CJS in NIMBLE

NIMBLE 2022 virtual EFI workshop

NIMBLE Development Team

April 2022

Bonus example: CJS capture-recapture model with "classic" dipper data.

Dipper example:

- 294 Dippers monitored 1981-1987.
- One of the most classic capture-recapture teaching datasets ever.
- Thanks to Daniel Turek and Olivier Gimenez for Dipper examples from previous workshops.

Load the data

```
dipper_example_dir <- file.path("..", "..", "content", "exa
dipper <- read.csv(file.path(dipper_example_dir,"dipper.csv
y <- as.matrix(dipper[ , 1:7])
y <- y + 1 # Code as 1 = not captured, 2 = captured.
first <- apply(y, 1, function(x) min(which(x != 1))) # firs
y <- y[ first != 7, ] # remove records with first capture
head(y)
```

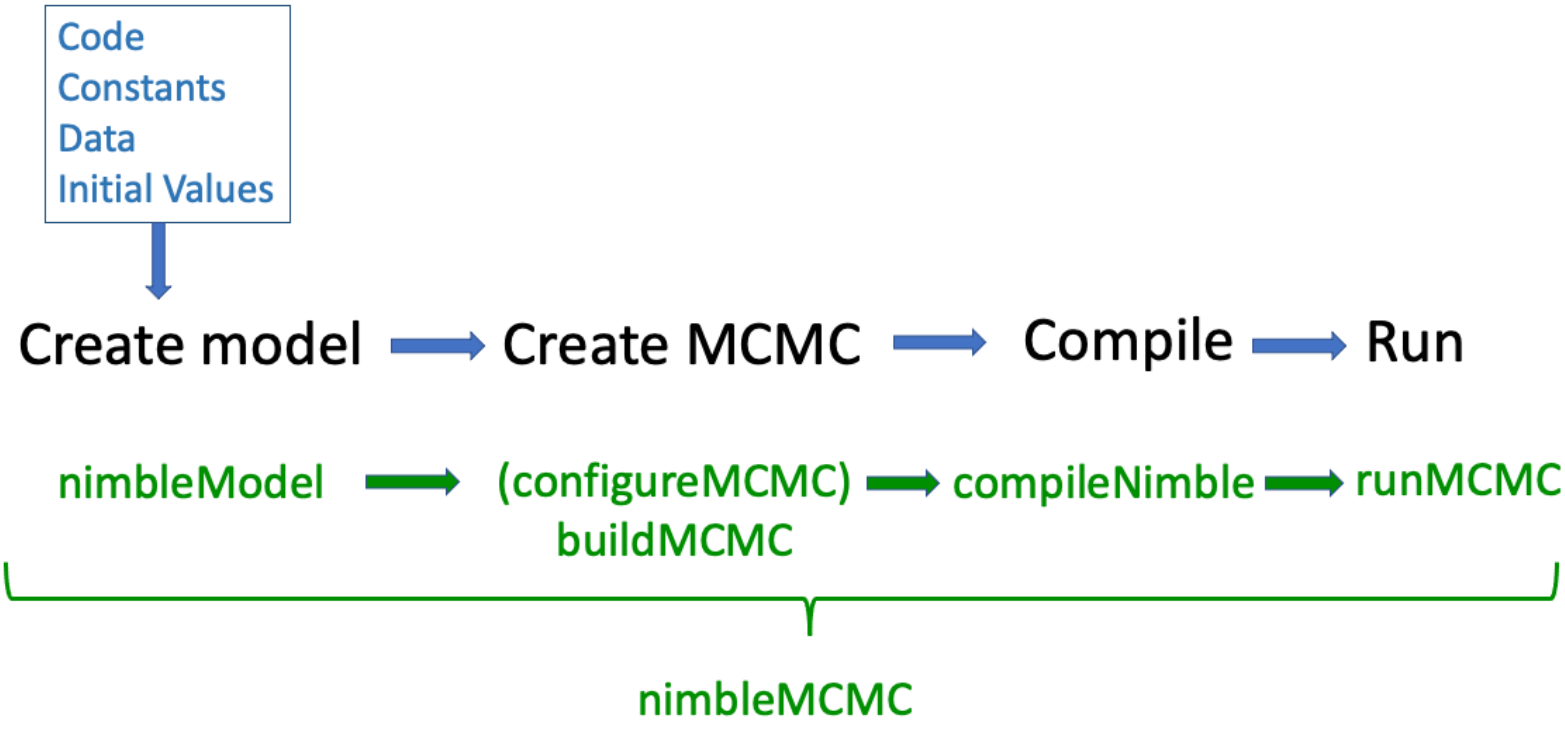
```
##      year_1981 year_1982 year_1983 year_1984 year_1985 y
## [1,]         2         2         2         2         2
## [2,]         2         2         2         2         2
## [3,]         2         2         2         2         1
## [4,]         2         2         2         2         1
## [5,]         2         2         1         2         2
## [6,]         2         2         1         1         1
```

Conventional Hidden Markov model code, slightly updated.

