An Ecological Modeler's Primer on JAGS

N. Thompson Hobbs

April 1, 2013

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

Contents

1 Aim

The purpose of this Primer is to teach the programming skills needed to estimate the marginal posterior distributions of parameters and derived quantities of interest in ecological models using software implementing Mote Carlo Markov chain methods. Along the way, I will reinforce some of the ideas and principals that we have learned in lecture. The Primer is organized primarily as a tutorial and contains only a modicum of reference material. [1]There is an important supplement to this primer, excised from the JAGS users manual, that covers functions and distributions.

2 Introducing MCMC Samplers

WinBugs, OpenBUGS, and JAGS are three systems of software that implement Monte Carlo Markov Chain 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 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 Unix. 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 I will teach. My experience is that that the GUI involves far to much tedious

[1]Other good references on the BUGS language are the WinBUGS manual (http://www.mrc-bsu.cam.ac.uk/bugs/winbugs/contents.shtml, look for the manual .pdf link) which has lots of detailed treatment of functions and syntax as well asMcCarthy (2007). 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.

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

3 Introducing JAGS

In this course we will use JAGS, which stands somewhat whimsically for "Just another Gibbs Sampler." There are three reasons I have chosen JAGS as the language for this course. First and most important, is because my experience is that JAGS is *far* 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.[2]

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 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. Consider the model,

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

[2]There is also software called GeoBUGS that is specifically developed for spatial models, but I know virtually nothing about it. However, if you are interested in landscape ecology otherwise have an interest in spatial modeling, I urge you to look into it after completing this tutorial. The manual can be found at http://www.mrc-bsu.cam.ac.uk/bugs/winbugs/contents.shtml

which, of course, is a linear model with intercept r and slope $\frac{r}{K}$. Note that these quantities enjoy a sturdy biological interpretation; r is the intrinsic rate of increase, $\frac{r}{K}$ 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) at which $\frac{dN}{dt} = 0$. Presume we have some data consisting of observations of per capita rate of growth of N 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., $y_i = \frac{1}{N_i}\frac{dN_i}{dt}$, $x_i = N_i$. A simple Bayesian model specifies the joint distribution of the parameters and data as

$$\begin{aligned}
\mu_i &= r - \frac{rx_i}{K} \\
P\left(r, K, \tau \mid \mathbf{y}, \mathbf{x}\right) &\propto \prod_{i=1}^{n} P\left(y_i \mid \mu_i, \tau\right) P\left(K\right) P\left(\tau\right) P(r) \\
P\left(r, K, \tau \mid \mathbf{y}, \mathbf{x}\right) &\propto \prod_{i=1}^{n} \text{normal}\left(y_i \mid \mu_i, \tau\right) \times \\
&\quad \text{gamma}\left(K \mid .001, .001\right) \text{gamma}\left(\tau \mid .001, .001\right) \text{gamma}\left(r \mid .001, .001\right),
\end{aligned} \tag{2}$$

where the priors are uninformative. Now, I 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, I am 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.

This code illustrates the purpose of JAGS (and other BUGS software): to translate the numerator of Bayes theorem (a.k.a., the joint, e.g., equation 2) into a specification of an MCMC sampler. JAGS parses this code, sets up proposal distributions and steps in the Gibbs sampler and returns the MCMC chain for each parameter. These chains form the basis for estimating posterior distributions and associated statistics, i.e., 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, $K/2$ (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. If this seems

as if it might be useful to you, you should continue reading.

Algorithm 1 Linear regression example

```
##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)
    }
} #end of model
```

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 the 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 in the supplementary material [JAGS functions and distributions.pdf, taken from the JAGS manual (Plummer, 2011). Rather than focus on these details, this tutorial presents general introduction JAGS models, how to call them from R, how to summarize their output, and how to check convergence.

4 Installing JAGS

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 from a CRAN mirror of your choice. Now go to `http://www-ice.iarc.fr/~martyn/software/jags/` and look in the section under downloads. Click on the files page link and then click on Download JAGSdist-.dmg (4.7 MB) where ____is the

number of the latest version to get the disk mounting image. Install as you would any other Mac software.

