NIMBLE Abundance

Colin Lewis-Beck

June 20, 2017

For Abundance Models the package unmarked takes 3 data objects: a matrix with the observed counts where rows are sites (R) and columns are visits (T); a matrix of site level covariates (rows are sites, columns are site level variables); and a list of site level covariates where each matrix within the list corresponds to one covariate, and is an R x T matrix.

As an example, below is the mallard data set from the unmarked package. Using TidyR, we can put the data into long format where each row is an observation, and each column is a variable.

```
##   site visit count   elev length forest   ivel   date
## 1    1   y.1     0 -1.173  0.801 -1.156 -0.506 -1.761
## 2    2   y.1     0 -1.127  0.115 -0.501 -0.934 -2.904
## 3    3   y.1     3 -0.198 -0.479 -0.101 -1.136 -1.690
## 4    4   y.1     0 -0.105  0.315  0.008 -0.819 -2.190
## 5    5   y.1     3 -1.034 -1.102 -1.193  0.638 -1.833
## 6    6   y.1     0 -0.848  0.741  0.917 -1.329 -2.619
```

However, I am not sure whether this data structure would help for an abundance.nim function: especially since the data needs to be passed to nimbleModel as a list. It seems like the structure of abundance.nim should be as follows. Let's assume we have R sites and T measurements, C site covariates, and O observation level covariates. The structure of the function is similar to the package unmarked.

```
abundance.nim <- function(observation formula, site formula, Observed Counts (RxT matrix),
Observation Covarites (list of length O of RxT matrices for each covariate), Site Covariates (R x C mat
```

As a toy example, the function would function as follows:

```
mod1 <- abundance.nim(~ 1 + date, ~ 1 + evel + A, y = y, ObsCov = list(date = date), Site Covariates = 
```

The function would generate the following code:

latent = N[1:R] ~ nim_glm(1 + evel[1:R] + A[1:nlevels], factors = "A", family = pois, link = log, priors = priors)
obs = y[1:R, 1:T] ~ nim_glm(1 + date[1:R, 1:T], factors = "None", family = binom, link = logit, priors = priors)

Then we use nim_glm expansion Module

```
latent.expand <- nim_glm$process(latent)
obs.expand <- nim_glm$process(obs)
```

This generates:

for (i in 1:R){
N[i] ~ dpoi(lambda[i])
}

lambda[i] <- lmPred(1 + evel[1:R] + A[1:nlevels], priors = priors, factors = "A", link = log)

```
for (i in 1:R){
for (t in 1:T){
y[i,j] ~ dbinom(p(i,j), N[i])
}
}
```

```
p(i,j) <- lmPred(1 + date[1:R, 1:T], priors = priors, factors = “None”, link = logit)
```

```
lmPred$process(latent.expand$LHS, latent.expand$RHS)
lmPred$process(obs.expand$LHS, obs.expand$RHS)
```

This expands the linear predictors

```
for (i in 1:R){
log(lambda[i]) <- l.intercept + evel*evel[i] + A.effect[A[i]]
}
```

```
for (i in 1:R){
for (t in 1:T){
logit(p(i,j)) <- p.intercept + date * date[i,j]
} }
```

```
l.intercept ~ dnorm(0, 100)
p.intercept ~ dnorm(0, 100)
ym ~ dnorm(0, 100)
xm ~ dnorm(0, 100)
for (i in 1:nlevels){
A.effect[i] ~ dnorm(0, 100)
}
```

All these code pieces would need to be grouped together into one final code object.

I think passing the data in seperate pieces rather than constructing a tidy data object will make the data easier to pass to nimbleModel. Once the initial values are generated, something like the following will make the nimbleModel.

```
nimMod.obj <- nimbleModel(code = full.expand, inits = values, constants = list(date = date, A = A, evel = evel), data = y)
```

Then a function can be called to runMCMC and return the final results and BUGS code.

Note: random slope and intercepts can be included with (1|A) or (xm|A). One key piece will be to be able to generated the nested loops and be able to extract the two formula objects from the initial function.