A Modeler's Primer on JAGS

N. Thompson Hobbs[1] and Christian Che-Castaldo[2]

[1]Natural Resource Ecology Laboratory, Department of Ecosystem Science and Sustainability, and Graduate Degree Program in Ecology, Colorado State University, Fort Collins CO, 80523

[2]Mount St. Helens Institute, Amboy WA, 98601

May 25, 2019

Contents

List of Algorithms

List of Exercises

1 Aim

The purpose of this Primer is to teach the programming skills needed to approximate the marginal posterior distributions of parameters, latent variables, and derived quantities of interest using software implementing Markov chain Monte Carlo methods. Along the way, we will reinforce some of the ideas and principles that we have learned in lecture. The Primer is organized primarily as a tutorial and contains only a modicum of reference material.[1] You will need to study the JAGS manual at some point to become a fully competent programmer.

2 Introducing MCMC Samplers

WinBugs, OpenBUGS, and JAGS are three systems of software that implement Markov chain Monte Carlo sampling using the BUGS language. BUGS stands for Bayesian Analysis Using Gibbs Sampling, so you can get an idea what this language does from its name. Imagine that you took the MCMC code you wrote for a Gibbs sampler and tried to turn it into an R function for building chains of parameter estimates. Actually, you know enough now to construct a very general tool that would do this. However, you are probably delighted to know that you can accomplish the same thing with less time and effort using the BUGS language.

OpenBUGS and WinBUGS run on Windows operating systems, while JAGS was specifically constructed to run multiple platforms, including macOS and Linux. Although all three programs use essentially the same syntax, OpenBUGS and WinBUGS run in an elaborate graphical user interface, while JAGS only runs from the command line of a Unix shell or from R. However, all three can be easily called from R, and this is the approach we will teach. Our experience is that the GUI involves far too much tedious pointing and clicking

[1] Other good references on the BUGS language include McCarthy (2007) and the JAGS manual. The JAGS manual can be a bit confusing because it is written as if you were going to use the software stand alone, that is, from a UNIX command line, which is one of the reasons we wrote this tutorial. However, it contains very useful information on functions and distributions that is not covered in detail here. You need a copy of the JAGS manual to supplement the material here.

and doesn't provide the flexibility that is needed for serious work.

There are two other options for software to approximate marginal posterior distributions, which are faster than any of the BUGS implementations. The first is Stan, which is definitely worth looking at after you have become comfortable with JAGS. We don't teach it because it uses an algorithm (Hamiltonian MCMC), that is more difficult to understand than conventional MCMC and because most published books on Bayesian analysis use some form of the BUGS language.[2] The general conclusion on the street is that JAGS is easier to implement for simple problems; Stan is faster for more complex ones. Once you have learned JAGS it is easy to migrate to Stan if it proves attractive to you.

The other option is INLA, which is dramatically faster than any MCMC method because it doesn't use sampling, but rather approximates the marginal distribution of the data. It is not for the new initiate to Bayesian analysis. It applies to a somewhat restricted set of problems.

3 Introducing JAGS

We will use JAGS, which stands somewhat whimsically for "Just Another Gibbs Sampler". There are three reasons for using JAGS as the language for this course. First and most important, is because our experience is that JAGS is less fussy than WinBUGS (or OpenBUGS), which can be notoriously difficult to debug. JAGS runs on all platforms, which makes collaboration easier. JAGS has some terrific features and functions that are absent from other implementations of the BUGS language. That said, if you learn JAGS you will have no problem interpreting code written for WinBugs or OpenBUGS (for example, the programs written in McCarthy 2007). The languages are almost identical except that JAGS is better.[3]

[2]The most recent ones give examples in both languages.

[3]There is a nice section in the JAGS manual on differences between WinBUGS and JAGS. There is also software called GeoBUGS) that is specifically developed for spatial models, but we know virtually nothing about it. However, if you are interested in landscape or spatial ecology, we urge you to look into it.

This tutorial will use a simple example of linear regression as a starting point for teaching the BUGS language implemented in JAGS and associated R commands. Although the problem starts simply, it builds to include some fairly sophisticated analysis in the problem set you will do after completing the tutorial.

The model that we will use is the linear relationship between the per-capita rate of population growth and the size a population, which, as you know, is the starting point for deriving the logistic equation[4]. For the ecosystem scientists among you, this problem is easily recast as the mass specific rate of accumulation of nitrogen in the soil; see for example Knops and Tilman (2000). Both the population example and the ecosystem example can happily use the symbol N to represent the state variable of interest. Social scientists will be delighted to know that the logistic equation was originally applied to portray human population growth (see here).

Consider a model predicting the per-capita rate of population growth (or the mass specific rate of nitrogen accumulation),

$$\frac{1}{N}\frac{dN}{dt} = r - \frac{r}{K}N, \tag{1}$$

which, of course, is a linear model with intercept r and slope $\frac{r}{K}$. Note that these quantities enjoy a sturdy biological interpretation in population ecology; r is the intrinsic rate of increase[5], $\frac{r}{K}$ is the strength of the feedback from population size to population growth rate, and K is the carrying capacity, that is, the population size (o.k., o.k., the gm N per gm soil for the ecosystem scientists) at which $\frac{dN}{dt} = 0$. Presume we have some data consisting of observations of per capita rate of growth paired with observations of N. The vector $\mathbf{y}$ contains values for the rate and the vector $\mathbf{x}$ contains aligned data on N, i.e., $y_i = \frac{1}{N_i}\frac{dN_i}{dt}$, $x_i = N_i$. To keep things simple, we start out by assuming that the x_i are measured without error. A simple Bayesian model specifies the joint distribution of the parameters and data as:

[4]Any of you claiming to be population ecologists must be able to derive the familiar, differential equation for population growth rate $\frac{dN}{dt} = rN(1 - \frac{N}{K}$ from equation 1.

[5]Defined as the per capita rate of increase when population size is near 0, such that there is no competition for resources. The quantity r is a characteristic of the physiology and life history of the organism.

$$\mu_i = r - \frac{r x_i}{K}, \tag{2}$$

$$[r, K, \tau \mid \mathbf{y}] \propto \prod_{i=1}^{n} [y_i \mid \mu_i, \tau] [r] [K] [\tau], \tag{3}$$

$$[r, K, \tau \mid \mathbf{y}] \propto \prod_{i=1}^{n} \text{normal}(y_i \mid \mu_i, \tau) \times \text{gamma}(K \mid .001, .001) \text{gamma}(\tau \mid .001, .001) \text{gamma}(r \mid .001, .001), \tag{4}$$

Note that equation 4 has meaning identical to:

$$g(r, K, x_i) = r - \frac{r x_i}{K}, \tag{5}$$

$$[r, K, \tau \mid \mathbf{y}] \propto \prod_{i=1}^{n} \text{normal}(y_i \mid g(r, K, x_i), \tau) \text{gamma}(K \mid .001, .001) \times \text{gamma}(\tau \mid .001, .001) \text{gamma}(r \mid .001, .001)$$

which is the same as

$$[r, K, \tau \mid \mathbf{y}] \propto \prod_{i=1}^{n} \text{normal}(y_i \mid r, K, \tau) \times \text{gamma}(K \mid .001, .001) \times \text{gamma}(\tau \mid .001, .001) \text{gamma}(r \mid .001, .001)$$

We will use the notation in equation 4 because it has a nice correspondence with the JAGS code, but you must remember that μ_i is simply a stand-in for $g(r, K, x_i)$.

where the priors are vague distributions for quantities that must, by definition, be positive. Note that we have used the precision (τ) as an argument to the normal distribution rather than the variance $\left(\tau = \frac{1}{\sigma^2}\right)$ to keep things consistent with the code below, a requirement of the BUGS language.[6] Now, we have full, abiding confidence that with a couple of hours worth of work, perhaps less, you could knock out a Gibbs sampler to estimate r, K, and τ.

[6] And also a source of suffering when you forget to use precision rather than variance.

However, we are all for doing things nimbly in 15 minutes that might otherwise take sweaty hours of hard labor, so, consider the code in algorithm 1, below. Note also that we have used "generic" vague priors, (i.e., flat gamma distributions), which may not be such a good idea. However, it suffices as a place to start without requiring a bunch of explanation.

Exercise 1: Factoring There is no $\mathbf{x}$ in the posterior distribution in equation 4. What are assuming if $\mathbf{x}$ is absent? Draw the Bayesian network, or DAG, for this model. Use the chain rule to fully factor the joint distribution into sensible parts, then simplify by assuming that r, K and τ are independent.

