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]Biology Department, University of Maryland, College Park MD, 20742

August 5, 2016

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 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]

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 accomplish the same thing with less time and effort using the BUGS language.

The BUGS language is currently implemented in three flavors of software: OpenBUGS, WinBUGS, and JAGS. OpenBUGS and WinBUGS run on Windows operating systems, while JAGS was specifically constructed to run multiple platforms, including Mac OS 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 that the GUI involves far to much tedious pointing and clicking and doesn't provide the flexibility that is needed for serious work.

[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.

There are two other options for software to approximate marginal posterior distributions, both of which are reported to be 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), which 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 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 and applies to a somewhat restricted set of problems.

3 Introducing JAGS

In this course we will use JAGS, which stands somewhat whimsically for "Just another Gibbs Sampler". There are three reasons we have chosen JAGS as the language for this course. First and most important, is because my experience is that JAGS is less fussy than WinBUGS (or OpenBUGS) which can be notoriously difficult to debug. Second is that JAGS runs on all platforms, which makes collaboration easier. Finally, 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] Very 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 (its manual can be found here) that is specifically developed for spatial models, but we know virtually nothing about it. However, if you are interested in landscape ecology otherwise have an interest in spatial modeling, we urge you to look into it after completing this tutorial.

This tutorial will use a simple example of 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. The model that we will use is the a 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. 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). Happily, both the population example and the ecosystem example can 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) and ecosystem scientists can take comfort in the application to nitrogen dynamics (e.g. Knops and Tilman 2000).

Consider the model,

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

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[4], $\frac{r}{K}$ 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]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.

$$\begin{aligned}
\mu_i &= r - \frac{r x_i}{K}, & (2)\\
[\, r, K, \tau \mid \mathbf{y}] &\propto \prod_{i=1}^{n} [\, y_i \mid \mu_i, \tau] \, [r] \, [K] \, [\tau] \, , \\
[r, K, \tau \mid \mathbf{y}] &\propto \prod_{i=1}^{n} \text{normal}\,(y_i \mid \mu_i, \tau) \times \text{gamma}\,(K \mid .001, .001) & (3)\\
&\quad \text{gamma}\,(\tau \mid .001, .001)\, \text{gamma}\,(r \mid .001, .001) \, ,
\end{aligned}$$

where the priors are vague distributions for quantities that must, by definition, be positive. Note that we have used the precision (τ) as a 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.[5] 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. However, we are all for doing things nimbly in 15 minutes that might otherwise take a sweaty hour 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: Writing a DAG. There is no x in the posterior distribution. What does this mean? Draw the Bayesian network, or DAG, for this model.

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) into a specification of an MCMC sampler. It is important for you to see the correspondence between the model (equation) and the code. JAGS parses this code, sets up proposal distributions and steps in the MCMC algorithm returns the MCMC chain for each parameter. These chains form the basis for approximating posterior distributions and associated statistics, i.e., means, medians, standard deviations, and quantiles. As we will soon learn, it easy to derive chains

[5]And also a source of hair loss when you forget to use precision rather than variance.

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 of 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.

```
## 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

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 constructing stochastic models. The syntax closely resembles R, but there are differences and of course, JAGS is far more limited. Detailed tables of functions and distributions can be found here, taken from the JAGS manual (Plummer, 2015). 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 would be wise to read the the JAGS manual at some point as well.

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 4-6.

4.1 Mac OS

Go to SourceForge and download `JAGS-4.2.0.dmg` to get the disk mounting image. Install as you would any other Mac software.

4.2 Windows

Go to SourceForge and download `JAGS-4.2.0.exe`. Occasionally, Windows users have problems loading `rjags` from R after everything has been installed properly. In all cases we have encountered, this problem 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-4.2.0.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 3) 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 3 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 3) 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.

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 3 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?

5.2 Technical notes

5.2.1 The model statement

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

```
model{
...
} # end of model
```

5.2.2 `for loops`

Notice that the for loop replaces the $\prod_{i=1}^{n}$in the likelihood. 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]
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[6] 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. As an alternative to giving an explicit argument for the

[6] 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.

number of iterations (e.g., n and m above), you can use the length() function. For example we could use:

```
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. If you would like, you can 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 `<-` operator

Note that, unlike R, you do not have the option in JAGS to use the `=` sign in an assignment statement. You must use `<-`.

