Introduction to JAGS

Michael Dietze

```
## Loading required package: coda

## Linked to JAGS 4.3.0

## Loaded modules: basemod,bugs
```

Introduction

The aim of this activity is to provide a tutorial on JAGS (Just Another Gibbs Sampler), a statistical software package designed specifically to do Bayesian analyses of simple to intermediate complexity using Markov Chain Monte Carlo (MCMC) numerical simulation methods. JAGS is one of a set of generalized software tools for doing Bayesian computation, with BUGS, NIMBLE, and STAN being other popular options. These tools use very similar and simple scripting languages that focus on specifying data and process models. The great thing about these tools is that they keeps a lot of the mathematical and computational details "under the hood" so that you can focus on the structure of your model at a high level rather than being bogged down in the details. Also, since JAGS is designed just to do Bayesian MCMC computation it is very efficient at this, which is nice since Bayesian MCMC computations can be time consuming. In this tutorial we will work through the basics of using BUGS: writing and compiling a model, loading data, executing the model, and evaluating the numerical output of the model.

While tools like JAGS can be run by themselves, they are not really designed for managing or manipulating data, and thus it is common to use R packages to interface with these tools. In the example below we will use the rjags package to call JAGS. To be able to install rjags you will need to first install JAGS itself: http://mcmc-jags.sourceforge.net/

The reason for choosing JAGS in this course is that it has much better cross-platform support than the earlier BUGS language, which really only operates well in Windows. The STAN language is newer, also works across platforms, and is becoming increasingly popular.

Resources

JAGS itself provides a reference manual that is useful for looking up the syntax for distributions and functions, but isn't particularly useful for learning how to code. However, the JAGS syntax is virtually identical to the BUGS syntax and there are a much wider range of resources available for learning BUGS (books, examples, email list, etc.). The OpenBUGS website, http://www.openbugs.net/, is a great first stop for information – in particular the Documentation section contains examples, tutorials, and other useful information.

There's a large amount of info in both the JAGS and BUGS manuals but I wanted to point out a few sections that you'll find yourself coming back to repeatedly. One of these topics is the BUGS "Model Specification" which provides a lot of detail on how to write models in BUGS/JAGS, how data must be formatted, and the list of functions and distributions that BUGS/JAGS knows about. The naming convention for distributions in BUGS/JAGS is very similar to the convention in R, but there are a few cases where the parameterization is different (e.g. Weibull) so it is always good to check that the values you are passing to a BUGS/JAGS distribution are what you think they are. One very important example of this is that the Normal distribution in BUGS/JAGS, dnorm, is parameterized in terms of a mean and precision (1/variance) rather than a mean and standard deviation, which is the parameterization in R. To make life even more complicated, there are a small number of places where the JAGS syntax is slightly different from the BUGS version, so it's best to check the JAGS manual specifically for the definitions of any distributions and functions.

Within the BUGS Examples page you'll find three volumes of examples that provided written explanations of analyses with the BUGS code. Working through these is a great way to learn more about how BUGS works. When you first start analyzing your own data it is often easiest to start from an existing example and modify it to meet your needs rather than starting from scratch.

A Simple JAGS model

As our first example, let's consider the simple case of finding the mean of a Normal distribution with a known variance and Normal prior on the mean:

$$
\begin{aligned}
prior &= P(\mu) = N(\mu|\mu_0, \tau) \\
L &= P(X|\mu) = N(X|\mu, \sigma^2)
\end{aligned}
$$

As noted earlier JAGS parameterizes the Normal distribution in terms of a mean and precision (1/variance), rather than a mean and standard deviation. Therefore, let's redefine the prior precision as $T = 1/\tau$ and the data precision as $S = 1/\sigma^2$. Let's also switch from the conditional notation for PDFs to the tilda notation (~ is read as "is distributed as"). Given this we can rewrite the above model as:

$$
\begin{aligned}
\mu &\sim N(\mu_0, T) \\
X &\sim N(\mu, S)
\end{aligned}
$$

This problem has an exact analytical solution so we can compare the numerical results to the analytical one (Dietze 2017, Eqn 5.7).

$$P(\mu|X) \sim N\left(\frac{S}{S+T}X + \frac{T}{S+T}\mu_0, S+T\right)$$

In JAGS the specification of any model begins with the word "model" and then encapsulates the rest of the model code in curly brackets:

```
model {
## model goes here ##
}
```