This code illustrates the purpose of JAGS (and other BUGS software): to translate the numerator of Bayes theorem (a.k.a., the joint distribution, e.g., equation 4) into a specification of an MCMC sampler. It is important for you to see the correspondence between the model (equation 4) and the code. JAGS parses this code, sets up proposal distributions and samplers in the MCMC algorithm, chooses values of tuning parameters if needed, and returns the MCMC chain for each parameter. These chains form the basis for approximating posterior distributions and associated statistics (e.g., means, medians, standard deviations, and quantiles). As we will soon learn, it easy to derive chains for other quantities of interest and their posterior distributions, for example, $K/2$ (by the way, what is $K/2$?), N as a function of time or dN/dt as a function of N. It is easy to construct comparisons between the growth parameters of two populations or among ten of them. It is straightforward to modify your model and code to incorporate errors in the observations of population growth rate or population size. If this seems as if it might be useful to you, continue reading, remembering that what you learn will apply to any quantity of interest: grams of N mineralized, height of trees, composition of landscapes, number of votes cast for a candidate.

JAGS is a compact language that includes a lean but useful set of scalar and vector functions for creating deterministic models as well as a full range of distributions for con-

```
## Logistic example for Primer
model{
  # priors
  K ~ dgamma(.001, .001)
  r ~ dgamma(.001, .001)
  tau ~ dgamma(.001, .001) # precision
  sigma <- 1/sqrt(tau) # calculate sd from precision
  # likelihood
  for (i in 1:n){
    mu[i] <- r - r/K * x[i]
    y[i] ~ dnorm(mu[i], tau)
  }
}
```

Algorithm 1: Linear regression example

structing stochastic models. The syntax closely resembles R[7], but there are differences and of course; JAGS is far more limited in what it does. Details of distributions used to fit models are found in section 9.2 of the JAGS manual (Plummer, 2015) and are summarized in the distribution cheat sheet. Rather than focus on these details, this tutorial presents a general introduction to JAGS models, how to call them from R, how to summarize their output, and how to check convergence. However, it necessary to read the the JAGS manual at some point as well, particularly the sections on functions, distributions, and differences between JAGS and WinBUGS.

4 Installing JAGS

Mac and Windows users, update your version of R to the most recent one. Go to the package installer under Packages and Data on the toolbar and check the box in the lower right corner for install dependencies. Install the `rjags` package (Plummer et al., 2016b) from a CRAN mirror of your choice. Check the version number of `rjags` – it should be 4-8.

The site for downloading the JAGS files occasionally gives security error messages that

[7]With a few irritating exceptions.

sound frightening. Try again a few times if you get one of these. My experience is that thye go away after a couple of tries.

4.1 Mac OS

Go to SourceForge and download `JAGS-4.3.0.dmg` to get the disk mounting image. Install as you would any other Mac software.

4.2 Windows

Go to SourceForge and download `JAGS-4.3.0.exe`. Occasionally, Windows users have problems loading `rjags` from R after everything has been installed properly. This problem usually occurs because they have more than one version of R resident on their computers (wisely, Mac OS will not allow that). So, if you can't seem to get `rjags` to run after a proper install, then uninstall all versions of R, reinstall the latest version, install the latest version of `rjags` and the version of JAGS that matches it.

4.3 Linux

There is a link to the path for binaries found here on SourceForge. If you want to compile from source code, there are detailed instructions on this blog post from Yu-Sung Su. Go to SourceForge and download `JAGS-4.3.0.tar.gz`. Our guess is that you will need to download the `rjags` package in R before installing `JAGS`. Here is a note on using the Ubuntu Software Center, compliments of Jean Fleming:

> "Elsie and I both use Ubuntu which is a specific linux distribution, it is one of the more commonly used distributions (it is user friendly!) so it is likely that many linux users in the future will be able to use this advice. If anyone does not have Ubuntu they may need to use the steps you described in the primer. I installed the rjags package following the directions in the primer. Ubuntu comes

with a Software Center where you can search for and download most open source software, so to download and install JAGS I just opened up Software Center, searched for JAGS, and installed it."

5 Running JAGS

5.1 The JAGS model

Study the relationship between the numerator of Bayes theorem (equation 4) and the code (algorithm 1). Although this model is a simple one, it has the same general structure as all Bayesian models in JAGS:

1. code for priors,
2. code for the deterministic model,
3. code for the likelihood(s).

The similarity between the code and equation 4 should be pretty obvious, but there are a few things to point out. Priors and likelihoods are specified using the $\sim$ notation that we have seen in class. For example, remember that $y_i \sim \text{normal}(\mu_i, \tau)$ is the same as $\text{normal}(y_i \mid \mu_i, \tau)$. So, it is easy to see the correspondence between the mathematical formulation of the model (i.e., the numerator of Bayes theorem, equation 4) and the code. In this example, we chose vague gamma priors for r, K and τ because they must be non-negative. We chose a normal likelihood because the values of y and μ are continuous and can take on positive or negative values.

5.2 Technical notes

5.2.1 The model statement

Your entire model must be enclosed in brackets, like this:

Exercise 2: Can you improve these priors? A recurring theme in this course will be to use priors that are informative whenever possible. The gamma priors in equation 4 include *the entire number line* > 0. Don't we know more about population biology than that? Let's, say for now that we are modeling the population dynamics of a large mammal. How might you go about making the priors on population parameters more informative?

```
model{

...

} # end of model
```

5.2.2 `for loops`

Notice that the for loop replaces the $\prod_{i=1}^{n}$in the likelihood. [8] Recall that when we specify an *individual* likelihood, we ask, what is the probability (or probability density) that we would obtain this data point conditional on the value of the parameter(s) of interest? The total likelihood is the product of the individual likelihoods. Recall in the Excel example for the light limitation of trees that you had an entire column of likelihoods adjacent to a column of deterministic predictions of our model. If you were to duplicate these "columns" in JAGS you would write:

```
mu[1] <- r - r/K * x[1]

y[1] ~ dnorm(mu[1], tau)

mu[2] <- r - r/K * x[2]

y[2] ~ dnorm(mu[2], tau)

mu[3] <- r - r/K * x[3]
```

[8] It is vital to understand that JAGS is *not* a procedural language like R. It is not a set of instructions to be executed sequentially. Instead it is a way to translate a mathematical statement into syntax that determine the MCMC sampler that JAGS composes "behind the scenes."

```
y[3] ~ dnorm(mu[3], tau)
...
mu[n] <- r - r/K * x[n]
y[n] ~ dnorm(mu[n], tau)
```

Well, presuming that you have something better to do with your time that to write out statements like this for every observation in your data set, you may substitute:

```
for (i in 1:n){
  mu[i] <- r - r/K * x[i]
  y[i] ~ dnorm(mu[i], tau)
}
```

for the line by line specification of the likelihood. Thus, the `for` loop specifies the elements in the product of the likelihoods. Note however, that the `for` structure in the JAGS language is subtly different from what you have learned in R. For example the following would be legal in R but not in the BUGS language:

```
# WRONG!!
for (i in 1:n){
  mu <- r - r/K * x[i]
  y[i] ~ dnorm(mu, tau)
}
```

If you write something like this in JAGS you will get a message that complains indignantly about multiple definitions of node `mu`. If you think about what the for loop is doing, you can see the reason for this complaint; the incorrect syntax translates to:

```
# WRONG!!
mu <- r - r/K * x[1]
```

```
y[1] ~ dnorm(mu, tau)
mu <- r - r/K * x[2]
y[2] ~ dnorm(mu, tau)
mu <- r - r/K * x[3]
y[3] ~ dnorm(mu, tau)
...
mu <- r - r/K * x[n]
y[n] ~ dnorm(mu, tau)
```

which is *nonsense* if you are specifying a likelihood because `mu` is used more than once in a likelihood for different values of y. This points out a fundamental difference between R and the JAGS language. In R, a `for loop` specifies how to repeat many operations in sequence. In JAGS a `for` construct is a way to specify a product likelihood or the distributions of priors for a vector. One more thing about the `for` construct. If you have two product symbols in the conditional distribution with different indices, that is $\prod_{i=1}^{n}\prod_{j=1}^{m}$, and two subscripts in the quantity of interest i.e. `quantity[i, j]` then this dual product is specified in JAGS using nested[9] for loops, i.e.,

```
for (i in 1:n){
  for (j in 1:m){
    quantity[i, j]
  } #end of j loop
} #end of i loop
```

The key is to look at the index of the quantity of interest and be sure that the `for` expressions span all values of the subscript. You can use the `length()` function as an alternative to giving an explicit argument for the number of iterations (e.g., `n` and `m` above),. For example we could use:

[9]There is also a way to do this with a single loop, called the index trick. We will learn this important approach soon. Stand ready.

```
for (1 in 1:length(x[])){
  mu[i] <- r - r/K * x[i]
  y[i] ~ dnorm(mu[i], tau)
}
```

Exercise 3: Using `for` loops. Write a code fragment to set vague normal priors for 5 regression coefficients – `dnorm(0, 10E-6)` – stored in the vector `b`.