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 `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.

```
## Logistic example for Primer
model{
  # priors
  K ~ dunif(0, 4000)
  r ~ dunif (0, 2)
  sigma ~ dunif(0, 100)
  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 library `SESYNCBayes` which has the data frame `Logistic`, which we then order by `PopulationSize`.[7] Next, we specify the initial conditions for the

[7] It would be very instructive for you to omit this ordering after you have successfully completed exercises 8 and 10. 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 chain in the statement `inits`. This is exactly the same thing as you did when you wrote you MCMC code and assigned a guess to the first element in the chain.

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

inits = list(
  list(K = 1500, r = .2, sigma = 1),
  list(K = 1000, r = .15, sigma = 5),
  list(K = 900, r = .3, sigma = 10))

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

n.adapt = 5000
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)
zm = coda.samples(jm, variable.names = c("K", "r", "sigma", "tau"),
n.iter = n.iter, n.thin = 1)
```

Algorithm 3: R code for running logisitics 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,[8] which are needed

[8]We start my work with a single chain. Once everything seems to be running, we add additional ones.

for 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 = 5),
  list(K = 900, r = .3, sigma = 10))
```

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. 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( )`[9] 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 data 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.[10] 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 = 5000
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)
```

[9]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!

[10]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 library `rjags`. Remember that MCMC required generating random numbers, so we use `set.seed(1)` to assure that our results are exactly reproducible.[11] 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.[12] 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 in this case, the file resided in the current working directory. 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.

[11]With enough iterations, this is not really necessary, but it is nice for teaching.
[12]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, 100)
  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 logistic regression. 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. You will find this a very convenient way to keep all your code in the same R script.

5.3.2 Parallelizing your JAGS run

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.

While there is nothing wrong with this approach, you can significantly speed up your model run time (up to a factor of n) by iterating all chains *simultaneously*, i.e. running the chains in parallel.[13] To do this, we will instruct R to have JAGS run one chain per core,[14] while leaving at least one core free to handle all other computer operations. For a JAGS model with three chains, you would need a CPU with ≥ 4 cores. We will refer to the group of cores used by R in this fashion as a cluster.

We implement this parallel run in R using algorithm 5. First, we create a function called `initFunc` to randomly draw each parameter's initial value from a subset[15] of its support:

```
initFunc <- function (){
return(list(
  K = runif(1, 10, 2000),
  r = runif(1, .1, 2),
  sigma = runif(1, 1, 80))}
```

We will use this function to automatically provide a different set initial values to each chain in our JAGS model.

After loading the `parallel` library in R, execute the command `detectCores()` to determine how many cores your computer has. Assuming you have at least 4 cores, the statements:

```
cl <- makeCluster(3)
clusterExport(cl, c("data", "initFunc", "n.adapt", "n.update", "n.iter"))
```

will create a cluster of three cores in R called `cl`, and pass the `data`, `inits`, `n.adapt`, `n.update`, and `n.iter` objects to each core in `cl` using the `clusterExport` function.

With a few key modifications (discussed below), we embed our previous R code for running the JAGS model inside the cluster `cl`. This allows the cluster to direct the R

[13]Remember, each chain starts with its own initial values and does not depend on the other chains.

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

[15]Technically, you can use the entire support for this purpose. However, for more complicated models, such as those with latent states, this will sometimes cause JAGS to return an error if it has trouble computing the likelihood. We will come back to this later in the course.

commands to each core simultaneously, collect output from each core, and combine the output into a list where each element corresponds to the output from a single core. 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 `zmCore`[16] and `out`, respectively:

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

Notice four things. First, we must load the library `rjags` and set the seed separately for each core, meaning the `library(rjags)` and `set.seed(1)` statements need to be inside the `clusterEvalQ` function. Second, we invoke our `initFunc` function directly in the `jags.model` statement.[17] Third, we only call a single chain instead of three, per core. Fourth, we use the `as.mcmc` function to save the output from the `coda.samples` statement as an MCMC object (again, more about that very soon).