When writing models in JAGS we have to specify the data model, process model, and parameter model using the tilda notation. However, we don't need to specify explicitly the connections between them — JAGS figures this out based on the conditional probabilities involved. Deterministic calculations (e.g. process model) always make use of an arrow (<-) for assignment (similar to R syntax). Assignment of random variables is always done with a tilde (~). **The equal sign (=) cannot be used for either process or data models. Furthermore, deterministic calculations and distributions can not be combined in the same line of code.** For example, a regression model where we're trying to predict data y based on observation x might include

```
mu <- b0 + b1* x # process model
y ~ dnorm(mu,tau) # data model
```

but in JAGS the same model can **NOT** be expressed as

```
y ~ dnorm(b0 + b1* x, tau)
```

Putting this all together, to specify our first model of a normal likelihood and normal prior, the JAGS code is:

```
model {
mu ~ dnorm(mu0,T) # prior on the mean
X ~ dnorm(mu,S) # data model
}
```

The first line of this model specifies the prior and says that "mu" is a random variable (~) that is Normally distributed (dnorm) with an expected value of "mu0" and a precision of T. The second line specifies the likelihood and says that there is a single data point "X" that is a random variable that has a Normal distribution with expected value "mu" and precision "S".

Hopefully the above JAGS code seems pretty clear. However, there are a few quirks to how JAGS evaluates a model, some of which are just idiosyncrasies (like no =) while others are deeply embedded in the logic but often misleading to newcomers. One of the more disorienting features is that, because JAGS isn't evaluating the code sequentially, the order of the code doesn't matter – the above model is equivalent to

```
model {
X ~ dnorm(mu,S) # data model
mu ~ dnorm(mu0,T) # prior on the mean
}
```

If you're used to R the above seems 'wrong' because you're using mu in the data model before you've 'defined' it. But JAGS models are not a set of sequential instructions, rather they are the definition of a problem which JAGS will parse into a graph and then analyze to how to best sample from. Furthermore, even if the code were evaluated sequentially, MCMC is a cyclic algorithm and thus, after initial conditions are provided, it doesn't really matter what order thing are in.

The other major "quirk" of JAGS is that what JAGS does with a model can depend a LOT on what's passed in as knowns! Understanding how this works will help you considerably when writing and debug models. Key to this is to realize that *no variables are 'unknowns' that JAGS is trying to solve for in the algebraic sense.* Rather, some variables are constants or derived (calculated) quantities while others are **random variables**. Random variables *HAVE to have a distribution.* If we look at our simple model, mu0, S, and T don't have distributions, so they have to be provided as inputs. If they are not, JAGS will throw an error. Next, mu is clearly a random variable as it shows up on the left-hand side of the prior and on the right-hand side of the likelihood. mu *cannot* be specified as an input – this likewise will throw an error. But the interesting part is that, depending on context, X could either be known or random and thus what JAGS does with this model depends a lot of whether X is in the list of inputs. If X is NOT specified, then the JAGS reads the above code as:

1. Draw a random mu from the prior distribution
2. Conditional on that value, draw a random X

However, if X IS specified then JAGS will generate samples of $\mu|X$ – in other words it will estimate the posterior of mu. This polymorphic behavior leads to another quirk – all distributions are in the "d" form even if you're using them to generate random numbers (i.e. there is no rnorm in JAGS).

Because of how JAGS views random variables, no variable can show up on the left-hand side more than once – it's can't be both calculated and random, it can't be calculated more than once, and it can't have multiple priors or data models. Most of the time this is fine, but what catches many people is that it means you **can't reuse temporary variables**, because JAGS will view this as an attempt to redefine them.

