**Appendix B.** R code for fitting GLMM and occupancy models in NIMBLE language.

###################################################################

#### Functions for defining occupancy distribution and running MCMC

##################################################

### NIMBLE custom dOccupancy distribution ########

##################################################

dOccupancy <- nimbleFunction(

run = function(x = double(), p\_occ = double(), p\_obs = double(), log.p = double()) {

if(x == 0) L <- 1 - p\_occ \* p\_obs else L <- p\_occ \* p\_obs

returnType(double())

if(log.p) return(log(L)) else return(L)

}

)

rOccupancy <- nimbleFunction(

run = function(n = integer(), p\_occ = double(), p\_obs = double()) {

print('not implemented')

returnType(double())

return(0)

}

)

registerDistributions(list(

dOccupancy = list(

BUGSdist = 'dOccupancy(p\_occ, p\_obs)',

discrete = TRUE

)

))

#################################################

#### Function for running MCMC multiple times

#################################################

mcmcClusterFunction <- function(x){

set.seed(x)

Cmcmc$run(niter)

samples <- as.matrix(Cmcmc$mvSamples)[(burnin+1):niter,]

return(samples)

}

##################################################

### RW\_shift and RW\_shift\_log samplers ##########

##################################################

## Sampler for a target node that offsets a set of other nodes by an opposite amount as the proposed random walk step for the target node

## Control element skipDepenendencies can be TRUE if you are SURE that the resulting likelihood will be completely unmodified

## Otherwise skipDependencies should be FALSE.

sampler\_RW\_shift <- nimbleFunction(

contains = sampler\_BASE,

setup = function(model, mvSaved, target, control) {

### control list extraction ###

adaptive <- control$adaptive

adaptInterval <- control$adaptInterval

scale <- control$scale

shiftNodes <- model$expandNodeNames(control$shiftNodes)

skipDependencies <- control[['skipDependencies']]

if(is.null(skipDependencies)) skipDependencies <- FALSE

### node list generation ###

targetAsScalar <- model$expandNodeNames(target, returnScalarComponents = TRUE)

if(length(targetAsScalar) > 1) stop('more than one target; cannot use RW sampler, try RW\_block sampler')

if(!skipDependencies) {

calcNodes <- model$getDependencies(c(target, shiftNodes))

determCalcNodes <- character()

}

else {

calcNodes <- c(target, shiftNodes)

determCalcNodes <- model$getDependencies(c(target, shiftNodes), determOnly = TRUE, self = FALSE)

}

### numeric value generation ###

scaleOriginal <- scale

timesRan <- 0

timesAccepted <- 0

timesAdapted <- 0

scaleHistory <- c(0, 0)

acceptanceRateHistory <- c(0, 0)

## variables previously inside of nested functions:

optimalAR <- 0.44

gamma1 <- 0

},

run = function() {

propShift <- rnorm(1, mean = 0, sd = scale)

model[[target]] <<- model[[target]] + propShift

values(model, shiftNodes) <<- values(model, shiftNodes) - propShift

logMHR <- calculateDiff(model, calcNodes)

jump <- decide(logMHR)

if(jump) {

nimCopy(from = model, to = mvSaved, row = 1, nodes = calcNodes, logProb = TRUE)

if(skipDependencies) { ## This ensures that deterministic dependencies are updated.

calculate(model, determCalcNodes)

nimCopy(from = model, to = mvSaved, row = 1, nodes = determCalcNodes, logProb = TRUE)

}

}

else

nimCopy(from = mvSaved, to = model, row = 1, nodes = calcNodes, logProb = TRUE)

if(adaptive) adaptiveProcedure(jump)

},

methods = list(

adaptiveProcedure = function(jump = logical()) {

timesRan <<- timesRan + 1

if(jump) timesAccepted <<- timesAccepted + 1

if(timesRan %% adaptInterval == 0) {

acceptanceRate <- timesAccepted / timesRan

timesAdapted <<- timesAdapted + 1

setSize(scaleHistory, timesAdapted)

setSize(acceptanceRateHistory, timesAdapted)

scaleHistory[timesAdapted] <<- scale

acceptanceRateHistory[timesAdapted] <<- acceptanceRate

gamma1 <<- 1/((timesAdapted + 3)^0.8)

gamma2 <- 10 \* gamma1

adaptFactor <- exp(gamma2 \* (acceptanceRate - optimalAR))

scale <<- scale \* adaptFactor

timesRan <<- 0

timesAccepted <<- 0

}

},

reset = function() {

scale <<- scaleOriginal

timesRan <<- 0

timesAccepted <<- 0

timesAdapted <<- 0

scaleHistory <<- scaleHistory \* 0

acceptanceRateHistory <<- acceptanceRateHistory \* 0

gamma1 <<- 0

}

), where = getLoadingNamespace()

)

## Sampler for a standard deviation (typically) on a log scale that shifts the magnitude of a set of other nodes by the same value, also on a log scale.