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

[16] Why is it possible to name the MCMC output from each chain the same thing?

[17] To see how this works type `initFunc()` in the R console repeatedly. This is an easy way to avoid having each chain receive the same initial values. You can use this approach for JAGS models run sequentially as well.

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.

```
# make initial values function
initFunc <- function (){
return(list(
  K = runif(1, 10, 2000),
  r = runif(1, .1, 2),
  sigma = runif(1, 1, 80)))}

# run JAGS model in parallel
library(parallel)
set.seed(1)
detectCores()
cl <- makeCluster(3) # Here we use three cores
clusterExport(cl, c("data", "initFunc", "n.adapt", "n.update", "n.iter"))
out <- clusterEvalQ(cl, {
  library(rjags)
  jm = jags.model("LogisticJAGS.R", data = data, inits = initFunc,
  n.chains = 1, n.adapt = n.adapt)
  update(jm, n.iter = n.update)
  zmCore = coda.samples(jm, variable.names = c("K", "r", "sigma", "tau"),
  n.iter = n.iter, thin = 1)
  return(as.mcmc(zmCore))
})

zmP <- mcmc.list(out)
stopCluster(cl)
```

Algorithm 5: R code for running logisitics JAGS script in parallel

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 cores, try running only 2 chains in parallel when doing this exercise. If you have fewer than 3 cores, work with a classmate that has at least 3 cores.

6 Output from JAGS

6.1 coda objects

The `coda` package (Plummer et al., 2016a) allows you to produce nice tabular summaries of statistics describing marginal posterior distributions from coda objects (i.e., MCMC lists). They are also really handy for plotting and for testing MCMC output for convergence. That said, Tom does most of his work using JAGS objects, which will be described in the next section. JAGS objects are part of the `rjags` package which you have already loaded.

6.1.1 Summarizing 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, or more precisely, an MCMC list. Assuming that the `coda` library is loaded, (i.e. `library(coda)`), you can obtain a summary of statistics from MCMC chains contained in a coda object using `summary(co)` where `co` is a coda object. All of the variables in the `variable.names = c( )` argument to the `coda.samples` function will be summarized. For the logistic example, `summary(zm)` produces:

```
Iterations = 15001:25000
Thinning interval = 1
Number of chains = 3
Sample size per chain = 10000
1. Empirical mean and standard deviation
for each variable, plus standard error of the mean:
Mean         SD  Naive SE Time-series SE
K     1.238e+03 6.323e+01      3.651e-01
```

```
r     2.006e-01 9.746e-03      5.627e-05
sigma 2.863e-02 2.989e-03     c1.726e-05
tau   1.259e+03 2.563e+02     c1.480e+00

2. Quantiles for each variable:

           2.5%       25%       50%       75%       95%
K     1.130e+03 1.194e+03 1.232e+03 1.276e+03 1.376e+03
r     1.812e-01 1.941e-01 2.007e-01 2.072e-01 2.196e-01
sigma 2.355e-02 2.652e-02 2.838e-02 3.048e-02 3.512e-02
tau   8.108e+02 1.077e+03 1.242e+03 1.422e+03 1.804e+03
```

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 columns `Naive SE` and `Time-series SE` refer only to the MCMC, and can be used as a rough indicator of convergence, although there are much better ways, below. We never use these for anything.

Each of the two tables above has the properties of a matrix.[18] You can output the cells of these tables using syntax as follows. To get the mean and standard deviation of r,

```
> summary(zm)$stat[2, 1:2]
       Mean          SD
0.199475142 0.009968849
```

To get the upper and lower 95% quantiles on K,

```
> summary(zm)$quantile[1, c(1, 5)]
    2.5%    97.5%
1147.160 1596.622
```

[18]Consider `m = summary(zm)` The object `m` is a list of two matrices, one for the table of means and the other for the table of quantiles. As with any list, you can access these tables with `m[1]` and `m[2]` or the syntax shown above. Try it.

