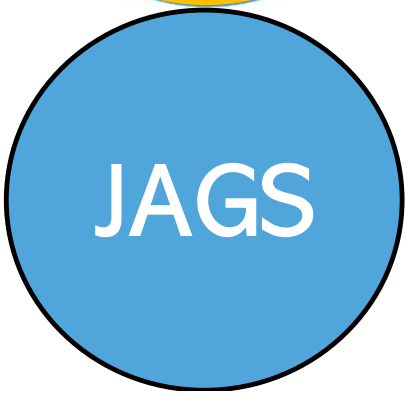
Nimble and You:

An incomplete primer on what you may want to know if you decide to move to Nimble for your Bayesian Hierarchical Programming needs

By Daniel Gibson
with edits from Alec Schindler

Obligatory Comparison Slide- JAGS

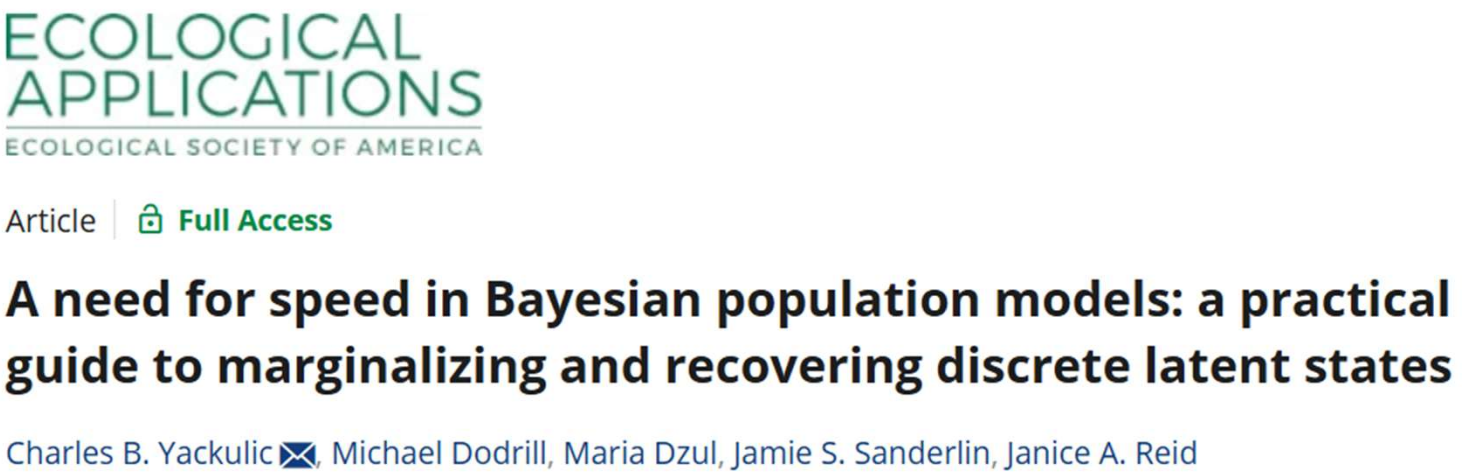
- JAGS [**J**ust **A**nother **G**ibbs **S**ampler] can be viewed as the spiritual successor to WinBUGS (the *true* successor, OpenBUGS, never really took off).
- Its development/release coincided with the pronounced shift in ecology from nearly entirely frequentist to a more mixed Bayesian/Frequentist approach to model building. This is why everyone's Bayesian-friendly PI uses or at least understands the JAGS language.
- Pretty much a one-man show (Martyn Plummer) and recent updates to the JAGS program have been infrequent.



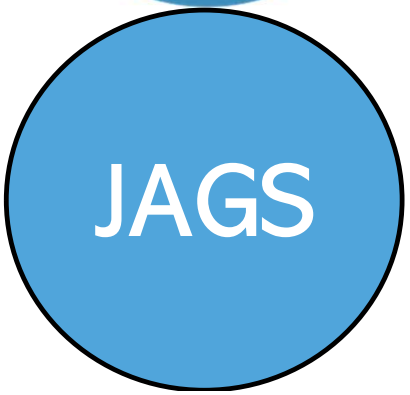
Obligatory Comparison Slide- Stan

- Stan is *NOT* a Gibbs sampler. It is based on the No-U-Turn Sampler [NUTS] algorithm developed by Andrew Gelman and Matt Hoffman that uses a derivative of the general MCMC process (**Hamiltonian Monte Carlo [HMC])** to more efficiently resolve parameter correlations in high dimensional spaces.
  - The correlations substantially impact the performance of the random-walk nature of MCMC-based Gibbs samplers.
- The Stan development team is enormous, and the team is well funded. There is an extremely active forum.
- The Stan programming language is substantially different from JAGS.
- The NUTS-HMC process does not allow for the direct estimation of discrete latent parameters (e.g., basically what I need). NUTS requires that these types of models “marginalized the likelihood”, which can be a minor headache.
- Stan has useful built in cross validation tools.