## This helps mixing for a set of random effects and their shared standard deviation

sampler\_RW\_log\_shift <- nimbleFunction(

contains = sampler\_BASE,

setup = function(model, mvSaved, target, control) {

### control list extraction ###

adaptive <- control$adaptive

adaptInterval <- control$adaptInterval

scale <- control$scale

shiftNodes <- model$expandNodeNames(control$shiftNodes)

### node list generation ###

targetAsScalar <- model$expandNodeNames(target, returnScalarComponents = TRUE)

if(length(targetAsScalar) > 1) stop('more than one target; cannot use RW sampler, try RW\_block sampler')

numTargets <- 1 + length(shiftNodes)

calcNodes <- model$getDependencies(c(target, shiftNodes))

### numeric value generation ###

scaleOriginal <- scale

timesRan <- 0

timesAccepted <- 0

timesAdapted <- 0

scaleHistory <- c(0, 0)

acceptanceRateHistory <- c(0, 0)

## variables previously inside of nested functions:

optimalAR <- 0.44

gamma1 <- 0

},

run = function() {

propLogShift <- rnorm(1, mean = 0, sd = scale)

propMult <- exp(propLogShift)

model[[target]] <<- model[[target]] \* propMult

newVals <- values(model, shiftNodes) \* propMult

values(model, shiftNodes) <<- newVals

logMHR <- calculateDiff(model, calcNodes) + numTargets \* propLogShift

jump <- decide(logMHR)

if(jump)

nimCopy(from = model, to = mvSaved, row = 1, nodes = calcNodes, logProb = TRUE)

else

nimCopy(from = mvSaved, to = model, row = 1, nodes = calcNodes, logProb = TRUE)

if(adaptive) adaptiveProcedure(jump)

},

methods = list(

adaptiveProcedure = function(jump = logical()) {

timesRan <<- timesRan + 1

if(jump) timesAccepted <<- timesAccepted + 1

if(timesRan %% adaptInterval == 0) {

acceptanceRate <- timesAccepted / timesRan

timesAdapted <<- timesAdapted + 1

setSize(scaleHistory, timesAdapted)

setSize(acceptanceRateHistory, timesAdapted)

scaleHistory[timesAdapted] <<- scale

acceptanceRateHistory[timesAdapted] <<- acceptanceRate

gamma1 <<- 1/((timesAdapted + 3)^0.8)

gamma2 <- 10 \* gamma1

adaptFactor <- exp(gamma2 \* (acceptanceRate - optimalAR))

scale <<- scale \* adaptFactor

timesRan <<- 0

timesAccepted <<- 0

}

},

reset = function() {

scale <<- scaleOriginal

timesRan <<- 0

timesAccepted <<- 0

timesAdapted <<- 0

scaleHistory <<- scaleHistory \* 0

acceptanceRateHistory <<- acceptanceRateHistory \* 0

gamma1 <<- 0

}

), where = getLoadingNamespace()

)

#### Load packages

my.packages <- c("nimble", "coda", "lattice", "akima")

lapply(my.packages, require, character.only = TRUE)

#### Load data set for model fitting

inputData <- readRDS('output/data\_nimble\_zib.rds')

source('R\_functions/nimble\_definitions.R')

#####################################################

#### Define GLMM model in BUGS/NIMBLE language

glmmCode <- nimbleCode({

mu\_alpha ~ dnorm(0, 0.001)

sigma\_alpha ~ dunif(0, 1000)

for(j in 1:nsite) {

alpha[j] ~ dnorm(mu\_alpha, sd = sigma\_alpha) ## site random effect

}

for(i in 1:8) {

beta[i] ~ dnorm(0, 0.001)

}

# for(i in 1:4) {

# betaseason[i] ~ dnorm(0, 0.001) ## new fixed effects for each season

# }

for(i in 1:N) {

logit(p\_occ[i]) <- alpha[siteID[i]] + beta[1]\*list\_length[i] + beta[2]\*year\_list\_length[i] +

beta[3]\*aet[i] + beta[4]\*tmn[i] + beta[5]\*tmx[i] + beta[6]\*year[i] + beta[7]\*month[i] + beta[8]\*month2[i]

y[i] ~ dbin(size = 1, prob = p\_occ[i])