4.1 Windows

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 from a CRAN mirror of your choice. Check the version number of `rjags`. Go to `http://sourceforge.net/projects/mcmc-jags/files/JAGS/`. Click on 3x then JAGS-3.3.0.exe.

Occasionally, students using windows operating systems have problems loading `rgags` from R after everything has been installed properly. In all cases I 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.

4.2 LINUX

There is a link to the path for binaries found at `http://mcmc-jags.sourceforge.net/` . If you want to compile from source code, there are detailed instructions at `http://yusung.blogspot.com/2009/01/install-jags-and-rjags-in-fedora.html`There are tar files found at `http://sourceforge.net/projects/mcmc-jags/files/JAGS/3.x/Source/`. You want `JAGS-3.0.0.tar.gz`. My guess is that you will need to download the `rjags` package in R before installing `JAGS`.

5 Running JAGS

5.1 The JAGS model

Study the relationship between the numerator of Bayes theorem (equation 2) 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 2should be pretty obvious, but there are a few things to point out. Priors and likelihoods are specified using the ~ 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 2) and the code. In this example, I chose uninformative gamma priors for r, K andτ because they must be positive. I chose a normal likelihood because the values of y and μ are continuous and can take on positive or negative values.

Exercise: always plot your priors Plot priors for each parameter, scaling the x axis appropriately for each value–r should be about .2, K about 1200, and τ should be about 2500. Discuss with you lab mates if gamma(θ|.001, .001) is vague for all parameters, i.e.,

$\theta = r, K, \tau$. Be sure to include lots of x points in your plots to get a good interpolation, at least 1000.

5.2 Technical notes

5.2.1 The model statement

Your entire model must be enclosed in

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

I am in the habit of putting a hard return (a blank line) after the `} #end of model` statement. If you fail to do so, you may get the message `#syntax error, unexpected NAME, expecting $end.` (This may have been fixed in the newer versions of JAGS, but just to be safe....)

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 (actually, 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[3],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 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 μ 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 species 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 $\prod_{i=1}^{n}\prod_{j=1}^{m}$.....then this dual product is specified in JAGS using nested for loops, i.e.,

```
    for(i in 1:n){
```

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

As an alternative to giving an explicit argument for the 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: using `for` loops Write a code fragment to set vague normal priors [`dnorm(0,10e-6)`] for 5 regression coefficients stored in the vector `B`.

5.2.3 Specifying priors

We specify priors in JAGS as `paramater ~ distribution(shape1, shape2)`. See the supplementary material for available distributions. Note that in the code (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`. 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:

```
    sigma~dunif(0,100) #presuming this more than brackets the posterior of sigma
    tau <- 1/sigma^2
```

There are times when this seems to work better than the gamma prior for

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

I don't use any vector operations in the example code, 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 supplementary material. However, you cannot form vectors using syntax like `c()`. If you need a specific-valued vector in JAGS, read it in as data.

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, I often use something like:

```
mu[i]<- log(max(.0001,expression))
```

Exercise: Coding the JAGS script Carefully write out all of the code in the Logistic example (algorithm 1) into a program window in R. You may save this code in any directory that you like and may name it anything you like. I use names like `logistic exampleJAGS.R` which lets me know that the file contains JAGS code. Using an R extension allows me to search these files easily with Spotlight.

5.3 Running JAGS from R

We implement our model using R (algorithm 2.) We will go through the R code step by step. We start by bringing the growth rate data into R as a data frame. 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 you MCMC code and assigned a guess to the first element in the chain. There are two important things to notice about this statement.