Guide for what “Marginalizing the Likelihood” means in a capture-mark-recapture context



https://doi.org/10.1002/eap.2112

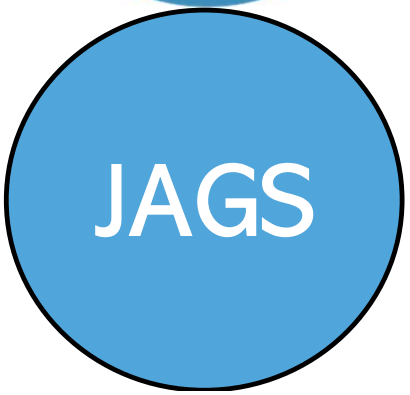


Obligatory Comparison Slide – Nimble

- Nimble [**N**umerical **I**nference for statistical **M**odels for **B**ayesian and **L**ikelihood **E**stimation] is a flexible programming tool released within the last decade. Like Stan, there is a fully formed development team led by Perry de Valpine and Chris Paciorek at UCB and Daniel Turek at Williams College.

- Nimble Dev team focuses on improving the functionality, flexibility, and breadth of Bayesian analysis.

- The Nimble google group is a very useful forum to ask questions or see other people's issues: https://groups.google.com/g/nimble-users.



Obligatory Comparison Slide – Nimble

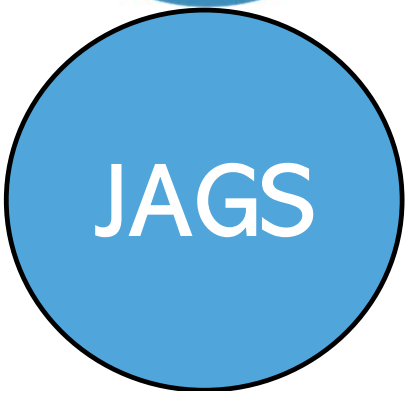
- Unlike JAGS and Stan, one of the main developers is an ecologist.

THE ROYAL SOCIETY

A new demographic function maximized by life-history evolution

Perry de Valpine

Department of Environmental Science and Policy, Center for Population Biology, and Institute for Theoretical Dynamics, University of California, One Shields Avenue, Davis, CA 95616, USA



Obligatory Comparison Slide – Nimble

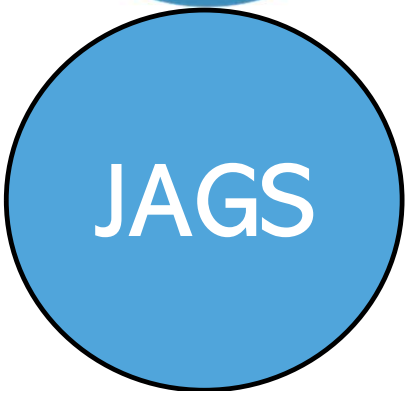
- Unlike JAGS and Stan, one of the main developers is an ecologist.

Ecological Monographs, 72(1), 2002, pp. 57–76

FITTING POPULATION MODELS INCORPORATING PROCESS NOISE AND OBSERVATION ERROR

PERRY DE VALPINE[1] AND ALAN HASTINGS

Department of Environmental Science and Policy, Center for Population Biology and Institute for Theoretical Dynamics, University of California, One Shields Avenue, Davis, California 95616 USA



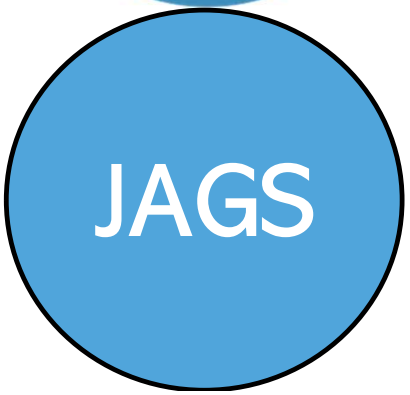
Obligatory Comparison Slide – Nimble

- Unlike JAGS and Stan, one of the main developers is an ecologist.