- data, y: 1 = not-detected, 2 = detected.
- latent states, z: 1 = alive, 2 = dead. (Following convention that "dead" is the last state.)
- Modified from Gimenez et al. capture-recapture workshop

```
dipper_code_dcat <- nimbleCode({
  phi ~ dunif(0, 1) # prior survival
  p ~ dunif(0, 1) # prior detection
  # likelihood
  gamma[1,1:2] <- c(phi, 1-phi)       # Pr(alive t -> alive
  gamma[2,1:2] <- c(0, 1)             # Pr(dead t -> alive
  delta[1:2] <- c(1, 0)               # Pr(alive t = 1) = 1
  omega[1,1:2] <- c(1 - p, p)         # Pr(alive t -> non-d
  omega[2,1:2] <- c(1, 0)             # Pr(dead t -> non-de
  for (i in 1:N){
    z[i,first[i]] ~ dcat(delta[1:2]) # Illustrates initial
    for (j in (first[i]+1):T){
      z[i,j] ~ dcat(gamma[z[i,j-1], 1:2])
      y[i,j] ~ dcat(omega[z[i,j], 1:2])
```

Basic nimble workflow



Setup data, constants, and inits

```
zinits <- matrix(2, nrow = nrow(y), ncol = ncol(y)) # crea
zdata <- matrix(NA, nrow = nrow(y), ncol = ncol(y)) # crea
for(i in 1:nrow(zinits)) {
  known_alive <- range(which(y[i,] == 2))
  zinits[i, known_alive[1] : known_alive[2] ] <- NA # inits
  zdata[i, known_alive[1] : known_alive[2] ] <- 1   # data
}
dipper_constants <- list(N = nrow(y),
                         T = ncol(y),
                         first = first)
dipper_data <- list(y = y,
                    z = zdata)
dipper_inits <- function() list(phi = runif(1,0,1),
                                p = runif(1,0,1),
                                z = zinits)
head(dipper_data$z)      # data and inits have comlpementary
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
```

What are constants? What are data?

Constants are values needed to define model relationships

- Index starting/ending values like `N`
- Constant indexing vectors for indexing data groupings (site, treatment, individual, time): `beta[ treatment[i] ]`.
- Constants must be provided when creating a model with `nimbleModel`.

Data represents a flag on the role a node plays in the model

- E.g., data nodes shouldn't be sampled in MCMC.
- Data *values* can be changed.
- Data can be provided when calling `nimbleModel` or later.

Providing data and constants together.

- Data and constants can be provided together **`as constants`**.
- It would be slightly easier for BUGS/JAGS users to call this "data", but that would blur the concepts.
- NIMBLE will usually disambiguate data and constants when they are provided together as `constants`.

What are covariates and other

Constants vs data for nested indexing

When values are grouped (particularly in irregular ways), we often have (potentially complicated) indexing.

Here is a model code snippet from a GLMM for disease occurrence in deer from different farms.

- `sex` is coded as 1 or 2 to index fixed-effect intercepts.
- `farm_ids` are coded as 1-24 for random-effect intercepts for data from 24 farms.
- Membership indices that are known in advance.
- **Provide known indexing vectors in `constants`.**
- Otherwise `nimble` allows that you (or your model) might change them later, which makes it handle them inefficiently (with unnecessary computation).

```
nimbleCode({
  # ...incomplete code snippet...
  for(i in 1:2) sex_int[i] ~ dnorm(0, sd = 1000)   # prior
  for(i in 1:num_farms) farm_effect[i] ~ dnorm(0, sd = farm
  farm_sd ~ dunif(0, 100)                          # prior
```