Exercise 6: Summarizing coda objects. Build a table that contains the mean, standard deviation, median and upper and lower 2.5% CI for parameters from the logistic example. Output your table with 3 significant digits to .csv file readable by Excel (Hint: see the `signif()` function).

6.1.2 The structure of coda objects (MCMC lists)

So, what is a coda object? Technically, the coda object is an MCMC list. For the first chain, it looks something like this:

```
[[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
[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 and JAGS 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.

Exercise 7: Understanding coda objects. Modify your code to produce a coda object with 3 chains called `zm.short`, setting `n.adapt = 500`, `n.update = 500`, and `n.iter = 20`.

1. Output the estimate of σ for the third iteration from the second chain.
2. Output all of the estimates of r from the first chain.
3. Verify your answers by printing the entire chain, i.e. enter `zm.short` at the console.

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 or JAGS 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.

6.1.3 Manipulating coda objects

Any coda object can be converted to a data frame using syntax like:

```
df = as.data.frame(rbind(co[[1]], co[[2]], ....co[[n]]))
```

where `df` is the data frame, `co` is the coda object and `n` is the number of chains in the coda object, that is, the number of elements in the list. Once the coda object has been converted to a data frame, you can use any of the R tricks you have learned for manipulating data frames. The thing to notice here is the double brackets, which is how we refer to the elements of a list. Think about what this statement is doing.

Exercise 8: Convert the `zm` object to a data frame. Using the elements of data frame (not `zm`) as input to functions:

1. Find the maximum value of σ.
2. Estimate the mean of r for the first 1000 and last 1000 iterations in the chain.
3. Produce a publication quality plot of the posterior density of K.
4. Estimate the probability that the parameter K exceeds 1600 and the probability that K falls between 1000 and 1300. (Hint: look into using the `ecdf()` function.)

6.2 JAGS objects

6.2.1 Why another object?

The coda object is strictly tabular – it is a list of matrices where each element of the list an MCMC chain with rows holding iterations and columns holding parameters being monitored. This is fine when the parameters you are estimating are entirely scalar, but sometimes you want posterior distributions for all of the elements of vectors or matrices and in this case, the coda object can be quite cumbersome. For example, presume you would like to get posterior distributions on the *predictions* of your regression model. To do this, you wold simply ask JAGS to monitor the values of μ 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)
```

> **Exercise 9: Vectors in coda objects.** Modify your code to include estimates of μ and summarize the coda object. What if you wanted to plot the model predictions with 95% credible intervals against the data. How would you do that?

6.2.2 Summarizing the JAGS object

As an alternative, add the following below the `coda.samples` function:

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

If you run this and enter `zj` at the console, R will return the means of all the monitored variables.[19] Try it. If you want other statistics, you would use syntax like:

```
summary(zj$variable.name, FUN)$stat
```

that will summarize the variable using the function, `FUN`. The most useful of these is illustrated here:

```
hat = summary(zj$mu, quantile, c(.025, .5, .975))$stat
```

[19] There is a *very important* caveat here. If the `rjags` library is not loaded when you enter an jags object name, R will not know to summarize it, and you will get the raw iterations. There can be a lot of these, leaving you bewildered as they fly by on the console. If you simply load the library, you will get more well behaved output.

which produces the median and upper and lower .025% quantiles for μ, preserving its vector structure. You can also give JAGS objects as arguments to other functions, a very handy one being the empirical cumulative distribution function,[20] `ecdf()`. For example, the following code would estimate the probability that the parameter K is less that 900:

```
pK.lt.900 = ecdf(zj$K)(900)
```

`FUN` can also be a function your write yourself.

Exercise 10: Making plots with JAGS objects. For the logistic model:

1. Plot the observations of growth rate as a function of observed population size.
2. Overlay the median of the model predictions as a solid line.
3. Overlay the 95% credible intervals as dashed lines.
4. Prepare a separate plot of the posterior density of K.