Algorithm 2 R code for running logistic JAGS script.

```
setwd("/Users/Tom/Documents/Ecological Modeling Course/JAGS Primer")
rm(list=ls())
pop.data=(read.csv("Logistic Data II.csv"))
names(pop.data)=c("Year","Population Size", "Growth Rate")
inits=list(
list(K=1500, r=.2, tau=2500)
)#chain 1

n.xy = nrow(pop.data)
data=list(

    n=n.xy,
    x=as.real(pop.data$"Population Size"),
    y=as.real(pop.data$"Growth Rate")
    )

library(rjags)
##call to JAGS
library(rjags)
##call to JAGS
n.adapt=5000
n.update = 10000
n.iter = 10000
jm=jags.model("Logistic example JAGS.R",data=data,inits,n.chains=length(inits),
n.adapt = n.adapt)
#Burnin the chain.
update(jm, n.iter=n.update)
#generate coda object
zm=coda.samples(jm,variable.names=c("K", "r", "sigma"), n.iter=n.iter, n.thin=1)
```

First, initial conditions must be specified as as "list of lists", as you scan 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[3], which are needed for some tests of convergence and to calculate DIC (more about these tasks later). For example, if you want three chains, you would use something like:

```
inits=list(
list(K=1500, r=.5, tau=1500), #chain 1
list(K=1000, r=.1, tau=1000), #chain 2
list(K=900, r=.3, tau=900) #chain 3
) #end of inits list
```

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 set of initial conditions, that is, a list of lists.

Which variables in your JAGS code require initialization? Anything you are estimating must be initialized, which means anything on the right hand side of a conditioning symbol (except, of course, data) Think about it this way. When you were writing your own Gibbs sampler, 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, I 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,

```
n.xy = nrow(pop.data)
data=list(n=n.xy,
```

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

```
        x=as.real(pop.data$"Population Size"),
        y=as.real(pop.data$"Growth Rate"))
```

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.real(pop.data$"Population Size")
```

says that the x vector in your JAGS program (algorithm 1) is composed of the column in your data frame called `Population Size` and the bit that reads

```
    y=as.real(pop.data$"Growth Rate")
```

creates a `y` vector required by the JAGS program from the column in your data frame called `Growth Rate` (pretty cool, I think). Notice that if I had named the variable `Growth.Rate` instead of `Growth Rate`, the quotes would not be needed. 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 `pop.data` is a data frame I used `as.real( )` to be sure that JAGS received real numbers instead of characters or factors, as can happen with data frames. 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 2) and it must be read as data in this statement[4]. 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:

```
    library(rjags)
    ##call to JAGS
```

[4]You could hard code the `for` index in the JAGS code, but this is bad practice.

```
n.adapt=5000
n.update = 10000
n.iter = 25000
jm=jags.model("Logistic example JAGS.R",data=data,inits,n.chains=length(inits),
    n.adapt = n.adapt)
#Burnin the chain.
update(jm, n.iter=n.update)
#generate coda object
zm=coda.samples(jm,variable.names=c("K", "r", "sigma"), 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 or come back to this tomorrow. First, we need to get the library `rjags`. 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. 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 BUGS code. Note that in this case, the file resided in the current working directory, which I specified at the top of the code (algorithm 2). Otherwise, you would need to specify the full path name. (It is also possible to embed the BUGS code within your R script, see Algorithm 3,). 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

Algorithm 3 Example of code for inserting BUGS code within R script. This should be placed above the `jags.model( )` statement (algorithm). You must remember to execute the code starting at sink and ending at sink every time you make changes in the model.

```
sink("logisticJAGS.R")
#This is the file name for the bugs code
cat(" model{

    K~dgamma(.001,.001)
    r~dgamma(.001,.001)
    tau~ dgamma(.001,.001)
    sigma<-1/sqrt(tau)
    #likelihood
    for(i in 1:n){
        mu[i] <- r - r/K * x[i]
        y[i] ~ dnorm(mu[i],tau)
    } #end of i for

} #end of model
",fill=TRUE)
sink()
```

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 in 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. Finally, `n.iter=n.iter` says we want 25000 elements in each chain and `n.thin` specifies how many of these to keep. For example, if n.thin = 10, we would store every 10th element. Sometimes setting `n.thin` > 1 is a good idea to reduce the size of the data files that you will analyze.