VOL. 172, NO. 4 THE AMERICAN NATURALIST OCTOBER 2008

Conspecific Brood Parasitism and Population Dynamics

Perry de Valpine[1,*] and John M. Eadie[2,†]



Obligatory Comparison Slide – Nimble

- Nimble is a much more flexible programming tool. Similar to Stan, there is a fully formed development team led by Perry de Valpine and Chris Paciorek at UCB and Daniel Turek at Williams College.
- The Nimble Dev ***focuses on improving the functionality and breadth of Bayesian analysis***.
- The Nimble google group is a very useful forum to ask questions or see other people's issues: https://groups.google.com/g/nimble-users.

Improving Functionality:

- Nimble is inherently computationally faster than JAGS as it compiles the entire model to the C++ language, which is much more optimized than R.
  - Stan is inherently more 'efficient' when compared to Nimble's MCMC algorithms.
- JAGS often has a hard time compiling and running extremely large models.
  - Large models often *appear* to get stuck in the compilation phase. I have never had a model fail to compile in Nimble.
- Nimble also allows users to separate the 'model-building' process into a series of discrete steps, which allows users to clear RAM between steps for large models.
  - Compile model to C++ -> configure MCMC parameters -> compile MCMC parameters to C++ -> run MCMC samplers

Improving Functionality:

- Users can create “functions” and user-defined distributions that can have enormous gains to efficiency.
- Nimble also allows for the user to choose what samplers to use for specific parameters within a model, which can substantially improve the efficiency of the MCMC process.
  - Nimble now allows for the use of NUTS-HMC and traditional MCMC samplers within the same model.

```
ddirchmulti <- nimbleFunction(
  run = function(x = double(1), alpha = double(1), size = double(0),
                 log = integer(0, default = 0)) {
    returnType(double(0))
    logProb <- lgamma(size+1) - sum(lgamma(x+1)) + lgamma(sum(alpha)) -
      sum(lgamma(alpha)) + sum(lgamma(alpha + x)) -
      lgamma(sum(alpha) + size)
    if(log) return(logProb)
    else return(exp(logProb))
  })

rdirchmulti <- nimbleFunction(
  run = function(n = integer(0), alpha = double(1), size = double(0)) {
    returnType(double(1))
    if(n != 1) print("rdirchmulti only allows n = 1; using n = 1.")
    p <- rdirch(1, alpha)
    return(rmulti(1, size = size, prob = p))
  })
```