6.2.3 The structure of JAGS objects (MCMC arrays)

JAGS objects have a list structure, as coda objects do, but instead of each element of the list holding a matrix for each chain, the JAGS objects holds an array. This is more easily illustrated than explained. The JAGS object below[21] below contains 5 iterations and two chains. Look at the object and think about how it is structured. Note how the vector structure is preserved for the 16 estimates of μ:

[20] See Hobbs and Hooten (2015) section 3.4.1.4 for discussion of quantile and cumulative distribution functions.

[21] Actually, `rjags` makes it hard to "see" the object. If `rjags` is loaded, it presumes you want summaries. If you want to look at a complete listing of a JAGS object you save it, quit R, and restart it, load the JAGS object without loading `rjags`. The JAGS object then has the structure shown in the example.

```
$K
, , 1

        [,1]     [,2]    [,3]     [,4]     [,5]
[1,] 1207.786 1214.638 1241.77 1403.883 1419.426

, , 2

        [,1]     [,2]     [,3]    [,4]     [,5]
[1,] 1262.989 1185.657 1197.613 1185.05 1190.977

attr(,"class")
[1] "mcarray"
attr(,"varname")
[1] "K"

$mu
, , 1

           [,1]       [,2]       [,3]       [,4]       [,5]
 [1,] 0.20525295 0.20691887 0.17244361 0.17436321 0.17310006
 [2,] 0.20345845 0.20512057 0.17097945 0.17306188 0.17182297
 [3,] 0.19610097 0.19774756 0.16497642 0.16772643 0.16658690
 [4,] 0.18748735 0.18911573 0.15794848 0.16148004 0.16045687
 [5,] 0.18407779 0.18569897 0.15516659 0.15900751 0.15803040
 [6,] 0.17456691 0.17616799 0.14740658 0.15211045 0.15126183
 [7,] 0.17259295 0.17418986 0.14579601 0.15067899 0.14985703
 [8,] 0.16972175 0.17131259 0.14345336 0.14859686 0.14781369
 [9,] 0.16900394 0.17059327 0.14286770 0.14807632 0.14730285
[10,] 0.16774779 0.16933446 0.14184279 0.14716539 0.14640889
[11,] 0.15769856 0.15926400 0.13364353 0.13987794 0.13925719
[12,] 0.15733966 0.15890434 0.13335070 0.13961767 0.13900177
[13,] 0.14370142 0.14523728 0.12222313 0.12972756 0.12929589
[14,] 0.13760010 0.13912307 0.11724501 0.12530303 0.12495379
[15,] 0.13616449 0.13768443 0.11607369 0.12426196 0.12393212
[16,] 0.13562614 0.13714494 0.11563444 0.12387157 0.12354899
[17,] 0.13544669 0.13696511 0.11548803 0.12374143 0.12342128
[18,] 0.13508779 0.13660545 0.11519519 0.12348117 0.12316586
[19,] 0.13024263 0.13175005 0.11124198 0.11996757 0.11971772
[20,] 0.11786054 0.11934180 0.10113932 0.11098839 0.11090581
[21,] 0.11696329 0.11844265 0.10040724 0.11033772 0.11026726
[22,] 0.11480988 0.11628470 0.09865026 0.10877612 0.10873475
[23,] 0.11391263 0.11538555 0.09791818 0.10812546 0.10809621
[24,] 0.10799076 0.10945117 0.09308647 0.10383106 0.10388181
[25,] 0.10350450 0.10495543 0.08942609 0.10057774 0.10068909
[26,] 0.09991549 0.10135883 0.08649778 0.09797507 0.09813491
[27,] 0.09471142 0.09614377 0.08225174 0.09420121 0.09443135
[28,] 0.08914845 0.09056905 0.07771286 0.09016709 0.09047238
[29,] 0.08591834 0.08733212 0.07507738 0.08782469 0.08817362
[30,] 0.08502109 0.08643297 0.07434530 0.08717403 0.08753507
[31,] 0.08322659 0.08463467 0.07288115 0.08587269 0.08625798
[32,] 0.08179098 0.08319603 0.07170983 0.08483163 0.08523631
[33,] 0.07838142 0.07977927 0.06892794 0.08235910 0.08280984
[34,] 0.07730472 0.07870029 0.06804944 0.08157830 0.08204359
[35,] 0.07263900 0.07402472 0.06424264 0.07819484 0.07872316
[36,] 0.07138285 0.07276591 0.06321774 0.07728391 0.07782919
[37,] 0.06976780 0.07114744 0.06190000 0.07611271 0.07667981
[38,] 0.06276923 0.06413408 0.05618980 0.07103752 0.07169916
[39,] 0.06133362 0.06269545 0.05501847 0.06999645 0.07067749
[40,] 0.06025692 0.06161647 0.05413998 0.06921566 0.06991124
[41,] 0.05900076 0.06035766 0.05311508 0.06830472 0.06901728
[42,] 0.05810351 0.05945851 0.05238300 0.06765406 0.06837873
[43,] 0.05307890 0.05442328 0.04828337 0.06401033 0.06480288
[44,] 0.05182274 0.05316447 0.04725846 0.06309940 0.06390892
[45,] 0.04392692 0.04525196 0.04081618 0.05737354 0.05828973
[46,] 0.03854340 0.03985707 0.03642372 0.05346955 0.05445846