Exercise: Coding the logistic regression Write R code (Algorithm 2) to use the JAGS model to estimate the parameters, $r, K$and σ. When your model is running without error messages, proceed to get output, as described below.

6 Output from JAGS

6.1 coda objects

6.1.1 Summarizing coda objects

The `zm` object produced by the statement

```
zm=coda.samples(jm,variable.names=c("K", "r", "sigma"), n.iter=n.iter,n.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(`*`objectname`*`)`. 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     1.313e+03 1.180e+02 6.811e-01      1.244e+00
r     1.998e-01 1.101e-02 6.359e-05      1.113e-04
sigma 2.538e-02 5.204e-03 3.004e-05      3.604e-05
2. Quantiles for each variable:
              2.5%       25%       50%       75%     97.5%
K     1.125e+03 1.235e+03 1.300e+03 1.374e+03 1.583e+03
r     1.776e-01 1.928e-01 1.999e-01 2.070e-01 2.213e-01
sigma 1.759e-02 2.169e-02 2.460e-02 2.814e-02 3.773e-02
```

Each of the two tables above has the properties of a matrix[5]. 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
0.19980128 0.01101439
```

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

```
> summary(zm)$quantile[1,c(1,5)]
    2.5%    97.5%
1124.539 1582.647
```

Exercise: Manipulating coda summaries Build a table that contains the mean, standard deviation, median and upper and lower 2.5% CI for parameter estimates 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. It looks like this:

```
[[1]]
Markov Chain Monte Carlo (MCMC) output:
Start = 60001
End = 60010
Thinning interval = 1
           K         r         sigma
```

[5]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.

```
     [1,] 1096.756 0.1914722 0.02889710
     [2,] 1196.326 0.2088859 0.03155777
     [3,] 1401.511 0.1804327 0.02553913
     [4,] 1471.539 0.1754886 0.03589013
     [5,] 1245.909 0.1567580 0.04248644
     [6,] 1134.738 0.2114307 0.04151478
     [7,] 1105.661 0.2303630 0.03141035
     [8,] 1108.569 0.2169765 0.03708956
     [9,] 1134.755 0.1964426 0.02660658
     [10,] 1161.750 0.2152418 0.03700475
     .
     .
     .
     as many rows as you have thinned iterations
```

So, the output of coda is a list of matrices (or tables if you prefer) where each matrix contains the output of the chains for each parameter to be estimated. Parameter values are stored in the columns of the matrix; values for one iteration of the chain are stored in each row. So, the example above is a case where we had 10 iterations of one chain. 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"
```

Exercise: 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.

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 coverted to a dataframe, 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: 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. (Hint: Look into using the `ecdf()` function.) Estimate the probability that iK falls between 1000 and 1300.

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 values to be estimated. 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 for 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 `mu` by changing your coda.samples statement to read:

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

Exercise: 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, replace `coda.samples` function with

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

If you run this and enter `zj` at the console, R will return the means of all the monitored variables[6]. 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
```

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, `ecdf()`. For example the following would estimate the probability that the parameter K is less that 900:

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

[6]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.

Exercise: making plots with JAGS objects For the logistic example:

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)

Like coda objects, JAGS objects have a list structure, but instead of each element of the list holding an array (i.e., matrix) for each chain, the JAGS objects holds an array for each quantity estimated. This is easier illustrated than explained. The JAGS object below[7] 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 `mu`:

```
> zj
$K
, , 1
        [,1]     [,2]     [,3]     [,4]     [,5]
[1,] 1424.628 1411.863 1307.185 1338.801 1351.346
, , 2
        [,1]     [,2]     [,3]     [,4]     [,5]
[1,] 1279.262 1326.353 1345.851 1243.561 1157.157
attr(,"class")
[1] "mcarray"
$mu
, , 1
            [,1]       [,2]       [,3]       [,4]       [,5]
 [1,] 0.17072948 0.19509308 0.19127273 0.19714752 0.19323022
 [2,] 0.16631829 0.19000444 0.18586162 0.19170919 0.18795213
 [3,] 0.16568811 0.18927749 0.18508861 0.19093228 0.18719812
 [4,] 0.16442777 0.18782360 0.18354257 0.18937848 0.18569010
 [5,] 0.15951244 0.18215340 0.17751305 0.18331862 0.17980879
 [6,] 0.15888227 0.18142645 0.17674003 0.18254172 0.17905478
 [7,] 0.14388420 0.16412508 0.15834225 0.16405139 0.16110928
 [8,] 0.13770852 0.15700098 0.15076670 0.15643772 0.15371995
 [9,] 0.12170217 0.13853649 0.13113209 0.13670435 0.13456802
[10,] 0.11628270 0.13228473 0.12448416 0.13002297 0.12808351
[11,] 0.09410068 0.10669615 0.09727399 0.10267593 0.10154226
```

[7] 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.

```
[12,] 0.09258827 0.10495147 0.09541876 0.10081136 0.09973263
[13,] 0.07822037 0.08837704 0.07779399 0.08309794 0.08254113
[14,] 0.06322230 0.07107567 0.05939621 0.06460761 0.06459562
[15,] 0.05288749 0.05915372 0.04671875 0.05186637 0.05222981
[16,] 0.03839356 0.04243390 0.02893938 0.03399757 0.03488752
, , 2
            [,1]       [,2]       [,3]       [,4]        [,5]
 [1,] 0.19328215 0.18103879 0.18031947 0.18834429 0.187960699
 [2,] 0.18768794 0.17599534 0.17537282 0.18272716 0.181909482
 [3,] 0.18688876 0.17527484 0.17466616 0.18192471 0.181045022
 [4,] 0.18529042 0.17383386 0.17325283 0.18031982 0.179316103
 [5,] 0.17905686 0.16821401 0.16774086 0.17406073 0.172573319
 [6,] 0.17825769 0.16749352 0.16703420 0.17325828 0.171708860
 [7,] 0.15923735 0.15034577 0.15021561 0.15416003 0.151134723
 [8,] 0.15140544 0.14328494 0.14329031 0.14629604 0.142663020
 [9,] 0.13110643 0.12498440 0.12534106 0.12591388 0.120705748
[10,] 0.12423353 0.11878816 0.11926375 0.11901283 0.113271397
[11,] 0.09610261 0.09342679 0.09438920 0.09076667 0.082842422
[12,] 0.09418460 0.09169760 0.09269321 0.08884080 0.080767719
[13,] 0.07596343 0.07527035 0.07658128 0.07054500 0.061058042
[14,] 0.05694309 0.05812261 0.05976269 0.05144675 0.040483906
[15,] 0.04383664 0.04630652 0.04817341 0.03828661 0.026306770
[16,] 0.02545564 0.02973517 0.03192015 0.01983031 0.006424201
attr(,"class")
[1] "mcarray"
$r
, , 1
          [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 0.1795519 0.2052704 0.2020950 0.2080242 0.2037864
, , 2
          [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 0.2044706 0.1911257 0.1902128 0.1995786 0.2000631
attr(,"class")
[1] "mcarray"
$sigma
, , 1
           [,1]       [,2]       [,3]       [,4]       [,5]
[1,] 0.03038826 0.02973461 0.03196986 0.02771297 0.02342979
, , 2
           [,1]       [,2]       [,3]       [,4]       [,5]
[1,] 0.02939191 0.02266891 0.01886645 0.01684712 0.02437535
attr(,"class")
[1] "mcarray"
```

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[8] is always = 1, the second

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

dimension = number of iterations and the third dimension = the number of the chain. For vectors, the first dimension of the JAGS object is the length of the vector, the second dimension is the number of iterations, and the third dimension is the number of the chain. An easy way to remember this is simply to enter `dim(jags.object)` at the console. Because the dimensions are named, there is no ambiguity about the structure of the object. So for example,

