parallel processing using 3 cores. Console output will be suppressed.
## Error in checkForRemoteErrors(val): 3 nodes produced errors; first error: Error in node y[1,1:3]
## Node inconsistent with parents
```

Nope, let's try a bigger value of eps2 then:

```
jags.inits <- function() {
    list(N = jagsdat$n + 10, alpha1 = 0, alpha2 = 0, eps2 = rep(2, jagsdat$nSites))
}
jags.inits()
## $N
## [1] 14 22 23 18 18 18 22 15 15
##
## $alpha1
## [1] 0
##
## $alpha2
## [1] 0
##
## $eps2
## [1] 2 2 2 2 2 2 2 2 2
```

```
jags.post.pt <- jags.basic(data = jagsdat, inits = jags.inits, parameters.to.save = jags.pars.pc,
    model.file = "PC_RE.txt", n.chains = 3, n.adapt = 100, n.burnin = 0,
    n.iter = 5000, parallel = TRUE)
##
## Processing function input.......
##
## Done.
##
## Beginning parallel processing using 3 cores. Console output will be suppressed.
## Error in checkForRemoteErrors(val): 3 nodes produced errors; first error: Error in node y[1,1:3]
## Node inconsistent with parents
```

Still bigger?

```
jags.inits <- function() {
    list(N = jagsdat$n + 10, alpha1 = 0, alpha2 = 0, eps2 = rep(8, jagsdat$nSites))
}
jags.inits()
## $N
## [1] 14 22 23 18 18 18 22 15 15
##
## $alpha1
## [1] 0
##
## $alpha2
## [1] 0
##
## $eps2
## [1] 8 8 8 8 8 8 8 8 8
```

```
jags.post.pt <- jags.basic(data = jagsdat, inits = jags.inits, parameters.to.save = jags.pars.pc,
    model.file = "PC_RE.txt", n.chains = 3, n.adapt = 100, n.burnin = 0,
    n.iter = 5000, parallel = TRUE)
##
## Processing function input.......
##
## Done.
##
## Beginning parallel processing using 3 cores. Console output will be suppressed.
##
## Parallel processing completed.
##
## MCMC took 0.017 minutes.
```

Woot! Model is running!

Less Lazy Way

Obviously sometimes you'll try a bunch of guesses and the guesses just won't be good and won't fix the problem. A better way is to do write up a big loop and test out some guesses without having to run the whole model. More coding in the short term, but a better solution for more complicated issues.

Here's the flow of how our initial value for eps2 connects to y[1,1:3]:

```
eps2 <- Myinitialvalue
log(sigma[i]) <- alpha1 * cloud[i] + alpha2 * wind[i] + eps2[i]
for (j in 1:nBins) {
    ## Trick to do integration for *point-transects*
    pbar[i, j] <- (sigma[i]^2 * (1 - exp(-b[j + 1]^2/(2 * sigma[i]^2))) -
        sigma[i]^2 * (1 - exp(-b[j]^2/(2 * sigma[i]^2)))) * 2 * 3.141593/area[j]
    pi[i, j] <- psi[j] * pbar[i, j]
}
y[i, ] ~ dmulti(pi[i, 1:nBins]/sum(pi[i, ]), n[i])
```

We can write ourselves a little test function. Notice that instead of running the dmulti, we're just going to ask R to give us a few example datasets that might come from the pi probabilities we're trying to start our MCMC at. If the output looks kind of similar to our own data, we can call that a decent initial value.

```
findeps2 <- function(x) {
    eps2 <- x
    sigma <- 0 * jagsdat$cloud[1] + 0 * jagsdat$wind[1] + eps2
    pbar <- pi <- array(NA, dim = jagsdat$nBins)  #empty array
    for (j in 1:jagsdat$nBins) {
        pbar[j] <- (sigma^2 * (1 - exp(-jagsdat$b[j + 1]^2/(2 * sigma^2))) -
            sigma^2 * (1 - exp(-jagsdat$b[j]^2/(2 * sigma^2)))) * 2 * 3.141593/jagsdat$area[j]
        pi[j] <- jagsdat$psi[j] * pbar[j]
    }

    return(pi/sum(pi))
}
```

We're trying to find a value for eps2 that produces *reasonable* values for our multinomial cell probabilities, but how do we know what's reasonable? Essentially all we need is for there to be SOME probability, even if it's very small, that we could see observations in our farther distance bins.

Consider if eps2 = 1:

```
findeps2(1)
## [1] 1 0 0
```

This says there's essentially 0 probability of an observation in any other distance bin besides the first. That's not really what we saw in our data and it's not a good place to start our MCMC. What about a bigger number?

```
findeps2(7)
## [1] 9.999881e-01 1.191314e-05 0.000000e+00
```

The probability that we see anything outside the first bin is still small, but not 0. Let's give that a go in JAGS.