Improving Functionality: Samplers

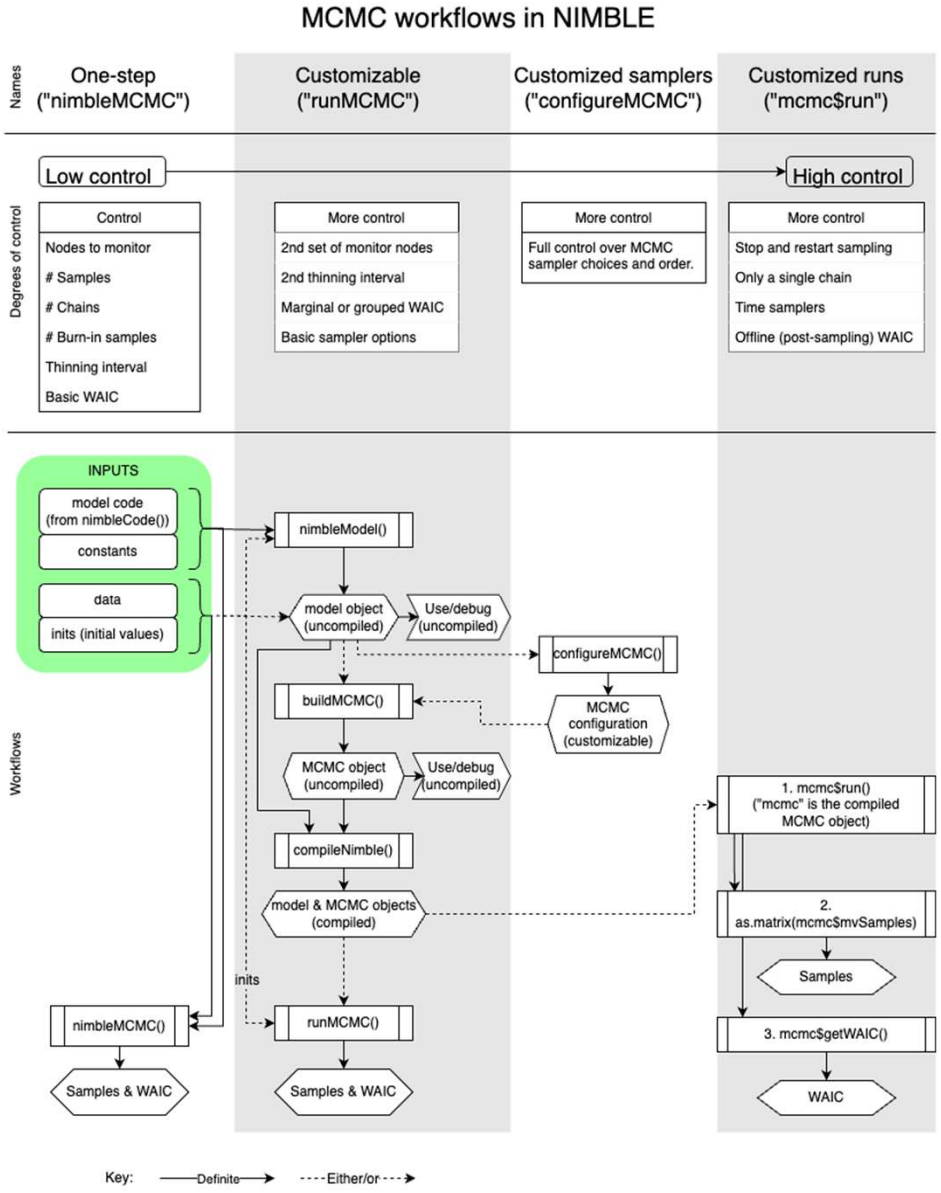
- Nimble also allows for the user to choose what samplers to use for specific parameters within a model, which can substantially improve the efficiency of the MCMC process.
- **Description of samplers:**
  - https://rdrr.io/cran/nimble/man/samplers.html
- **Default samplers:**
  - Conjugate (Gibbs) sampler when possible
  - Special samplers for Bernoulli, categorical, Dirichlet, multinomial, CAR, BNP, possibly others.
  - Slice samplers for discrete distributions such as Poisson.
  - Adaptive random-walk Metropolis-Hastings samplers for other continuous distributions.

Improving Functionality: Samplers

- Nimble also allows for the user to choose what samplers to use for specific parameters within a model, which can substantially improve the efficiency of the MCMC process.
- **Automatic Factor Slice Sampler:**
  - https://r-nimble.org/blog/better-block-sampling-in-mcmc-with-the-automated-factor-slice-sampler.html
  - Type of “block sampler” – useful when parameters are correlated, helps these parameters be more identifiable
- **Using NUTS in Nimble:**
  - https://danielturek.r-universe.dev/nimbleHMC/doc/manual.html#buildHMC
  - Nimble now allows for the use of NUTS-HMC and traditional MCMC samplers within the same model
  - Typically achieves convergence with fewer iterations, but takes longer to run per iteration
  - Not available for discrete parameters

Nimble workflow

1) Write model code, define constants, data, and initial values
2) Create model object (it is an R object)
3) Build MCMC
   a. Configure MCMC
   b. Customize MCMC
   c. Build MCMC
4) Compile the model and MCMC
5) Run MCMC
6) Extract results



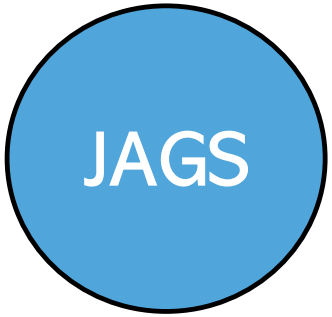
From: https://github.com/nimble-training/nimble-virtual-2023

Data vs. constants

- **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** (e.g., what BUGS/JAGS calls "data".
  - NIMBLE will usually disambiguate data and constants when they are provided together as constants.