```
#dimensions of mu in the zj jags object:
dim(zj$mu)
#a vector containing all iterations of the second chain for K:
zj$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 `mu` across all iterations and all chains, you would use

```
mean(zj$mu[3,,])
```

Exercise: 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, which can be helpful for using convergence diagnostics (as described in th next section) if you haven't created a coda object directly using the `coda.samples` function. The syntax is

```
coda.object=as.mcmc.list(object.name$element.name).
```

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 my work I eventually create both types. Coda objects are somewhat better for producing tabular summaries of estimates and are required for checking convergence, but JAGS objects are somewhat 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. I generally start development of models using coda objects alone, and when I reach the final output stage, I produce 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). I 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.

8.1 Trace and density plots

There are three useful ways to plot the chains and the posterior densities. I am particularly fond of the latter two because they show more detail.

```
plot(coda.object)
xyplot(coda.object)
densityplot(coda.object)
```

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"(**?**). It requires at least 2 chains to work. For a complete treatment of how this works, enter `?gelman.diag` at the console and read the section on Theory. 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(coda.object)
```

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 syntax is

```
heidel.diag(coda.object)
```

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 10000 iterations. It returns and estimate of the number of iterations required for convergence for each of the parameters being estimated. Syntax is

```
raftery.diag(coda.object)
```

Exercise: Using the `zm.short` object your created above, increase `n.iter` in increments of 500 until you get convergence. For each increment:

1. Plot the chain and the posterior distributions of parameters using `xyplot` and `densityplot`.
2. Do Gelman-Rubin, Heidelberger and Welch, and Raftery diagnostics.

Discuss with you labmates how the plotting reveals convergence.

9 Monitoring deviance and calculating DIC

It is often a good idea to report the deviance of a model which is defined as $-2log\left[P\left(y|\theta\right)\right]$. 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, n.thin=1)
```

Later in the course we will learn about the Bayesian model selection statistic, the deviance information criterion (DIC). DIC 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 WinBUGS / OpenBUGS

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 (**?**). 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 Table 1.

Message	Interpretation
Unable to resolve parameter O[38,1:2] (one of its ancestors may be undefined)	May be due to NA in data or illegal value in variable on rhs of <- or ~.
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	You will get this with a Possion density if you give it continuous numbers as data. It will also occur if variables take on undefined values like log of negative.
Warning message: In readLines(file) : incomplete final line found on 'SS2.R'	Will occur when you don't have a hard return after the last } for the model
syntax error, unexpected '}', expecting $end	Occurs when there are mismatched parens
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 distribution or Poisson, divide by 0, log of negative, etc.
Error in setParameters(init.values[[i]], i) : Error in node sigma.s[1] Attempt to set value of non-variable node	You get this error when you have a variable in your init list that is not a stochastic node in the model, i.e., it is constant

12 Answers to exercises

Exercise: using `for` loops Write a code fragment to set vague normal priors [`dnorm(0,10e-6)`] for 5 regression coefficients stored in the vector `B`.

```
for(i in 1:5){
    B[i] ~ dnorm(0,.000001)
    }
```

Exercise: Understanding coda objects Modify your code to produce a coda object with 3 chains with 5 iterations each. Output

1. The estimate of σ for the third iteration from the second chain, `zm[[2]][2,3]`
2. All of the estimates of r from the first chain. `zm[[1]][,2]`

Exercise: Manipulating coda summaries

```
m=summary(zm)
mu_sd=m$stat[,1:2]  #make columns for mean and sd
q=m$quantile[,c(3,1,5)]  #make columns for median and CI
table=cbind(mu_sd,q) #make table
write.csv(file="/Users/Tom/Documents/Ecological Modeling Course/JAGS Primer/table_e
```

Exercise: 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. Plot the density of K. (This is very handy for producing publication quality graphs of posterior distributions.)