```
jags.inits <- function() {
    list(N = jagsdat$n + 10, alpha1 = 0, alpha2 = 0, eps2 = rep(7, jagsdat$nSites))
}
jags.inits()
## $N
## [1] 14 22 23 18 18 18 22 15 15
##
## $alpha1
## [1] 0
##
```

```
## $alpha2
## [1] 0
##
## $eps2
## [1] 7 7 7 7 7 7 7 7 7
```

```
jags.post.pt <- jags.basic(data = jagsdat, inits = jags.inits, parameters.to.save = jags.pars.pc,
    model.file = "PC_RE.txt", n.chains = 3, n.adapt = 100, n.burnin = 0,
    n.iter = 5000, parallel = TRUE)
##
## Processing function input.......
##
## Done.
##
## Beginning parallel processing using 3 cores. Console output will be suppressed.
##
## Parallel processing completed.
##
## MCMC took 0.017 minutes.
```

Lovely!

Out of curiousity, let's see what the model actually estimate eps2 to be:

```
summary(jags.post.pt[, "eps2[1]", ])
##
## Iterations = 1:5000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean             SD       Naive SE Time-series SE
##         5.3811         2.4858         0.0203         0.1781
##
## 2. Quantiles for each variable:
##
##    2.5%     25%     50%     75%   97.5%
##  0.3902  3.7512  5.3692  6.9831 10.3823
```

Model 2

Let's try the same concept with another model. This one is a simple occupancy model, stolen from another one of my tutorials (check out my website if you want to learn more!)

Here's the model:

```
writeLines("
model {
for (i in 1:n.sites) {
  logit(psi[i]) <- psi.b0 + psi.b1*elevation[i]
  occ[i] ~ dbern(psi[i])
  for (t in 1:6) {
    logit(p[i,t]) <- p.b0 + p.b1*noise[i,t]
    obs[i,t] ~ dbern(p[i,t] * occ[i])
  }
}
psi.b1 ~ dnorm(0, 0.37)
psi.b0 ~ dnorm(0, 0.37)
p.b0 ~ dnorm(0, 0.37)
p.b1 ~ dnorm(0, 0.37)

totalocc <- sum(occ[])

}",
    "Occ.txt")
```

And here's the data/parameters we've been given:

```
params <- c("psi.b0", "psi.b1", "p.b0", "p.b1", "totalocc")
jd <- dget("jags.data_occ.txt")
str(jd)
## List of 6
##  $ n.sites  : int 40
##  $ elevation: num [1:40] 0.754 0.712 0.347 1.352 0.315 ...
##  $ noise    : num [1:40, 1:6] 15.02 0.759 18.246 13.725 11.021 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:6] "noise1" "noise2" "noise3" "noise4" ...
##  $ obs      : int [1:40, 1:6] 0 1 0 0 0 1 0 0 0 0 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:6] "Obs1" "Obs2" "Obs3" "Obs4" ...
##  $ fake.elev: num [1:53] 0.2 0.225 0.25 0.275 0.3 0.325 0.35 0.375 0.4 0.425 ...
##  $ n.graph  : int 53
```

Let's give it a go

A Lovely Error

```
jags.post.occ <- jags.basic(data = jd, parameters.to.save = params, model.file = "Occ.txt",
    n.chains = 3, n.adapt = 100, n.burnin = 0, n.iter = 5000, parallel = TRUE)
##
## Processing function input.......
##
## Done.
##
## Beginning parallel processing using 3 cores. Console output will be suppressed.
## Error in checkForRemoteErrors(val): 3 nodes produced errors; first error: Error in node obs[2,1]
## Node inconsistent with parents
```

Excellent, our first error. “Error in node obs[2,1]. Node inconsistent with parents”.

Let's see what obs[2,1] connects with in our model:

```
obs[i, t] ~ dbern(p[i, t] * occ[i])
```

And what is obs[2,1]?

```
jd$obs[2, 1]
## Obs1
##    1
```

Why would it be angry about a 1 coming out of a bernouli draw? Either:

1. p[2,1] is very low and a 1 is very unlikely
2. occ[2] is 0.

Where is occ[2] drawn from?

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

Awesome, it looks like occupancy is just drawn from another bernouli, so we can set our intial values on occ itself. To save ourselves some trouble, we'll just set all sites as occ = 1.

```
inits.occ <- function() {
    list(occ = rep(1, jd$n.sites))
}
inits.occ()
## $occ
##  [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 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [39] 1 1
```

```
jags.post.occ <- jags.basic(data = jd, inits = inits.occ, parameters.to.save = params,
    model.file = "Occ.txt", n.chains = 3, n.adapt = 100, n.burnin = 0,
    n.iter = 5000, parallel = TRUE)
##
## Processing function input.......
##
## Done.
##
## Beginning parallel processing using 3 cores. Console output will be suppressed.
##
## Parallel processing completed.
##
## MCMC took 0.026 minutes.
```

Whoooo, our model runs!