First steps: How to translate a BUGS or JAGS model into the Nimble language or write your first Nimble Model

- https://r-nimble.org/examples/nimble_build_a_model.html
- https://r-nimble.org/examples/converting_to_nimble.html
- https://r-nimble.org/examples/nimble_basic_mcmc.html




```
# Specify model in JAGS language
sink("cjs-c-c.jags")
cat("
model {

# Priors and constraints
for (i in 1:nind){
   for (t in f[i]:(n.occasions-1)){
      phi[i,t] <- mean.phi
      p[i,t] <- mean.p
      } #t
   } #i

mean.phi ~ dunif(0, 1)         # Prior for mean survival
mean.p ~ dunif(0, 1)           # Prior for mean recapture

# Likelihood
for (i in 1:nind){
   # Define latent state at first capture
   z[i,f[i]] <- 1
   for (t in (f[i]+1):n.occasions){
      # State process
      z[i,t] ~ dbern(mu1[i,t])
      mu1[i,t] <- phi[i,t-1] * z[i,t-1]
      # Observation process
      y[i,t] ~ dbern(mu2[i,t])
      mu2[i,t] <- p[i,t-1] * z[i,t]
      } #t
   } #i
}

",fill = TRUE)
sink()
```

```
#####################################
# Specify model in Nimble language
cjs_c_c <- nimbleCode({

  # Priors and constraints
  for (i in 1:nind){
    for (t in f[i]:(n.occasions-1)){
          phi[i,t] <- mean.phi
            p[i,t] <- mean.p
    } #t
  } #i

  mean.phi ~ dunif(0, 1)          # Prior for mean survival
   mean.p ~ dunif(0, 1)           # Prior for mean recapture

  # Likelihood
  for (i in 1:nind){
    # Define latent state at first capture
    z[i,f[i]] <- 1
    for (t in (f[i]+1):n.occasions){
      # State process
        z[i,t] ~ dbern(mu1[i,t])
      mu1[i,t] <- phi[i,t-1] * z[i,t-1]
      # Observation process
        y[i,t] ~ dbern(mu2[i,t])
      mu2[i,t] <- p[i,t-1] * z[i,t]
    } #t
  })
```

Main differences between Nimble and JAGS model code

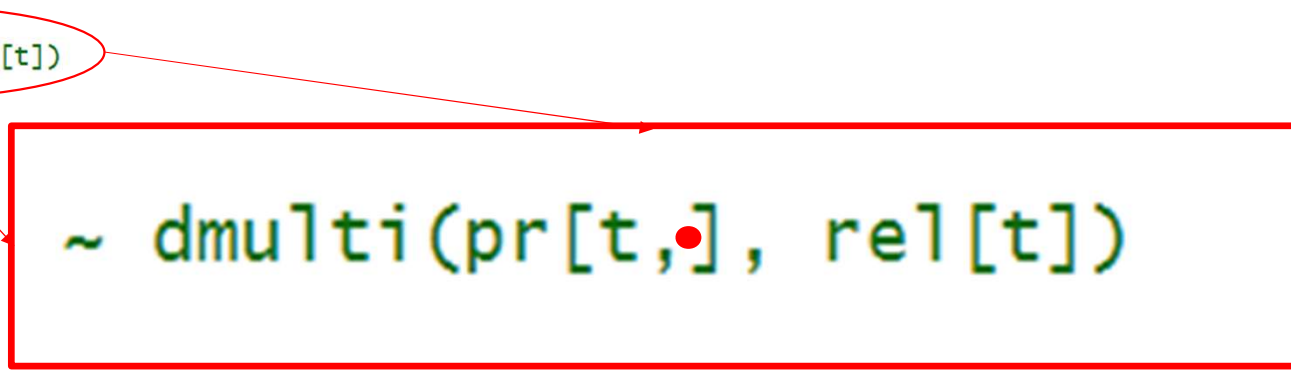
- Nimble does not support ‘empty indexing’, whereas JAGS does.