5.2.3 Specifying priors

We specify priors in JAGS as:

```
randomVariable ~ distribution(parameter1, parameter2)
```

See the JAGS manual for available distributions. Note that in algorithm 1, the second argument to the normal density function is `tau`, which is the precision, defined as the reciprocal of the variance. This means that we must calculate `sigma` from `tau` if we want a posterior distribution on `sigma`. Be very careful about this – it is easy to forget that you must use the precision rather than the standard deviation as an argument to `dnorm` or `dlnorm`. Failing to do this is a source of immense suffering. (We know from experience.) For the lognormal, it is the precision on the log scale. We usually express priors on σ rather than τ using code like this:

```
# presuming 0-100 is far greater than the tails of the posterior of sigma
sigma ~ dunif(0, 100)
tau <- 1/sigma^2
```

This is what we normally do for two reasons. We can *think* about standard deviations but precisions are bewildering to us. This allows us to put reasonable constraints on σ, which we can then convert into τ. In addition, it can be very difficult to get τ to converge using the flat prior gamma($\tau \mid .001, .001$), especially if models are hierarchical. But we are getting ahead of ourselves.

5.2.4 The assignment operator, <- or =

It used to be that you were required to use `<-` as the assignment operator in JAGS. You can now use `<-` or `=` to accomplish the same thing. So, `a <- 4` and `a = 4` can be used to assign the value 4 to `a`. This corresponds with syntax in R.

5.2.5 Vector operations

We don't use any vector operations in algorithm 1, but JAGS supports a rich collection of operations on vectors. You have already seen the `length()` function – other examples include means, variances, standard deviations, quantiles, etc. See the JAGS manual. However, you cannot form vectors using syntax like R's concatenate function, `c()`. If you need a specific-valued vector in JAGS, assign elements directly, something like:

```
v[1] <- 7.8
v[2] <- 3.4
v[3] <- 2.3
```

would define the vector $v = (7.8, 3.4, 2.3)'$. Or, you can read them in as data from the R side of things, as we will soon learn.

5.2.6 Keeping variables out of trouble

Remember that all of the variables you are estimating will be sampled from a broad range of values, at least initially, and so it is often necessary to prevent them from taking on undefined values, for example logs of negatives, divide by 0, etc. You can usually use JAGS' `max()` and `min()` functions to do this. For example, to prevent logs from going negative, we often use something like `mu[i] <- log(max(.0001, expression))`.

5.3 Running JAGS from R

5.3.1 Stepping through a JAGS run

First, we transfer the code in the Logistic example (algorithm 2) into an R script. Note that we have changed the priors to reflect the answer to exercise 2 and the discussion above regarding precision versus variance. You may save this code in any directory that you like and may name it anything you like. Here we save the file as `LogisticJAGS.R`. In algorithm 4 we show you how to create `LogisticJAGS.R` directly using the `sink` function in the same R script you use to run the JAGS model itself. Either method will work so try both. [10]

```
## Logistic example for Primer
model{
  # priors
  K ~ dunif(0, 4000)
  r ~ dunif (0, 2)
  sigma ~ dunif(0, 2)
  tau <- 1/sigma^2
  # likelihood
  for(i in 1:n){
    mu[i] <- r - r/K * x[i]
    y[i] ~ dnorm(mu[i], tau)
  }
}
```

Algorithm 2: Refined linear regression example

We implement our model in R using algorithm 3. We will go through the R code step by step. We start by loading the package `ESS575` which has the data frame `Logistic`, which we then order by `PopulationSize`.[11] Next, we specify the initial conditions for the MCMC chain in the statement `inits`. This is exactly the same thing as you did when you wrote

[10] We prefer using a separate file for two reasons. Changes can be made and saved much more quickly if the JAGS code is in its own file. Error messages produce line numbers that mean something when the code is in a file. IWuse the `sink` approach only with very small models.

[11] It would be very instructive for you to omit this ordering after you have successfully completed exercises **??** and **??**. See what you get in your plots using unordered data. Then change all of the `lines` functions to `points`. Think this over and explain it. It is worth the effort. You will see it again, we promise.

MCMC code and assigned a guess to the first element in the chain.

```
rm(list = ls())
library(ESS575)
library(rjags)
Logistic  = Logistic[order(Logistic$PopulationSize),]

inits = list(
  list(K = 1500, r = .2, sigma = 1),
  list(K = 1000, r = .15, sigma = .1),
  list(K = 900, r = .3, sigma = .01))


data = list(
  n = nrow(Logistic),
  x = as.double(Logistic$PopulationSize),
  y = as.double(Logistic$GrowthRate))

n.adapt = 1000
n.update = 10000
n.iter = 10000

# Call to JAGS
set.seed(1)
jm = jags.model("LogisticJAGS.R", data = data, inits = inits,
n.chains = length(inits), n.adapt = )
update(jm, n.iter = n.update)
zm = coda.samples(jm, variable.names = c("K", "r", "sigma", "tau"),
n.iter = n.iter, n.thin = 1)
```

Algorithm 3: R code for running logistics JAGS script

Initial conditions must be specified as as "list of lists", as you can see in the code. If you create a single list, rather than a list of lists, i.e.:

```
inits = list(K = 1500, r = .5, tau = 2500) # WRONG
```

you will get an error message when you execute the `jags.model` statement and your code will not run. Second, this statement allows you to set up multiple chains,[12] which are needed for

[12]WE often start our work with a single chain. Once everything seems to be running, we add additional ones.

some tests of convergence and to calculate DIC (more about these tasks later). For example, if you want three chains, you would use:

```
inits = list(
  list(K = 1500, r = .2, sigma = 1),
  list(K = 1000, r = .15, sigma = .1),
  list(K = 900, r = .3, sigma = .01))
```

Now it is really easy to see why we need the "list of lists" format – there is one list for each chain; but remember, you require the same structure for a single chain, that is, a list of lists.

Which variables in your JAGS code require initialization? All unknown quantities that appear on the left hand side of the conditioning in the posterior distribution require initial values. Think about it this way. When you were writing your own MCMC algorithm, every chain required a value as the first element in the vector holding the chain. That is what you are doing when you specify initial conditions here. You can get away without explicitly specifying initial values – JAGS will choose them for you if you don't specify them – however, we strongly urge you to provide explicit initial values, particularly when your priors are vague. This habit also forces you to think about what you are estimating.

The next couple of statements,

```
data = list(
  n = nrow(Logistic),
  x = as.double(Logistic$PopulationSize),
  y = as.double(Logistic$GrowthRate))
```

specify the data that will be used by your JAGS program. Notice that you can assign data vectors on the R side to different names on the JAGS side. For example, the bit that reads `x = as.double(Logistic$PopulationSize)` says that the x vector in your JAGS program (algorithm 2) is composed of the column in your data frame called `PopulationSize`, and

the bit that reads `y = as.double(Logistic$GrowthRate)` creates a `y` vector required by the JAGS program from the column in your data frame called `GrowthRate`. You might want to know why we assigned the data as a double rather than as an integer. The answer is that the execution of JAGS is about 5 times faster on double precision than on integers.

It is important for you to understand that the left hand side of the = corresponds to variable name for the data in the JAGS program and the right hand side of the = is what they are called in R. Also, note that because `Logistic` is a data frame we used `as.integer()` and `as.double( )`[13] to be sure that JAGS received numbers instead of characters or factors, as can happen with data frames. In this case, it was unnecessary, but be aware you may need these. This can be particularly important if you have missing values in the data. The `n` is required in the JAGS program to index the `for` structure (algorithm 3) and it must be read as data in this statement.[14] By the way, you don't need to call this list "data" – it could be anything "apples", "bookshelves", "xy", etc.)