[47,] 0.03656945 0.03787895 0.03481315 0.05203808 0.05305366
[48,] 0.03172428 0.03302354 0.03085994 0.04852449 0.04960552
[49,] 0.02975033 0.03104542 0.02924937 0.04709303 0.04820072
[50,] 0.02077780 0.02205393 0.02192860 0.04058637 0.04181527

, , 2

           [,1]       [,2]       [,3]       [,4]       [,5]
 [1,] 0.19167519 0.20890879 0.19915023 0.19141099 0.19777146
 [2,] 0.19007655 0.20704629 0.19739345 0.18970357 0.19601658
 [3,] 0.18352213 0.19941003 0.19019068 0.18270312 0.18882155
 [4,] 0.17584866 0.19047002 0.18175816 0.17450747 0.18039811
 [5,] 0.17281124 0.18693127 0.17842029 0.17126336 0.17706383
 [6,] 0.16433845 0.17706001 0.16910938 0.16221400 0.16776294
 [7,] 0.16257995 0.17501126 0.16717693 0.16033583 0.16583256
 [8,] 0.16002212 0.17203126 0.16436609 0.15760395 0.16302475
 [9,] 0.15938267 0.17128626 0.16366338 0.15692098 0.16232279
[10,] 0.15826362 0.16998251 0.16243363 0.15572578 0.16109438
[11,] 0.14931123 0.15955250 0.15259569 0.14616419 0.15126702
[12,] 0.14899151 0.15918000 0.15224434 0.14582271 0.15091605
[13,] 0.13684184 0.14502498 0.13889285 0.13284627 0.13757892
[14,] 0.13140647 0.13869248 0.13291981 0.12704102 0.13161231
[15,] 0.13012756 0.13720248 0.13151439 0.12567507 0.13020841
[16,] 0.12964797 0.13664373 0.13098736 0.12516285 0.12968194
[17,] 0.12948810 0.13645747 0.13081168 0.12499210 0.12950645
[18,] 0.12916837 0.13608497 0.13046033 0.12465062 0.12915548
[19,] 0.12485205 0.13105622 0.12571704 0.12004057 0.12441729
[20,] 0.11382143 0.11820496 0.11359529 0.10825932 0.11230858
[21,] 0.11302211 0.11727371 0.11271690 0.10740561 0.11143114
[22,] 0.11110374 0.11503871 0.11060877 0.10535670 0.10932528
[23,] 0.11030442 0.11410745 0.10973038 0.10450299 0.10844784
[24,] 0.10502891 0.10796120 0.10393303 0.09886848 0.10265672
[25,] 0.10103231 0.10330494 0.09954109 0.09459991 0.09826951
[26,] 0.09783503 0.09957994 0.09602754 0.09118506 0.09475974
[27,] 0.09319898 0.09417869 0.09093289 0.08623352 0.08967057
[28,] 0.08824320 0.08840493 0.08548689 0.08094050 0.08423043
[29,] 0.08536564 0.08505243 0.08232470 0.07786713 0.08107164
[30,] 0.08456632 0.08412118 0.08144631 0.07701342 0.08019420
[31,] 0.08296768 0.08225868 0.07968953 0.07530599 0.07843931
[32,] 0.08168877 0.08076867 0.07828411 0.07394005 0.07703541
[33,] 0.07865136 0.07722992 0.07494624 0.07069594 0.07370112
[34,] 0.07769217 0.07611242 0.07389218 0.06967149 0.07264819
[35,] 0.07353571 0.07126992 0.06932456 0.06523218 0.06808549
[36,] 0.07241666 0.06996616 0.06809482 0.06403698 0.06685707
[37,] 0.07097789 0.06828991 0.06651372 0.06250030 0.06527768
[38,] 0.06474319 0.06102616 0.05966230 0.05584133 0.05843363
```

```
[39,] 0.06346428 0.05953616 0.05825688 0.05447539 0.05702972
[40,] 0.06250509 0.05841865 0.05720281 0.05345094 0.05597679
[41,] 0.06138605 0.05711490 0.05597307 0.05225574 0.05474837
[42,] 0.06058673 0.05618365 0.05509468 0.05140202 0.05387093
[43,] 0.05611054 0.05096865 0.05017571 0.04662123 0.04895725
[44,] 0.05499149 0.04966490 0.04894597 0.04542603 0.04772883
[45,] 0.04795747 0.04146989 0.04121616 0.03791335 0.04000734
[46,] 0.04316155 0.03588238 0.03594584 0.03279108 0.03474269
[47,] 0.04140305 0.03383363 0.03401338 0.03091291 0.03281231
[48,] 0.03708672 0.02880488 0.02927009 0.02630285 0.02807413
[49,] 0.03532822 0.02675613 0.02733764 0.02442469 0.02614375
[50,] 0.02733502 0.01744362 0.01855377 0.01588755 0.01736933

attr(,"class")
[1] "mcarray"
attr(,"varname")
[1] "mu"

$r
, , 1

          [,1]     [,2]      [,3]      [,4]      [,5]
[1,] 0.2167378 0.218428 0.1818142 0.1826917 0.1812734

, , 2

          [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 0.2019065 0.2208288 0.2103936 0.2023385 0.2090027

attr(,"class")
[1] "mcarray"
attr(,"varname")
[1] "r"

$sigma
, , 1

           [,1]       [,2]       [,3]       [,4]       [,5]
[1,] 0.02868276 0.02953674 0.02596939 0.03019077 0.03100996

, , 2

          [,1]       [,2]       [,3]       [,4]       [,5]
[1,] 0.0264764 0.02691578 0.02387002 0.02591833 0.02631205

attr(,"class")
[1] "mcarray"
attr(,"varname")
[1] "sigma"
```