Empty indexing

```
# Specify mark-recovery model in BUGS language
sink("mr-mnl.jags")
cat("
model {

# Priors and constraints
for (t in 1:n.occasions){
   s[t] <- mean.s
   r[t] <- mean.r
   }
mean.s ~ dunif(0, 1)              # Prior for mean survival
mean.r ~ dunif(0, 1)              # Prior for mean recovery

# Define the multinomial likelihood
for (t in 1:n.occasions){
   marr[t,1:(n.occasions+1)] ~ dmulti(pr[t,], rel[t])
   }
# Define the cell probabilities of the m-array
# Main diagonal
for (t in 1:n.occasions){
   pr[t,t] <- (1-s[t])*r[t]
   # Above main diagonal
   for (j in (t+1):n.occasions){
      pr[t,j] <- prod(s[t:(j-1)])*(1-s[j])*r[j]
      } #j
   # Below main diagonal
   for (j in 1:(t-1)){
      pr[t,j] <- 0
      } #j
   } #t
# Last column: probability of non-recovery
for (t in 1:n.occasions){
   pr[t,n.occasions+1] <- 1-sum(pr[t,1:n.occasions])
   } #t
}
",fill = TRUE)
```



Empty indexing

```
# Specify mark-recovery model in BUGS language
mr_mnl <- nimbleCode({

# Priors and constraints
for (t in 1:n.occasions){
   s[t] <- mean.s
   r[t] <- mean.r
   }
mean.s ~ dunif(0, 1)              # Prior for mean survival
mean.r ~ dunif(0, 1)              # Prior for mean recovery

# Define the multinomial likelihood
for (t in 1:n.occasions){
   marr[t,1:(n.occasions+1)] ~ dmulti(pr[t,1:(n.occasions+1)], rel[t])
   }
# Define the cell probabilities of the m-array
# Main diagonal
for (t in 1:n.occasions){
   pr[t,t] <- (1-s[t])*r[t]
   # Above main diagonal
   for (j in (t+1):n.occasions){
      pr[t,j] <- prod(s[t:(j-1)])*(1-s[j])*r[j]
      } #j
   # Below main diagonal
   for (j in 1:(t-1)){
      pr[t,j] <- 0
      } #j
   } #t
# Last column: probability of non-recovery
for (t in 1:n.occasions){
   pr[t,n.occasions+1] <- 1-sum(pr[t,1:n.occasions])
   } #t
})
```

```
~ dmulti(pr[t,1:(n.occasions+1)], rel[t])
```

Main differences between Nimble and JAGS model code

- Nimble does not support ‘empty indexing’, whereas JAGS does.
- Nimble allows for multiple parameterizations for each distribution, JAGS generally just has one parameterization for a specific distribution.

Parameterizations of Distributions

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# example of a random year effect in JAGS
tau.eps <- 1/ (sd.eps * sd.eps)
  sd.eps ~ dunif(0, 5)

for (i in 1:nyear){
  eps[i] ~ dnorm(0, tau.eps)      # Random year effects
}
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
```

Parameterizations of Distributions

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# example of a random year effect in JAGS
tau.eps <- 1/ (sd.eps * sd.eps)
  sd.eps ~ dunif(0, 5)

for (i in 1:nyear){
  eps[i] ~ dnorm(0, tau.eps)      # Random year effects
}
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# first example of a comparable random year effect in Nimble
tau.eps <- 1/ (sd.eps * sd.eps)
sd.eps ~ dunif(0, 5)

for (i in 1:nyear){
  eps[i] ~ dnorm(0, tau = tau.eps)     # Random year effects
}
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
```

Parameterizations of Distributions

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# example of a random year effect in JAGS
tau.eps <- 1/ (sd.eps * sd.eps)
  sd.eps ~ dunif(0, 5)

for (i in 1:nyear){
  eps[i] ~ dnorm(0, tau.eps)      # Random year effects
}
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# first example of a comparable random year effect in Nimble
tau.eps <- 1/ (sd.eps * sd.eps)
sd.eps ~ dunif(0, 5)

for (i in 1:nyear){
  eps[i] ~ dnorm(0, tau = tau.eps)     # Random year effects
}
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
# second example of a comparable random year effect in Nimble
sd.eps ~ dunif(0, 5)

for (i in 1:nyear){
  eps[i] ~ dnorm(0, sd = sd.eps)       # Random year effects
}
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~#
```

Main differences between Nimble and JAGS model code

- Nimble does not support ‘empty indexing’, whereas JAGS does.
- Nimble allows for multiple parameterizations for each distribution, JAGS generally just has one parameterization for a specific distribution.
- Truncation, censoring, and constraints are coded slightly different.