Now that you have a list of data and initial values for the MCMC chain you make calls to JAGS using the following statements:

```
library(rjags)
n.adapt = 1000
n.update = 10000
n.iter = 10000
# Call to JAGS
set.seed(1)
jm = jags.model("LogisticJAGS.R",data = data, inits = inits,
n.chains = length(inits), n.adapt = n.adapt)
update(jm, n.iter = n.update)
```

[13] This says the number is real and is stored with double precision, i.e., 64 bits in computer memory. This varies with the type of number being stored, but a good rule of thumb is that 16 decimal places can be kept in memory. This is usually sufficient for ecology!

[14] You could hard code the `for` index in the JAGS code, but this is bad practice. The best practice, which we use now, is to use something like `for (i in 1:length(y))`.

```
zm = coda.samples(jm, variable.names = c("K", "r", "sigma", "tau"),
n.iter = n.iter, n.thin = 1)
```

There is a quite a bit to learn here, so if your attention is fading, go get an espresso. First, we need to load the package `rjags`. Remember that MCMC required generating random numbers, so we use `set.seed(1)` to assure that our results are exactly reproducible.[15] We then specify 3 scalars, `n.adapt`, `n.update`, and `n.iter`. These tell JAGS the number of iterations in the chain for adaptation (`n.adapt`), burn-in (`n.udpate`) and the number to keep in the final chain (`n.iter`). The first one, `n.adapt`, may not be familiar – it is the number of iterations that JAGS will use to choose the sampler and to assure optimum mixing of the MCMC chain.[16] The second, `n.update`, is the number of iterations that will be discarded to allow the chain to converge before iterations are stored (aka, burn-in). The final one, `n.iter`, is the number of iterations that will be stored in the chain as samples from the posterior distribution – it forms the "rug".

The `jm = jags.model...` statement sets up the MCMC chain. Its first argument is the name of the file containing the JAGS code. Note that the file resided in the current working directory in this example. Otherwise, you would need to specify the full path name. The next two expressions specify where the data come from, where to get the initial values, and how many chains to create (i.e., the length of the list inits). Finally, it specifies the burn-in how many samples to throw away before beginning to save values in the chain. Thus, in this case, we will throw away the first 10,000 values.

The second statement (`zm = coda.samples...`) creates the chains and stores them as an MCMC list (more about that soon). The first argument (`jm`) is the name of the jags model you created with the `jags.model` function. The second argument (`variable.names`) tells JAGS which variables to "monitor". These are the variables for which you want posterior distributions.

[15] With enough iterations, this is not really necessary, but it is nice for teaching.
[16] Remember the tuning parameter in Metropolis Hastings?

Finally, `n.iter = n.iter` says we want 10,000 iterations in each chain and `thin` specifies how many of these to keep. For example, if thin = 10, we would store every 10th element. Sometimes setting `thin > 1` is a good idea to reduce the size of the data files that you will analyze, but that is not the only reason you should thin the chain (Link and Eaton, 2012).

```
{ # Extra bracket needed only for R markdown files - see answers
sink("LogisticJAGS.R") # This is the file name for the jags code
cat("
model{
  # priors
  K ~ dunif(0, 4000)
  r ~ dunif (0, 2)
  sigma ~ dunif(0, 2)
  tau <- 1 / sigma^2
  # likelihood
  for(i in 1:n){
    mu[i] <- r - r / K * x[i]
    y[i] ~ dnorm(mu[i], tau)
  }
}
",fill = TRUE)
sink()
} # Extra bracket needed only for R markdown files - see answers
```

Algorithm 4: Example of code for inserting JAGS code into an R script. This should be placed above the R code in algorithm 3. You must remember to execute the code in between the `sink` commands every time you make changes in the model.

Exercise 4: Coding the model. Write R code (algorithm 3) to run the JAGS model (algorithm 2) and estimate the parameters, r, K σ, and τ. We suggest you insert the JAGS model into this R script using the `sink` command as shown in algorithm 4 because this model is small. You will find this a convenient way to keep all your code in the same R script. For larger models, you will be happier using a separate file for the JAGS code.

5.3.2 Parallelizing your JAGS run (optional)

This section is completely optional and you can safely ignore it for the entire course. However, at some point in the future you will likely want to parallelize a JAGS run, either on your own computer or on a high performance cluster or cloud computing service such as AWS, and this can be confusing. We suggest either revisiting this section when you are ready to do that or if you have completed the rest of the primer and are looking to tackle something a bit more challenging.

Running your model using the code in section 5.3.1 will cause JAGS to run *sequentially*, meaning JAGS will first iterate chain 1, followed by chain 2, all the way through chain n. There is nothing wrong with this approach, but you can significantly speed up time to convergence by iterating all chains *simultaneously*, i.e. running the chains in parallel. To do this, we will instruct R to have JAGS run one chain per thread,[17] while leaving at least one core free to handle all other computer operations. To get started, load the `parallel` package in R and execute the command `detectCores()` to determine how many cores (i.e., threads) your computer has. If you have a 2 core machine that is hyper-threaded, there are 4 threads, which is what is reported by `detectCores()`. Each thread will run an R instance called a worker, where each worker is given a unique identifier called a PID (you can think of this as each worker's name, something we will come back to at the end of this section). We will refer to the group of R workers as a cluster. Assuming you have at least 4 threads, the statement:

```
cl <- makeCluster(3)
```

will create a cluster of three R workers, one per thread.

Instead of using fixed initial values as we did in 5.3.1, we will assign initial values randomly to each chain using the function:

[17] Here is short description explaining the difference between CPUs, multiple cores, and hyper-threading.

```
initsP = function() {list(K = runif(1, 10, 4000), r = runif(1, .01, 2),
  sigma = runif(1, 0, 2),  .RNG.name = "base::Mersenne-Twister",
  .RNG.seed = runif(1, 1, 2000))
```

There are a few things worth noting in this function, so study it carefully. The usual `inits` list has been replaced with a a function that uses `runif()` to generate random initial values. You need to specify the parameters to be initialized here and some reasonable ranges for their starting values (this is done with the parameters governing the `runif()` functions themselves). Every time `initP()` is invoked, it will produce a new set of initial values.

Even if we make sure that the initial values are different for each chain running in parallel, we need to consider how JAGS handles randomization[18]. JAGS sets its own seed based on the time (based on the time the model was implemented) and random number generator, or RNG (JAGS defaults to Mersenne-Twister just as R does), independently from the RNG and seed specified for its worker. If each worker implements the model at the same time then each chain will have the same `.Random.seed` vector when performing MCMC. If you have different initial values (or at least one different initial initial value for a free parameter in your model) then the chains will not be identical, despite the fact that all chains will have used identical random values. Is that enough randomization between chains? We are not sure, but a safe approach is to specify the seed and RNG (this is required by JAGS when setting the seed, but you can use the same RNG) for each chain, as well as the initial values. Fortunately, this is easy. We do this by adding `.RNG.seed` and `.RNG.name` arguments to the

[18] In R random numbers are not truly random. Instead, they are generated using a seed value (set.seed) and a pseudo-random number generator or RNG (the default in R is Mersenne-Twister, but others exist). This RNG is a function that uses the seed to generate a deterministic sequence of numbers that approximates a sequence of truly random numbers, which is stored in a vector called `.Random.seed`. The first value of `.Random.seed` identifies the RNG. The length and structure of the remaining portion of the vector depends on the RNG itself. For Mersenne-Twister, the total vector is 625 elements long, the second of which identifies how many of the 623 values have been "used up" in creating output from functions that call `.Random.seed` (runif, rnorm, sample, etc). Once the end of the vector is reached, a fresh set of 623 values replace the first set and the position counter is set to 1. Oddly the initial position counter starts on the 623rd value of the RNG output when first created with set.seed. Invoke set.seed again with a new seed value and you will get an entirely new `.Random.seed` starting at position 624 (the end of the first set). Invoke `set.seed` with the same seed value and you will recreate `.Random.seed` from the beginning.

list of initial values for each chain. Here we assign each chain the Mersenne-Twister RNG and select the seed at random in the same fashion as we do the initial values themselves.