Create a model

```
dipper_model <- nimbleModel(code = dipper_code_dcat,
                            constants = dipper_constants,
                            data = dipper_data,      # data
                            inits = dipper_inits()   # inits
                            )                        # dimen
```

```
## Defining model

## Building model

## Setting data and initial values

## Running calculate on model
##   [Note] Any error reports that follow may simply reflec

## Checking model sizes and dimensions
```

Create an MCMC

```
dipper_MCMCconf <- configureMCMC(dipper_model, monitors = c
```

```
## ===== Monitors =====
## thin = 1: p, phi
## ===== Samplers =====
## RW sampler (2)
##   - phi
##   - p
## categorical sampler (613)
##   - z[]  (613 elements)
```

```
dipper_MCMC <- buildMCMC(dipper_MCMCconf)
```

Compile the model and MCMC

3. Compile the model and MCMC

```
C_dipper_model <- compileNimble(dipper_model) # These two
```

```
## Compiling
##   [Note] This may take a minute.
##   [Note] Use 'showCompilerOutput = TRUE' to see C++ comp
```

```
C_dipper_MCMC <- compileNimble(dipper_MCMC, project = dippe
```

```
## Compiling
##   [Note] This may take a minute.
##   [Note] Use 'showCompilerOutput = TRUE' to see C++ comp
```

4. Run the MCMC

```
samples <- runMCMC(C_dipper_MCMC, niter = 10000, samplesAs(
```

```
## Running chain 1 ...
## |-------------|-------------|-------------|-------------
## |-------------------------------------------------------
```

```
# Alternative:
```

How can I use the model in R?

```
## Defining model

## Building model

## Setting data and initial values

## Running calculate on model
##   [Note] Any error reports that follow may simply reflec

## Checking model sizes and dimensions

## Compiling
##   [Note] This may take a minute.
##   [Note] Use 'showCompilerOutput = TRUE' to see C++ comp
class(dipper_model)[1]  # This is a reference class (S5) o

## [1] "dipper_cod_MID_2_modelClass_UID_17_UID_18"
dipper_model$gamma              # Look at a model variable,

##           [,1]      [,2]
```

NIMBLE might insert nodes into your model!

These are called *lifted nodes*.

Example 1: reparameterization

You give NIMBLE this:

```
nimbleCode({
  tau <- 1E-0.6
  mu ~ dnorm(0, tau)
})
```

- NIMBLE defaults to parameterizations from **WinBUGS/OpenBUGS/JAGS, not R**.
- Default SD/Var/precision for `dnorm` is **precision** = 1/variance.
- NIMBLE converts this to a *canonical* parameterization for computations by treating it like this:

```
nimbleCode({
  tau <- 1E-0.6
  some_long_name_created_by_nimble <- 1/sqrt(tau) # a lift
  mu ~ dnorm(0, sd = some_long_name_created_by_nimble)
})
```

How can I use the MCMC configuration in R?

- Change the set of samples that compose an MCMC algorithm.
- See `help(samplers)` for samplers built in to `nimble`.
- Default sampler assignments:
  - Conjugate (Gibbs) sampler when possible.
  - Special samplers for Bernoulli, categorical, Dirichlet, multinomial, possibly others.
  - Slice samplers for discrete distributions such as Poisson.
  - Adaptive random-walk Metropolis-Hastings samplers for other continuous distributions.
- MCMC efficiency = Effective sample size (mixing) / computation time
  - Both speed and mixing matter.
  - There is often a tradeoff between these.
- Some ways to customize samplers
  - Block (jointly) sample correlated dimensions.
  - Block (joint) samplers include adaptive random-walk

How can I use uncompiled vs. compiled models and algorithms?

- An important and perhaps unfamiliar principle:
  - (Almost) everything can be run **uncompiled** (in R) or **compiled** (in C++).
- Uncompiled use of models and algorithms in R allows debugging.
  - Behavior is not always identical but is close.
  - Example: Error trapping will behave differently. Errors in C++ might not occur in R.

Differences from NIMBLE to JAGS and/or BUGS

- See our guide on converting from JAGS to NIMBLE
- Wrap your model code in `nimbleCode({})` directly in R.
- Provide information about missing or empty indices:
  - Use `x[1:n, 1:m]` or
  - `x[,]` with `dimensions = list(x = c(n,m))`.
  - Do not use `x` without brackets (unless it is a scalar).
- Decide how much control you need:
  - `nimbleMCMC` will do everything.
  - `nimbleModel`, `configureMCMC`, `buildMCMC`, `compileNimble` and `runMCMC` give you more control.