```
half_normal_jags ~ dnorm(0, tau) T(0, )
half_normal_nimble ~ T(dnorm(0, sd = sigma), 0, )
```

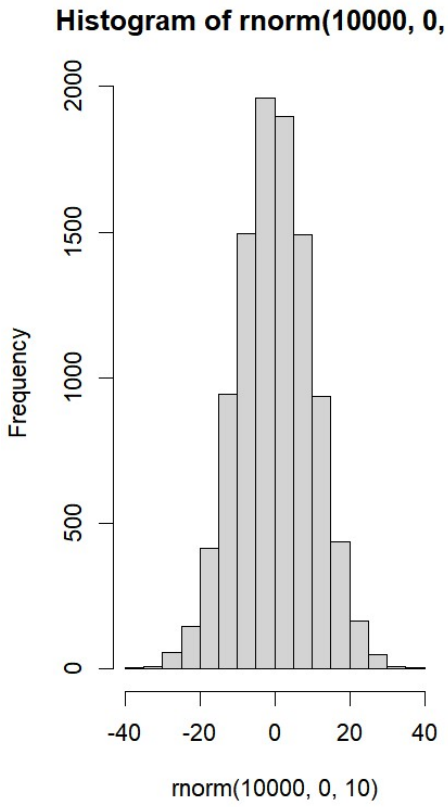
Main differences between Nimble and JAGS model code

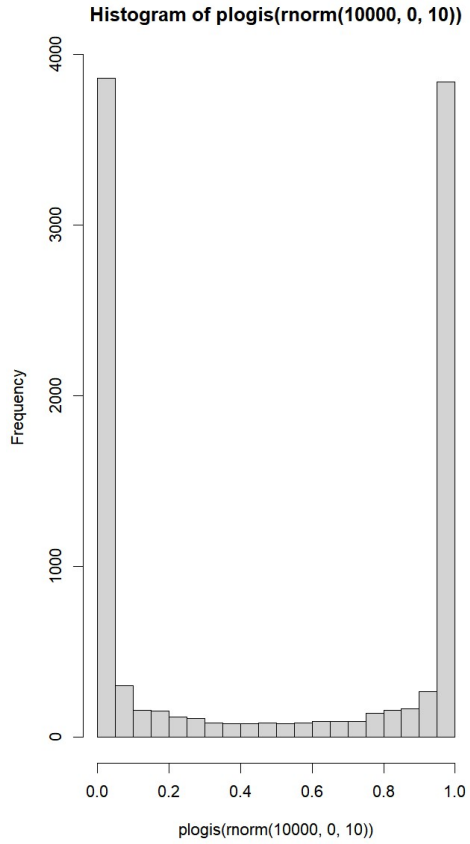
- Nimble does not support ‘empty indexing’, whereas JAGS does.
- Nimble allows for multiple parameterizations for each distribution, JAGS generally just has one parameterization for a specific distribution.
- Truncation, censoring, and constraints are coded slightly different.

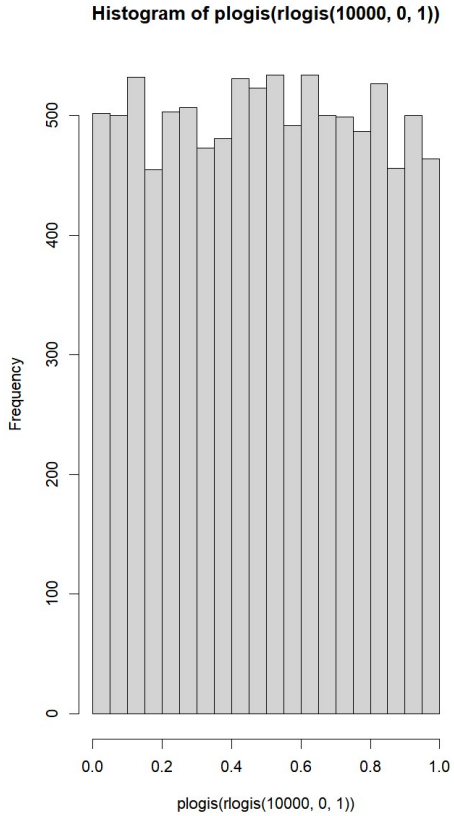
```
half_normal_jags ~ dnorm(0, tau) T(0, )
half_normal_nimble ~ T(dnorm(0, sd = sigma), 0, )
```

- **Initial values**

Initial Values: In reality, you should provide prior values for all estimated parameters that ‘work’.







Initial Values: In reality, you should provide prior values for all estimated parameters that ‘work’.

I cannot stress this enough. Provide initial values for everything

How do you know if your initial values worked?

- **Step 1: Build the model based on current model code, constants, data, and initial values.**