We then provide each worker with the R objects `data`, `initsP`, `n.adapt`, `n.update`, and `n.iter` using the `clusterExport` function:

```
parallel::clusterExport(cl, c("data", "initsP", "n.adapt", "n.update",
  "n.iter"))
```

Wee embed our previous R code for running the JAGS model inside the cluster `cl`. This allows the cluster to direct an identical set of R commands to each worker, collect output from each worker, and combine this output into a list where each element corresponds to the output from a single worker. You can name the MCMC object outputted from each core and the list of all MCMC objects produced by the entire cluster whatever you like. Here we name them `zm` and `out`, respectively:

```
out <- clusterEvalQ(cl, {
  library(rjags)
  jm = jags.model("LogisticJAGS.R", data = data, n.chains = 1,
  n.adapt = n.adapt, inits = initsP())
  update(jm, n.iter = n.update)
  zm = coda.samples(jm, variable.names = c("K", "r", "sigma", "tau"),
  n.iter = n.iter, thin = 1)
  return(as.mcmc(zm))
})
stopCluster(cl)
zmP = mcmc.list(out)
```

Notice how we must now load the `rjags` library for each worker, meaning `library(rjags)` needs to be inside the `clusterEvalQ` function. Each worker then runs a single chain and

we use the `as.mcmc` function to save the output from the `coda.samples` statement as an MCMC object. What is the object `out` in this case? Most confusingly, it is a list of MCMC objects but not yet an MCMC list. We convert the garden-variety list `out` to the MCMC list `zmP` with the command `zmP = mcmc.list(out)`. We also stop the cluster `cl` to free up our computer's resources with the command `stopCluster(cl)`. It is worth noting that the MCMC list `zmP` you just created will be equivalent to the MCMC list `zm` you made in section 5.3.1. The structure of these objects and how to manipulate them is the subject of the next section.

```
library(parallel)

cl <- makeCluster(3)

initsP = function() {list(K = runif(1, 10, 4000), r = runif(1, .01, 2),
  sigma = runif(1, 0, 2),  .RNG.name = "base::Mersenne-Twister",
  .RNG.seed = runif(1, 1, 2000))

parallel::clusterExport(cl, c("data", "initsP", "n.adapt", "n.update",
  "n.iter"))

out <- clusterEvalQ(cl, {
  library(rjags)
  jm = jags.model("LogisticJAGS.R", data = data, n.chains = 1,
  n.adapt = n.adapt, inits = initsP())
  update(jm, n.iter = n.update)
  zm = coda.samples(jm, variable.names = c("K", "r", "sigma", "tau"),
  n.iter = n.iter, thin = 1)
  return(as.mcmc(zm))
})
stopCluster(cl)
zmP = mcmc.list(out)
```

Algorithm 5: R code for running logistics JAGS script in parallel with random initial values

There is an important caveat concerning the use of random initial values for JAGS running either in parallel or sequentially. Although choosing initial values randomly is widely done by accomplished Bayesian analysts, a problem can result. The Gelman diagnostic,

discussed in 7.2, requires that some of the initial values are diffuse relative to the posterior mean, i.e. there should be some that are well above it and some well below. You cannot be sure this is the case when initial values are random. Our general advice is to use random initial values when fitting models in JAGS. When you have you model running, converging, and producing sensible output, choose fixed but diffuse initial values *relative to each parameter's posterior* prior to publishing.

This creates a bit of a roadlock when running JAGS in parallel, since it is not clear how to have R pass different initial values to each R workers. To do this, we need to learn the names of our faithful workers. Just who are these folks, anyway? The function `capture.output()` will tell as the PID (or name) of any worker in a cluster. For example, `capture.output(cl[[1]])` tells us the PID of our first worker in the cluster `cl`:

```
"node of a socket cluster on host 'localhost' with pid 24752"
```

This worker's PID is annoyingly embedded in a text string, but we can write a short user function using the `word()` function from the `stringer` package to access it, along with all the other PIDs in the cluster:

```
myWorkers <- NA
for(i in 1:3) myWorkers[i] <- word(capture.output(cl[[i]]),-1)
```

The `myWorkers` object is simply a vector of the names of all your recently created R workers, each patiently standing by to manage JAGS for a single MCMC chain. So, say hello to:

```
> myWorkers
[1] "24752" "24762" "24772"
```

We must not forget to provide each worker with this list of all the worker names:

```
parallel::clusterExport(cl, c("myWorkers", "data", "initsP", "n.adapt",
  "n.update", "n.iter"))
```

We return to using fixed inits, but this time we also specify the RNG and seed for each JAGS chain as done in algorithm 5. Note that `initsP` is not a function here:

```
initsP = list(
  list(K = 1500, r = .2, sigma = 1, RNG.seed = 1,
    .RNG.name = "base::Mersenne-Twister"),
  list(K = 1000, r = .15, sigma = .1, RNG.seed = 23,
    .RNG.name = "base::Mersenne-Twister"),
  list(K = 900, r = .3, sigma = .01, .RNG.seed = 255,
    .RNG.name = "base::Mersenne-Twister"))
```

So how do we tell the first worker to use the first set of initial values from `initsP`, the second worker the second set, and so on and so forth? We do this by asking each worker its own name (PID) with the `Sys.getpid()` function before assigning it a set of initial values from `initsP` based on its corresponding position in the `myWorkers` vector:

```
out <- clusterEvalQ(cl, {
  library(rjags)
  jm = jags.model("LogisticJAGS.R", data = data, n.chains = 1,
  n.adapt = n.adapt, inits = initsP[[which(myWorkers==Sys.getpid())]])
  update(jm, n.iter = n.update)
  zm = coda.samples(jm, variable.names = c("K", "r", "sigma", "tau"),
  n.iter = n.iter, thin = 1)
  return(as.mcmc(zm))
})
```

This is equivalent to asking all the workers line up in a row alphabetically by name before handing them individual tasks depending on their position in line.

```
library(parallel)
library(stringr)

cl <- makeCluster(3)
myWorkers <- NA
for(i in 1:3) myWorkers[i] <- word(capture.output(cl[[i]]),-1)

initsP = list(
  list(K = 1500, r = .2, sigma = 1, RNG.seed = 1,
    .RNG.name = "base::Mersenne-Twister"),
  list(K = 1000, r = .15, sigma = .1, RNG.seed = 23,
    .RNG.name = "base::Mersenne-Twister"),
  list(K = 900, r = .3, sigma = .01, .RNG.seed = 255,
    .RNG.name = "base::Mersenne-Twister"))

parallel::clusterExport(cl, c("myWorkers", "data", "initsP",
  "n.adapt", "n.update", "n.iter"))

out <- clusterEvalQ(cl, {
  library(rjags)
  jm = jags.model("LogisticJAGS.R", data = data, n.chains = 1,
  n.adapt = n.adapt, inits = initsP[[which(myWorkers==Sys.getpid())]])
  update(jm, n.iter = n.update)
  zm = coda.samples(jm, variable.names = c("K", "r", "sigma", "tau"),
  n.iter = n.iter, thin = 1)
  return(as.mcmc(zm))
})
stopCluster(cl)
zmP = mcmc.list(out)
```

Algorithm 6: R code for running logistics JAGS script in parallel with fixed initial values

6 Output from JAGS

6.1 coda objects

The `coda.samples` function in `rjags` package produces output in the form of an mcmc.list object, sometimes referred to as a coda object (in reference to the `coda` package (Plummer et al., 2016a), recalling the final movements of symphonies).

Exercise 5: Coding the logistic regression to run in parallel. Append R code (algorithm 5) to the script you made in exercise 4 to run the JAGS model (algorithm 2) in parallel and estimate the parameters, r, K σ, and τ. Use the `proc.time` function in R to compare the time required for the sequential and parallel JAGS run. If your computer has 3 threads, try running only 2 chains in parallel when doing this exercise. If you have fewer than 3 threads, work with a classmate that has at least 3 threads. What happens when you set the seed inside of each R worker prior to running the JAGS code? Repeat this exercise with fixed initial values using algorithm 6. What happens when you use the same seed for each chain?