6.2.4 Manipulating JAGS objects

To understand how you can extract elements of the JAGS object you need to know its dimensions. For MCMC arrays that include scalars and vectors, each element in the list has three dimensions. For the scalars in the list, the first dimension[22] is always = 1, the second dimension = the number of iterations, and the third dimension = the number of chains. For vectors, the first dimension of the JAGS object is the length of the vector, the second dimension = the number of iterations, and the third dimension = the number of chains. An easy way to remember this is simply to enter `dim(jags.object$parameter)` at the console. Because the dimensions are named, there is no ambiguity about the structure of the object. So for example, the dimensions of `mu` in the `zj` JAGS object:

```
> dim(zj$mu)
```

[22] This gives the the length. A scalar is a vector with length = 1.

```
##           iteration     chain
##        50     10000         3
```

A vector containing all iterations of the second chain for `K`:

```
zj$K[1,,2]
```

A matrix for `sigma` with 2 rows, one for each chain, containing iterations 1 to 1000:

```
zj$sigma[1,1:1000,]
```

A matrix containing 16 rows, one for each element of `mu` containing elements from the third chain:

```
zj$mu[,,3]
```

So, if you wanted to find the mean of the third prediction of μ across all iterations and all chains, you would use:

```
> mean(zj$mu[3,,])
## [1] 0.09473373
```

Exercise 11: Manipulating JAGS objects.