```
model <- nimbleModel( code = nimble_model, constants = nimble_cons,  dat =  nimble_data, inits = inits)
```

- **Step 2: Initialize the model**

```
model$initializeInfo()
```

  - All parameters initialized? Proceed to step 3.
  - Some number of estimated parameters are not initialized, go back to step 1 but add initial values for the 'missing' parameters *or* use the $simulate function in nimble to simulate initial values for these parameters and proceed with caution.

- **Step 3: Calculate the log-likelihood based on the current initial values**

```
model$calculate()
```

  - Negative number: Good to go.
  - NA: Proceed with caution or double check that an initial value was not forgotten for a parameter.
  - -Inf: Initial values are resulting in an impossible estimate or the model is wrong

Running parallel chains in Nimble?

- It's currently not native, but you can run parallel chains of Nimble models in R using libraries like `parallel`.

- Create a native parallel processing function is on the list of tasks that the Nimble devs want to complete soon.

- Until then, this approach works:
  - http://danielturek.github.io/public/parallelMCMC/parallelMCMC.html

```
inits_1 <- initsFunction()
inits_2 <- initsFunction()
inits_3 <- initsFunction()
# If you are running this model on a cluster and your initsFunction is complex, it may make more sense to generate some random initial values ahead of time
inits.prime <- list(inits_1,inits_2, inits_3)

nc <- 3 # number of chains

cl <-makeCluster(nc,timeout=5184000)

clusterExport(cl, c("nimble_model", "initsFunction", "nimble_data", "nimble_cons",'inits.prime'))

for (j in seq_along(cl)) {
  inits <- inits.prime[[j]]
  clusterExport(cl[j], "inits")
}
out <- clusterEvalQ(cl, {
  # the libraries you need must be included within the parallel function
  library(nimble)
  library(coda)
  library(parallel)
  model <- nimbleModel( code = nimble_model, constants = nimble_cons,  dat =  nimble_data, inits = inits, buildDerivs = TRUE)

  model$initializeInfo() # see what parameters are not initialized

  model$simulate(c(      # add the parameters you need to simulate to complete the initialize step here

  model$initializeInfo() # is everything initialized?
  model$calculate()      # Did you receive a -LL? NAs are a warning, -Inf indicates you need to make a change

  modelConf  <- configureMCMC(model, useConjugacy = FALSE,  thin = 2, thin2 = 5,
                              monitors =  c( # add the primary parameters you want to monitor here
                              ),

                              monitors2 = c( # add the other parameters you want to monitor here
                              ))

  #~~~~~~~~~~~~~~~~~~#
  # Add all your changes to the samplers here.
  #
  #~~~~~~~~~~~~~~~~~~#

  modelMCMC  <- buildMCMC(modelConf)
  Cmodel     <- compileNimble(model)
  CmodelMCMC <- compileNimble(modelMCMC)

  CmodelMCMC$run(2500, thin = 2, thin2 = 5)

  return(list( as.mcmc(as.matrix(CmodelMCMC$mvSamples)),
               as.mcmc(as.matrix(CmodelMCMC$mvSamples2))))

})

samples1 <- list(chain1 =   out[[1]][[1]][-c(1:101),],  chain2 =   out[[2]][[1]][-c(1:101),], chain3 =   out[[3]][[1]][-c(1:101),])
samples2 <- list(chain1 =   out[[1]][[2]][-c(1:41),],  chain2 =   out[[2]][[2]][-c(1:41),], chain3 =   out[[3]][[2]][-c(1:41),])

mcmcList1 <- as.mcmc.list(lapply(samples1, mcmc))
mcmcList2 <- as.mcmc.list(lapply(samples2, mcmc))
```

Creating the parallel process

Running a Nimble Model

Ending the parallel process

Working with models that take days/weeks to run? Consider a protocol that pauses and restarts run.

- Here are some guides to restart previously run models where they ‘ended’
  - http://danielturek.github.io/public/restartingMCMC/restartingMCMC.html
  - https://danielturek.github.io/public/saveMCMCstate/saveMCMCstate.html

Other resources

- Nimble user manual: https://r-nimble.org/manual/NimbleUserManual.pdf.
- Nimble cheat sheet: https://r-nimble.org/NimbleCheatSheet.pdf.