6.1.1 The structure of coda objects

The `zm` object produced by the statement:

```
zm = coda.samples(jm, variable.names = c("K", "r", "sigma", "tau"),
n.iter = n.iter, thin = 1)
```

is a coda object (more precisely, an mcmc.list object). But what is a coda object? For the first chain, it looks something like this[19]:

```
[[1]]
Markov Chain Monte Carlo (MCMC) output:
Start = 15001
End = 15007
Thinning interval = 1
            K         r      sigma       tau
[1,] 1177.867 0.1871039 0.02207365 2052.3520
[2,] 1317.527 0.1984132 0.03435912  847.0633
[3,] 1205.740 0.2029698 0.03561197  788.5114
```

[19] Don't worry if your output doesn't match this exactly. The example here was run with a different set of priors and initial values.

```
[4,] 1219.347 0.2025454 0.02059334 2358.0146
[5,] 1334.068 0.2028072 0.02342167 1822.9062
[6,] 1297.067 0.1916264 0.01886448 2810.0267
[7,] 1319.495 0.2063581 0.01723530 3366.3722
...
as many rows as you have thinned iterations
```

So, the output of coda is a list of matrices where each matrix contains the output of a single chain for all parameters being estimated. Parameter values are stored in the columns of the matrix; values for one iteration of the chain are stored in each row. If we had 2 chains, 5 iterations each, the coda object would look like:

```
[[1]]
Markov Chain Monte Carlo (MCMC) output:
Start = 10001
End = 10005
Thinning interval = 1
           K          r      sigma
[1,] 1070.013 0.2126878 0.02652204
[2,] 1085.438 0.2279789 0.02488036
[3,] 1170.086 0.2259743 0.02331958
[4,] 1094.564 0.2228788 0.02137309
[5,] 1053.495 0.2368199 0.03209893
[[2]]
Markov Chain Monte Carlo (MCMC) output:
Start = 10001
End = 10005
Thinning interval = 1
            K         r      sigma
[1,] 1137.501 0.2657460 0.04093364
```

```
[2,] 1257.340 0.1332901 0.04397191
[3,] 1073.023 0.2043738 0.03355776
[4,] 1159.732 0.2339060 0.02857740
[5,] 1368.568 0.2021042 0.05954259
attr(,"class")
[1] "mcmc.list"
```

It is important that you be able to manipulate the MCMC output stored in coda objects. Why would you want to do that? There are two reasons. By dissecting the coda object, you can see the MCMC process that we worked so hard to understand earlier. A more useful reason is describe in Box 1.

6.1.2 Summarizing coda objects with the `MCMCvis` package

Manipulating and summarizing coda objects manually can be tedious work prone to coding errors. Fortunately there are existing tools to help out with these tasks. The `MCMCvis` package (Youngflesh, 2018)[20] provides a series of functions to summarize, manipulate, and visualize model output. You should get the package `MCMCvis` and load the library.

This is the first year we have used `MCMCvis` in teaching, although Chris has used it in his work for some time now. We are adopting it for two reasons. First, it will work with coda objects produced by any of the flavors of MCMC software, JAGS, OpenBUGS, WinBUGS and Stan, or for matrices produce from your own MCMC samplers. It allows you to run chains in parallel and make useful inference on the parallel output, which must be a coda object. A related advantage is the you will need to produce only a single inferential object while the past, we produced two (coda objects and jags objects), almost doubling the computation time.

[20] Casey Youngflesh took our Bayesian short course at SESYNC in 2016. He has obviously made lots of progress since then!

Exercise 6: Understanding coda objects.

1. Convert the coda object `zm`, into a data frame using `df = as.data.frame(rbind(zm[[1]], zm[[2]], zm[[3]]))` Note the double brackets, which effectively unlist each element of zm, allowing them to be combined. Another way to do this is `do.call(rbind,zm)`.
2. Look at the first six rows of the data frame.
3. Find the maximum value of `sigma`.
4. Find the mean of `r` for the first 1000 iterations.
5. Find the mean of `r` after the first 1000 iterations.
6. Make two publication quality plots of the marginal posterior density of `K`, one as a smooth curve and the other as a histogram.
7. Compute the probability that `K` < 1600. (Hint– what type of probability distribution would you use for this computation? Investigate the the dramatically useful R function `ecdf()`).
8. Compute the probability that $1000 <$ `K` < 1300.
9. Compute the .025 and .975 quantiles of `K`. Hint–use the R `quantile()` function. This is an equal-tailed Bayesian credible interval on K.

We cover the essential elements here, but you should must work through the full `MCMCvis` tutorial to get maximum benefit of these powerful functions. We suggest that you have the tutorial open in the help window of R studio as you work through the exercises below.[21]

[21] Go to R help, packages, MCMCvis, User guides, package vignettes and other documentation, MCMCvis::MCMCvis.

Box 1: Using R to calculate derived quantities from MCMC objects

One of the most powerful benefits of Bayesian analysis using MCMC is the ability to make inference on *derived* quantities using what is called the equivariance principle (Hobbs and Hooten (2015) page 194), which simply says that any function of a random variable becomes a random variable with its own marginal posterior distribution. This principle allows you to write simple functions in your JAGS code and make inference on quantities that are calculated from the random variables included in the model (e.g., r, K, σ) as you will soon see in the exercises associated with this lab.

However, what can you do if the function you need to apply is too complex to write out in JAGS? A good example in Tom's work is finding the dominant eigenvalue and eigenvector of projection matrices. What do you do if you need to use a function like R's `eigen()`? This is where knowing how to "get under the hood" with MCMC output stored in coda objects is critical. You simply take output from each iteration, apply the function, and store the result, enabling you to make inference on the stored results in the same way you make inferences on the output from the MCMC itself by creating a "derived" chain. This is covered nicely in Hobbs and Hooten (2015) section 8.3.

`MCMCvis` includes five functions that we will use a lot:

- `MCMCsummary` - summarize MCMC output for particular parameters of interest
- `MCMCpstr` - summarize MCMC output and extract posterior chains for particular parameters of interest while preserving parameter structure
- `MCMCtrace` - create trace and density plots of MCMC chains for particular parameters of interest
- `MCMCchains` - easily extract posterior chains from MCMC output for particular parameters of interest

- `MCMCplot` - create caterpillar plots from MCMC output for particular parameters of interest

6.1.3 Tabular summaries with `MCMCsummary`

You can obtain a summary of statistics from MCMC chains contained in a coda object by loading the `MCMCvis` package (using `library(MCMCvis)`) and then running `MCMCsummary(co)` (where `co` is a coda object produced from your model run). All of the variables in the `variable.names = c( )` argument to the `coda.samples` function will be summarized. For the example here, `MCMCsummary(zm)` produces:

```
         mean     sd    2.5%     50%   97.5% Rhat
K     1238.19  62.97 1129.05 1233.29 1377.56    1
r        0.20   0.01    0.18    0.20    0.22    1
sigma    0.03   0.00    0.02    0.03    0.04    1
tau   1259.39 258.98  804.20 1242.71 1809.77    1
```

A few things deserve note. First, it is imperative that you understand that the `sd` in this table is the standard deviation of the marginal distribution of the parameter, analogous to the standard error of the mean. The 2.5%, 50%, and 95.5% quantiles are given for each parameter, as is the Rhat value (used to assess convergence, which we will discuss later). The table above has the properties of a matrix. You can output the cells of these tables using syntax as follows. To get the mean and standard deviation of r,

```
> MCMCsummary(zm)[2, 1:2]
mean   sd
0.20 0.01
```

To get the upper and lower 95% quantiles on K from the tabular summary:

```
> MCMCsummary(zm)[1, c(1, 5)]
   mean   97.5%
1238.19 1377.56
```

We will learn easier, less error-prone ways to do this below.

The parameter name of interest can be passed directly to the `params` argument of the `MCMCsummary` function to more easily summarize output. For example, to see statistics for the marginal posterior distribution of σ,

```
> MCMCsummary(zm, params = 'sigma')
      mean sd 2.5%  50% 97.5% Rhat
sigma 0.03  0 0.02 0.03  0.04    1
```

There are some particularly useful options for this function. You can control the number of significant digits (`signif = `) or the number of decimal places (`round =`). You can also specify the convergence diagnostics `Rhat` and `n.eff`. See the MCMCvis tutorial and sections 7.2 and 7.5 in this document or more explanation.

Exercise 7: Using `MCMCsummary`

1. Summarize the coda output from the logistic model with 4 significant digits. Include Rhat and effective sample size diagnostics (more about these soon).

2. Summarize the coda output for `r` alone.

6.1.4 Extracting portions of coda object with `MCMCchains`

`MCMCchains` can be used to extract chains from coda object. These can then be summarized using the usual R functions. For example

```
> r.ex = MCMCchains(zm, params="r")
> dim(r.ex)
[1] 30000     1
> head(r.ex)
             r
[1,] 0.1840311
[2,] 0.1993270
[3,] 0.1914906
[4,] 0.1930457
[5,] 0.1862841
[6,] 0.1897411
> mean(r.ex)
[1] 0.2007835
> ecdf(r.ex)(.18)
[1] 0.01743333
> quantile(r.ex,c(.025,.975))
     2.5%     97.5%
0.1816953 0.2201085
```

6.1.5 Extracting portions of coda objects while preserving their structure using `MCMCpstr`

The coda object is strictly tabular – it is a list of matrices with the number of elements of the list equal to the number of chains. So, for example three chains will produce a list of three matrices. The columns of each matrix include the variables specified in the `variable.names = ...` argument of `coda.samples`. The rows hold the output of individual iterations. This is fine when the parameters you are estimating are entirely scalar, but sometimes you want

Exercise 8: Using `MCMCchains`

1. Extract the chains for `r` and `sigma` as a data frame using `MCMCchains` and compute their .025 and .975 quantiles from the extracted object. Display three significant digits.