}

})

constants <- with(inputData,

list(N=N, nsite=nsite,

aet=aet, tmn=tmn, tmx=tmx,

year=year,

month=month,

month2=month2,

list\_length=list\_length,

year\_list\_length=year\_list\_length,

siteID=siteID))

data <- with(inputData, list(y=y))

inits <- list(mu\_alpha=0, sigma\_alpha=1, alpha=rep(0,inputData$nsite), beta=rep(0,8))

modelInfo\_glmm <- list(code=glmmCode, constants=constants, data=data, inits=inits, name='glmm\_month\_model')

#### Set up model and samplers

Rmodel <- nimbleModel(modelInfo\_glmm$code,

modelInfo\_glmm$constants,

modelInfo\_glmm$data,

modelInfo\_glmm$inits)

Cmodel <- compileNimble(Rmodel)

spec <- configureMCMC(Rmodel)

#### Best configuration of samplers for random effect occupancy model

spec$removeSamplers('beta[1:8]')

spec$addSampler('beta[1:8]', 'RW\_block') # linear coefficients

spec$removeSamplers('sigma\_alpha')

spec$addSampler('sigma\_alpha', 'RW\_log\_shift', list(shiftNodes='alpha')) # random effect sampler

spec$getSamplers() # Check samplers

spec$addMonitors('p\_occ') # add a monitor to get p\_occ in output

#### Compile MCMC in R and C++

Rmcmc <- buildMCMC(spec)

Cmcmc <- compileNimble(Rmcmc, project = Rmodel)

#### Run MCMC with 150,000 iterations and 50,000 burn-in

niter <- 150000

burnin <- 50000

samplesList <- lapply(1:3, mcmcClusterFunction)

########################################################################

#### Occupancy model

########################################################################

#####################################################

#### Define model in BUGS/NIMBLE language

code <- nimbleCode({

mu\_alpha ~ dnorm(0, 0.001)

sigma\_alpha ~ dunif(0, 1000)

for(j in 1:nsite) {

alpha[j] ~ dnorm(mu\_alpha, sd = sigma\_alpha) ## site random effect

}

for(i in 1:9) {

beta[i] ~ dnorm(0, 0.001)

}

# for(i in 1:4) {

# betaseason[i] ~ dnorm(0, 0.001) ## new fixed effects for each season

# #betaseasonyear[i] ~ dnorm(0, 0.001)

# }

for(i in 1:N) {

logit(p\_occ[i]) <- alpha[siteID[i]] + beta[4]\*aet[i] + beta[5]\*tmn[i] + beta[6]\*tmx[i] + beta[7]\*year[i] + beta[8]\*month[i] + beta[9]\*month2[i]

logit(p\_obs[i]) <- beta[1] + beta[2]\*list\_length[i] + beta[3]\*year\_list\_length[i]

y[i] ~ dOccupancy(p\_occ[i], p\_obs[i])

}

})

constants <- with(inputData,

list(N=N, nsite=nsite,

aet=aet, tmn=tmn, tmx=tmx,

year=year,

month=month,

month2=month2,

list\_length=list\_length,

year\_list\_length=year\_list\_length,

siteID=siteID))

data <- with(inputData, list(y=y))

inits <- list(mu\_alpha=0, sigma\_alpha=1, alpha=rep(0,inputData$nsite), beta=rep(0,9))#, betaseason=rep(0,4))

modelInfo\_month <- list(code=code, constants=constants, data=data, inits=inits, name='month\_model')

#### Set up model and samplers

Rmodel <- nimbleModel(modelInfo\_month$code,

modelInfo\_month$constants,

modelInfo\_month$data,

modelInfo\_month$inits)

Cmodel <- compileNimble(Rmodel)

spec <- configureMCMC(Rmodel)

#### Best configuration of samplers for random effect occupancy model

spec$removeSamplers('beta[1:9]')

spec$addSampler('beta[1:3]', 'RW\_block') # detection sub-model sampler

spec$addSampler('beta[4:9]', 'RW\_block') # occupancy sub-model sampler

spec$removeSamplers('sigma\_alpha')

spec$addSampler('sigma\_alpha', 'RW\_log\_shift', list(shiftNodes='alpha')) # random effect sampler

spec$getSamplers() # Check samplers

spec$addMonitors('p\_occ') # add a monitor to get p\_occ in output

#### Compile MCMC in R and C++

Rmcmc <- buildMCMC(spec)

Cmcmc <- compileNimble(Rmcmc, project = Rmodel)

#### Run MCMC with 150,000 iterations and 50,000 burn-in

niter <- 150000

burnin <- 50000

samplesList <- lapply(1:3, mcmcClusterFunction)

save(samplesList, file = 'output/MCMC\_month\_list2.RData')