4. Estimate the probability that the parameter K exceeds 1600. (Hint: Look into using the `ecdf()` function.) Estimate the probability that it falls between 800 and 1200.

```
#exercises on manipulating coda objects converted to data frames
df=as.data.frame(rbind(zm[[1]],zm[[2]],zm[[3]]))
max(df$sigma)  #problem 1
mean(df$K[1:1000])  #problem 2, first part
nr=length(df$K)
mean(df$K[(nr-1000):nr]) #problem 2, second part
plot(density(df$K),main="",xlim=c(800,2000),xlab="K") #problem 3
1-ecdf(df$K)(1600)  #problem 4, first part
ecdf(df$K)(1200)-ecdf(df$K)(800)  #problem 4, second part.
```

Exercise: vectors in coda objects: Modify you code as described above 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? There are several ways this can be done, but the general idea is that you need to extract the rows of the coda object that contain the quantiles for μ, which can be tedious and error prone. For example, if you use rows in the summary table and add or subtract parameters to be estimated, then your row counts will be off. There are ways to use rownames, but a far better way to plot vectors is described in the section on JAGS objects.

Exercise: using JAGS objects to plot vectors For the logistic example:

1. Plot the data as points,
2. Overlay the median of the model predictions as a solid line
3. Overlay the 95% credible intervals as dashed lines.

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

```
    n.iter=50000, n.thin=1)
    b=summary(zj$K,mean)$stat b=summary(zj$mu,quantile,
    c(.025,.5,.975))$stat
    plot(pop.data$"Population Size", pop.data$"Growth Rate", xlab="N",
    ylab="Per capita growth rate")
    lines(pop.data$"Population Size",b[2,])
    lines(pop.data$"Population Size",b[1,],lty="dashed")
    lines(pop.data$"Population Size",b[3,],lty="dashed")
    plot(density(zj$K),xlab="K", main="", xlim=c(800,2500))
```

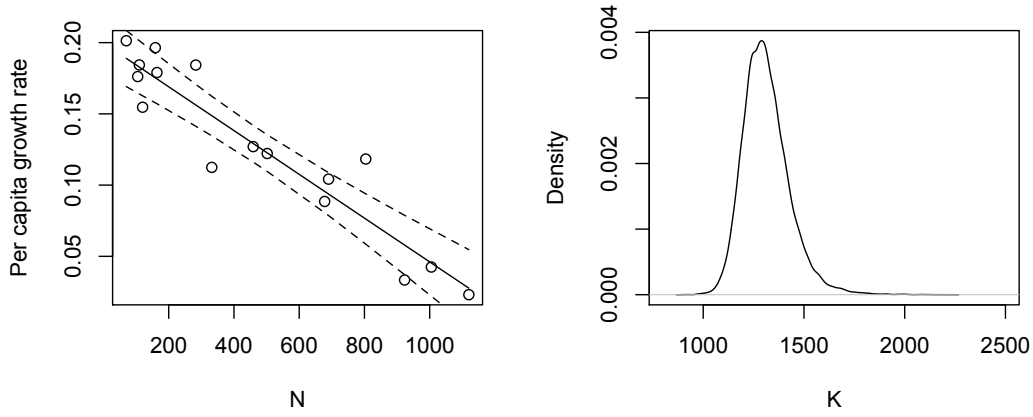

Figure 1: Median and 95% credible intervals for predicted growth rate and posterior density of K.

Exercise: 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 $\mu < 0$.

```
> median(zj$K[1,,2])
[1] 1275.208
> quantile(zj$mu[16,,],c(.025,.975))
```

2.5% 97.5%

```
-0.01539839  0.05925297
> ecdf(zj$mu[16,,])(0)
[1] 0.1096533
>
```

Literature Cited

Gelman, A. and D. B. Rubin, 1992. Inference from iterative simulation using multiple sequences. *Statistical Science* **7**:457–511.

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

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.

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

Plummer, M., 2011. JAGS version 3.0.0 user manual. *http://sourceforge.net/projects/mcmc-jags/files/Manuals/3.x/jags_user_manual.pdf* .

Raftery, A. and S. Lewis, 1995. The number of iterations, convergence diagnostics and generic Metropolis algorithms. Chapman and Hall, London, UK.

Index