2. Make a publication quality histogram for the chain for `sigma`. Indicate the .025 and .975 Bayesian equal-tailed credible interval value with vertical red lines.

3. Overlay the .95 highest posterior density interval with vertical lines in blue. This is a "learn on your own" problem intended to allow you to rehearse the most important goal of this class: being sufficiently confident to figure things out. Load the package `HDinterval`, enter `?HDInterval` at the console and follow your nose. Also see Hobbs and Hooten Figure 8.2.1.

posterior distributions for all of the elements of vectors or matrices and in this case, the coda object can be quite cumbersome because the elements of vectors and matrices loose their structure, being shoe-horned into rows an a matrix.

We have been working with scalars, `r, K,` and `sigma` up to now. We will often seek inference on vectors, particularly when we want to make predictions. These predictions can be made at the in-sample $x's$ of for out-of-sample $x's$ that we specify. For example, take a look at your JAGS model. The quantity `mu[i]` is a the prediction of our deterministic model for each of the `x[i]`. The quantity `mu` is a vector, a *derived quantity*. We call it a derived quantity because it is a function of random variables in the model. The equivariance property of MCMC means that any quantity that is a function of random variables becomes a random variable with its own marginal posterior distribution.

Presume you would like to get posterior distributions on the *predictions* of your regression model for each of the `x` values. Note in the JAGS code that we make this prediction as `mu[i]`

`<- r - r / K * x[i]`. Making predictions requires no more than adding `mu` to the list of random variables being monitored by changing your `coda.samples` statement to read:

```
zm = coda.samples(jm, variable.names = c("K", "r", "sigma", "mu"),
  n.iter = n.iter, thin = 1)
```

Instead of using `MCMCsummary` to summarize your data, use `MCMCpstr`, to preserve the structure of the `mu` parameter a vector of length equal to the number of data points. `MCMCsummary` returns the mean of each parameter by default, but any function can be specified using the `func` argument. The format is a list, with each parameter listed as a separate element in that list. Any parameter of interest can be specified, as with the other functions in the `MCMCvis` package. Use the following general syntax to summarize the coda object while preserving the parameter structure:

```
MCMCpstr(co, params = 'PARAMS', func = FUN)
```

where `co` is a coda object, `PARAMS` are the parameters of interest (all parameters by default), and `FUN` is the function of interest (mean by default). Only one function can be applied at a given time. The most useful of these are illustrated here, but there are an infinite number of possibilities, including functions that you write.

```
> BCI <- MCMCpstr(zm, params = "mu",  func = function(x)
   quantile(x, probs = c(.025,.5,.975)))
> BCI
$mu
              2.5%        50%      97.5%
mu[1]  0.172880664 0.19043630 0.20734240
mu[2]  0.171524527 0.18881415 0.20546368
mu[3]  0.165875964 0.18213155 0.19775789
.
```

```
.
.
mu[48] 0.018303545 0.03273251 0.04706730
mu[49] 0.016248511 0.03093309 0.04551707
mu[50] 0.006896648 0.02279023 0.03858062
```

Be sure you understand that `MCMCpstr()` returns a *list*, not a vector or matrix. Each element of the list is named for the parameters listed in the `params = option`. Obtaining the matrices computed above requires `BCI$mu`. Here is an example for more than one variable:

```
library(HDInterval)
> HPDI <-  MCMCpstr(zm, params = c("r", "K"), func = function(x)
  hdi(x, .95))
> HPDI
$r
      lower     upper
r 0.1827305 0.2202985
$K
     lower    upper
K 1117.529 1357.458
```

6.1.6 Visualizing model output

Posterior densities (representing the model estimate for each parameter and their uncertainty) can be easily visualized with caterpillar plots produced with the `MCMCplot` function. Once again, the coda object and parameters are fed directly to the function:

```
MCMCplot(co, params = c('alpha, 'beta''))
```

Exercise 9: Plotting model predictions using the MCMCpstr formatting.

1. Plot the observations of per-capita growth rate as a function of observed population size. Be sure that the population size data are sorted from lowest to highest as described above
2. Overlay the median of the model predictions as a solid black line.
3. Overlay the 95% credible intervals as dashed lines in red.
4. Overlay the 95% highest posterior density intervals as dashed lines in blue.
5. What do you note about these two intervals? Will this always be the case? Why or why not?
6. What do the dashed lines represent? What inferential statement can we make about `mu` relative these lines?

Exercise 10: Creating caterpillar plots with `MCMCplot`. Use the `MCMCplot` function to create caterpillar plots to visualize the posterior densities for `r`, and `sigma` using the coda object `zm` from earlier. Then use `?MCMCplot` to explore different plotting options. There are **lots** of these options, including ways to make the plots publication quality, show overlap with zero, etc.

7 Checking convergence

Recall from lecture that MCMC output will provide a reliable approximation of the marginal posterior distribution of unobserved quantities only after convergence, which means that adding more iterations will not appreciably change the shape of marginal posterior distributions. Both `MCMCvis` and the `coda` packages contains tools for evaluating and manipulating MCMC chains produced in coda objects. We urge you to look at the package documentation in R Help, because we will use only a few of the tools it offers. There are several ways to

check convergence, but we will use four here: 1) visual inspection of density and trace plots 2) Gelman and Rubin diagnostics, 3) Heidelberger and Welch diagnostics, and 4) Raftery diagnostics. Also see Hobbs and Hooten (2015) section 7.

7.1 Trace and density plots

Trace plots (showing parameter estimates at each iteration in the chain over time) as well as posterior density plots can be plotted with `MCMCtrace`. The `type` argument can be used to specify which types of plots should be generated (trace, density, or both). As with other function in the `MCMCvis` package, parameters of interest can be specified. Trace plots indicate convergence if they are horizontal bands with all chains mixed within the band.

`MCMCtrace` is an exceptionally useful function allowing you to overlay priors and marginal posteriors (computing their %overlap) as well as overlaying marginal posteriors on values of parameters used to generate simulated data. Look at the `MCMCvis` vignette to learn more about these useful capabilities.

7.2 Gelman and Rubin diagnostics (Rhat)

The standard method for assuring convergence is the Gelman and Rubin diagnostic (Gelman and Rubin, 1992), which "determines when the chains have forgotten their initial values, and the output from all chains is indistinguishable" (R Development Core Team, 2016). This is also commonly referred to as Rhat. It requires at least 2 chains to work. For a complete treatment of how this works, see Hobbs and Hooten (2015) section 7.3.4.2. We can be sure of convergence if all values for point estimates and 97.5% quantiles approach 1. More iterations should be run if the 95% quantile > 1.05. `MCMCsummary` displays Rhat values by default for each parameter. It can be calculated with the following:

```
MCMCsummary(codaObject)
```

Reliable inference from the Gelman and Rubin diagnostic requires initial values that are diffuse relative to marginal posterior distribution, that is, that are in the extreme tails of the distribution. How do you set these values? Run a single chain until things look right. Then choose initial values that are on well in the tails of the distribution on either side of the mean.

The effective sample size `n.eff` is a particularly useful quantity, not reported by most MCMC software. Think about it this way. Imagine you have 10,000 iterations. You might think that the sample size for computing statistics is, well, 10,000 and, yes, in fact, when you compute a mean it is based on n = 10,000. However, the information about that mean is less than 10,000 when chains are autocorrelated, that is when a value at iteration k is more similar to the value a $k+1$ than to the value at $k+10$. Therefore, the *effective* sample size can be far less than 10,000. This is what `n.eff` is telling you. You may have a converged chain but will want more iterations to simply increase the accuracy of numeric approximations of statistics.