1. Calculate the median of the second chain for K.
2. Calculate the upper and lower 95% quantiles for the 16th estimate of μ without using the `summary` function.
3. Calculate the probability that the 16th estimate of $\mu < 0$.

6.2.5 Converting JAGS objects to coda objects

It is possible to convert individual elements of the JAGS object to coda objects using the as.mcmc.list function, which can be helpful for using convergence diagnostics (as described in the next section) if you haven't created a coda object directly using the `coda.samples` function. The syntax is:

```
codaObject = as.mcmc.list(JAGSObjectName$Parameter)
```

So, for example, if you want to create a coda object for K, you would use:

```
K.coda = as.mcmc.list(zj$K)
```

It is not possible to convert all of the elements of a JAGS object into coda objects in a single statement, i.e., the following will not work:

```
# wrong
jm = as.mcmc.list(zj)
```

7 Which object to use?

Coda and JAGS objects are both useful, and for most of our work we usually create both types. Coda objects are somewhat better for producing tabular summaries of estimates and are required for checking convergence, but JAGS objects are *far* better for plotting. Coda objects are also produced by WinBUGS and OpenBUGS, so if you ever need to use them, everything you learned about coda objects will apply. Tom generally starts development of models using coda objects alone, and when he reaches the final output stage, he produces both types of objects with multiple chains.

8 Checking convergence using the coda package

Remember from lecture that the MCMC chain will provide a reliable estimate of the posterior distribution only after it has converged, which means that it is no longer sensitive to initial conditions and that the estimates of parameters of the posterior distribution will not change appreciably with additional iterations. The `coda` package contains a tremendous set of tools for evaluating and manipulating MCMC chains produced in coda objects (i.e., MCMC lists). 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. For all of these to work, the `coda` library must be loaded. Also see Hobbs and Hooten (2015) section 7.

8.1 Trace and density plots

There are three useful ways to plot the chains and the posterior densities. The first one, `plot(codaObject)`, produces a collage. The second two, `traceplot(codaOobject)` and `densplot(codaObject)` produce plots one panel at a time. You will examine how to use these for diagnosing convergence in the subsequent exercise.

8.2 Gelman and Rubin diagnostics

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). 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.

The syntax is:

```
gelman.diag(codaObject)
```

It is *vital* to know that if the coda object contains *derived quantities* that are *perfectly correlated* with other elements in the coda object (i.e. quantities calculated from model parameters like `mu`) you will get an error message from R causing you of creating a matrix for which "the leading minor of order 7 is not positive definite", which of course makes it really clear what the problem is. In this case the multivariate potential scale reduction factor (PSRF) computed as part of the Gelman output is the secret culprit. You can turn this off by adding the argument `multivariate = FALSE` to `gelman.diag`.

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.

8.3 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. 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)
```

8.4 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 and estimate of the number of iterations required for convergence for each of the parameters being estimated. The syntax is:

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

Exercise 12: Assessing convergence. Rerun the logistic model with `n.adapt = 100`. Then do the following:

1. Keep the next 500 iterations. Assess convergence visually with `traceplot` 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?

9 Monitoring deviance and calculating DIC

It is often a good idea to report the deviance of a model which is defined as $-2\log\left[P\left(y \mid \theta\right)\right]$. 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 `jags.samples` statement and/or 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
```

10 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.

11 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 so 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 parsing model file: syntax error on line 9 near "="`	You used an = instead of <- for assignment.
`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.
`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.

Continued on next page

Table 1 – *Continued from previous page*

Message	**Interpretation**
`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
`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.

Continued on next page

Table 1 – *Continued from previous page*

Message	**Interpretation**
`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.

Subject Index