7.3 Heidelberger and Welch diagnostics

The Heidelberger and Welch diagnostic (Heidelberger and Welch, 1983) works for a single chain, which can be useful during early stages of model development before you have initialized multiple chains. The diagnostic tests for stationary in the distribution and also tests if the mean of the distribution is accurately estimated. The function `heidel.diag` in the `coda` package can be used to calculated this metric. For details do `?heidel.diag` and read the part on Details. We can be confident of convergence if out all chains and all parameters pass the test for stationarity and half width mean. We can be sure that the chain converged from the first iteration (i.e, burn in was sufficiently long) if the start iteration = 1. If it is greater than 1, the burn in should be longer, or `1:start.iteration` should be discarded from the chain. The synatx is:

```
heidel.diag(codaObject)
```

7.4 Raftery diagnostic

The Raftery diagnostic Raftery and Lewis (1995) is useful for planning how many iterations to run for each chain. It is used early in the analysis with a relatively short chain, say 10,000 iterations. It returns an estimate of the number of iterations required for convergence for each of the parameters being estimated. The `raftery.diag` function in the `coda` package can be used in this case. The syntax is:

```
raftery.diag(codaObject)
```

Exercise 11: Assessing convergence. Rerun the logistic model with `n.adapt = 100`. Then do the following:

1. Keep the next 500 iterations. Assess convergence visually with `MCMCtrace` and with the Gelman-Rubin, Heidelberger and Welch, and Raftery diagnostics.
2. Update another 500 iterations and then keep 500 more iterations. Repeat your assessment of convergence.
3. Repeat steps 1 and 2 until you feel you have reached convergence.
4. Change the adapt phase to zero and repeat steps 1 – 4. What happens?

7.5 Effective sample size (n.eff)

MCMC chains will almost always have some degree of autocorrelation, which is to say samples that are close to each other in the chain will be more similar than samples that are far apart. One hundred samples that are highly correlated may only contain as much information about the marginal posterior as 20 samples that are truly independent. This lead to the concept of effective sample size, abbreviated by MCMCsummary as `n.eff`. The effective sample size

is an estimate of the number of samples drawn that are independent. It is

$$\text{ESS} = \frac{n}{1 + 2\sum_1^\infty \rho(k)}$$

where $\rho(k)$ is the correlation between samples at lag k.

8 Monitoring deviance and calculating DIC

It is often a good idea to report the deviance of a model, defined as $-2\log[P(y \mid \theta)]$. To obtain the deviance of a JAGS model you need to do two things. First, you need to add the statement:

```
load.module("dic")
```

above your `coda.samples` statement. In the list of variables to be monitored, you add "deviance" i.e.,

```
zm = coda.samples(jm,variable.names=c("K", "r", "sigma", "deviance"),
n.iter = 25000, thin = 1)
```

Later in the course we will learn about the Bayesian model selection statistic, the deviance information criterion (DIC). DIC samples values are generated using syntax like this:

```
dic.object.name = dic.samples(jags.model, n.iter, type = "pD")
```

So, to use your regression example, you would write something like:

```
dic.j = dic.samples(jm, n.iter = 2500, type = "pD")
```

If you enter `dic.j` at the console (or run it as a line of code in your script) R will respond with something like:

```
Mean deviance:  -46.54
penalty 1.852
Penalized deviance:  -44.69
```

9 Differences between JAGS and Win(Open)BUGS

The JAGS implementation of the BUGS language closely resembles the implementation in WinBUGS and OpenBUGS, but there are some important structural differences that are described in Chapter 8 of the JAGS manual (Plummer, 2015). There are also some functions (for example, matrix multiplication and the ^ symbol for exponentiation) that are available in JAGS has but that are not found in the other programs.

10 Troubleshooting

Some common error messages and their interpretation are found in the table below.

Table 1: Troubleshooting JAGS

Message	**Interpretation**
`Unable to resolve the following parameters: x[1] Either supply values for these nodes with the data or define them on the left hand side of a relation.`	You are using **x** in a function but have NA for x[1]. You must define x[1] to perform this operation. See here.
`Possible directed cycle involving some or all of the following nodes: y[1] y[2]`	You have defined y[1] to depend on y[2] and vice-versa. This is called a directed cycle and is not allowed in JAGS. See here.
`Error: Error in node ... Failure to calculate log density`	Occurs when there are illegal values on the lhs. For example, variables that take on undefined values, like log of a negative. You will also get this with a Poisson distribution if you give it continuous numbers as data.
`Syntax error, unexpected '}', expecting $end`	Occurs when there are mismatched parentheses.

Continued on next page

Table 1 – *Continued from previous page*

Message	**Interpretation**
`Error in jags.model("beta", data = data, n.chain = 1, n.adapt = 1000) : Error in node y[7] Invalid parent values`	Occurs when there is an illegal mathematical operation or argument on the rhs. For example, negative values for argument to beta or Poisson distribution, division by zero, log of a negative, etc.
`Error in setParameters(init.values[[i]], i) : Error in node sigma.s[1] Attempt to set value of non-variable node`	You have a variable in your init list that is not a stochastic node in the model, i.e., it is constant.
`Error in jags.samples(model, variable.names, n.iter, thin, type = "trace", : Failed to set trace monitor for node ...`	The variable list in your coda.samples or jags.samples statement includes a variable that is not in your model. It also may mean that you asked JAGS to monitor a vector that does not have an initial value. You can fix this by giving the vector any initial value.
`Error: Error in node x[3,5,193] All possible values have probability zero`	You have uninitialized values for **x**.
`Error in jags.model("LogisticJAGS.R", data = data, inits, : RUNTIME ERROR: Unable to evaluate upper index of counter i`	You omitted the value for the upper range of the loop from the data statement.
`Error in jags.model("LogisticJAGS.R", data = data, inits, n.chains = length(inits), : RUNTIME ERROR: Unknown parameter sgma`	You misspelled a parameter name. In this case, `sgma` should have been `sigma`. Rejoice, this is an easy one!
`multiple definitions of node [x]`	You probably forgot the index on a variable within a for loop.
`Wrong number of arguments to distribution`	You have a <- instead of a ~ on the lhs of the distribution

Continued on next page

Table 1 – *Continued from previous page*

Message	**Interpretation**
`Error in jags.model("model", data = data, inits = inits, n.adapt = 3) : Length mismatch in inits`	You have a list of inits that specifies more than one chain, but you failed include the number of chains in the jags.model function. Adding the `n.chain = length(inits)` to the jags.model function will fix it.
`Error in jags.model("model", data = data, inits = inits, n.adapt = 3000) : Error in node y[15] Observed node inconsistent with unobserved parents at initialization`	This will happen whenever you have latent 0–1 quantities, as is common in mark recapture or occupancy models, and you fail to initialize them. Initialize these latent states at 1.
`Slicer stuck at value with infinite density`	From Martyn Plummer on the JAGS listserv, "Distributions with a shape parameter (Beta, Dirichlet, Gamma) can cause trouble when the shape parameter is close to zero and the probability mass gets concentrated into a single point." Alter your priors, use offsets, or truncate to prevent this from happening.
When using `gelman.diag( ). Error in chol.default(W) : the leading minor of order 7 is not positive definite`	You have derived quantities in you coda output that are functions of parameters you estimate. Set the argument `multivariate = FALSE` on `gelman.diag`.

References

Gelman, A. and D. B. Rubin, 1992. Inference from iterative simulation using multiple sequences. *Statistical Science* **7**:457–472.

Heidelberger, P. and P. D. Welch, 1983. Simulation run length control in the presence of an initial transient. *Operations Research* **31**:1109–1044.

Hobbs, N. T. and M. B. Hooten, 2015. Bayesian models: A statistical primer for ecologists. Princeton University Press, Princeton, New Jersey, USA.

Knops, J. M. H. and D. Tilman, 2000. Dynamics of soil nitrogen and carbon accumulation for 61 years after agricultural abandonment. *Ecology* **81**:88–98.

Link, W. A. and M. J. Eaton, 2012. On thinning of chains in MCMC. *Methods in Ecology and Evolution* **3**:112–115.

McCarthy, M. A., 2007. Bayesian methods for ecology. Cambridge University Press, Cambridge, United Kingdom.

Plummer, M., 2015. JAGS Version 4.0.0 user manual.

Plummer, M., N. Best, K. Cowles, K. Vines, D. Bates, R. Almond, and A. Magnusson, 2016a. coda: Output analysis and diagnostics for MCMC. R package version 0.18-1.

Plummer, M., A. Stukalov, and M. Denwood, 2016b. rjags: Bayesian graphical models using MCMC. R package version 4-6.

R Development Core Team, 2016. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria.

Raftery, A. E. and S. M. Lewis, 1995. The number of iterations, convergence diagnostics and generic metropolis algorithms. In W. R. Gilks, D. J. Spiegelhalter, and S. Richardson, editors, *Practical Markov Chain Monte Carlo*. Chapman and Hall, London, United Kingdom.

Youngflesh, C., 2018. MCMCvis: Tools to visualize, manipulate, and summarize mcmc output. *Journal of Open Source Software* **3(24)**.