Evaluating a JAGS model

OK, so how do we now use JAGS to fit our model to data? To start with, we need to realize that JAGS code is not identical to R code, so if we write it in R as is, R will throw an error. Therefore, we either need to write this code in a separate file or treat it as text within R. In general, I prefer the second option because I feel it makes coding and debugging simpler. Here I define the above JAGS model as a text string in R:

```
NormalMean = "
model {
mu ~ dnorm(mu0,T) # prior on the mean
X ~ dnorm(mu,S) # data model
}
"
```

To pass this model to JAGS we'll use the `rjags` function `jags.model`, which has required arguments of the model and the data, and optional arguments for specifying initial conditions and the number of chains (I encourage you to read the man page for this function!). So before running this function let's first set up these other inputs

Data as lists

In a nutshell, pass data to JAGS using R's *list* data type. JAGS models don't have a function interface, and as noted before the code can be polymorphic, so the order that data is specified is meaningless. However, **the NAMES of the variables in the list and in the code have to match EXACTLY (this is a common bug)**. The other thing to remember is that all constants in the code that are not defined inline (which is generally discouraged), need to be passed in with the data list. In the example above our data is X and our constants are mu0, S, and T. These can be defined in a list as follows

```
sigma = 5  ## prior standard deviation
data = list(X = 42, ## data
            mu0 = 53, ## prior mean
            S = 1/sigma^2, ## prior precision
            T = 1/185) ## data precision
```

In this example I demonstrated converting from standard deviation to precision because, at least initially, most people are not accustomed to thinking in terms of precisions. The exact numeric values used for the data and priors are not important here and are just for illustration – how priors are chosen is discussed elsewhere.

Initial conditions: lists, lists of lists, functions

The next step is to specify the initial conditions. In JAGS this is optional because if initial conditions are not specified then the code will draw them randomly from the priors. If you use informative priors this is not usually a problem. If you use uninformative priors, however, the initial parameter value can start far from the center of the distribution and take a long time to converge, just like in maximum likelihood optimization. Initial conditions are also specified using a list, but in this case, we want to name the parameter variable names instead of the data variable names. A nice feature in JAGS is that we only have to specify some of the variable names, and those that we don't specify will be initialized based on the priors. Therefore, we can often get away with only initializing a subset of variables.

As an example, if we wanted to specify an initial condition for μ we would do this with a list, same as we did with the data.

```
inits = list(mu=50)
```

Unlike priors, which strictly cannot be estimated from data, it is perfectly fair (and in most cases encouraged) to use the available data to construct the initial guess at model parameters. Good initial conditions improve convergence by reducing the time spent exploring low probability parameter combinations. In more complex models, bad choices of initial conditions can cause the model to blow up immediately. Indeed, if they start getting really funky results out of MCMC's many people's first reaction is to change the priors, which is generally the wrong thing to do (unless the priors really did have a bug in them). A better place to start is to check the initial conditions, especially if parameters are not being specified.

In JAGS we will commonly want to run multiple independent MCMC chains, as this better allows us to assess whether all chains have converged to the same part of parameter space. As with numerical optimization, it is best practice to start each chain from different initial conditions, but the above example would give all the chains the same initial mu. To specify multiple initial conditions, we need to construct a list of lists, with each sub-list providing one initial condition list for each chain. In this example we'll run three chains starting at three different values for . We specify this in R as:

```
inits <- list()
inits[[1]] <- list(mu = 40)
inits[[2]] <- list(mu = 45)
inits[[3]] <- list(mu = 50)
```

Finally, it is also possible to pass a function that returns a list of initial values

Running JAGS

Now that we've got the data and initial conditions set up, we can call JAGS. Running the model in JAGS is broken into two steps. First, *jags.model* is called to establish the connection to JAGS, compile the model, and run through an initial number of adaptation steps (n.adapt is an option argument that defaults to 1000). Second, once the model is compiled then *coda.samples* is called to sample from the posterior. Separating this into two steps is actually advantageous because it makes it easier to monitor the progress of the MCMC and to add/remove variables from what's being sampled.

The call to jags.model is fairly straightforward

```
j.model   <- jags.model (file = textConnection(NormalMean),
                             data = data,
                             inits = inits,
                             n.chains = 3)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 1
##    Unobserved stochastic nodes: 1
##    Total graph size: 5
##
## Initializing model
```

The *textConnection* function is part of the normal R base and allows a text string (in this case our model) to be passed to a function that's expecting an external file. n.chains sets the number of chains. We'll use three for this example; 3-5 is typical.

If you have any bugs in your model, this is the step that will most likely throw an error.

Once the model is correctly compiled and initialized you can call *coda.samples* to sample from the model. Besides the initialized model object, the other arguments to the function are *variable.names*, a vector of variables you want stored, and *n.init* the number of iterations.

```
jags.out   <- coda.samples (model = j.model,
                            variable.names = c("mu"),
                                n.iter = 5000)
```

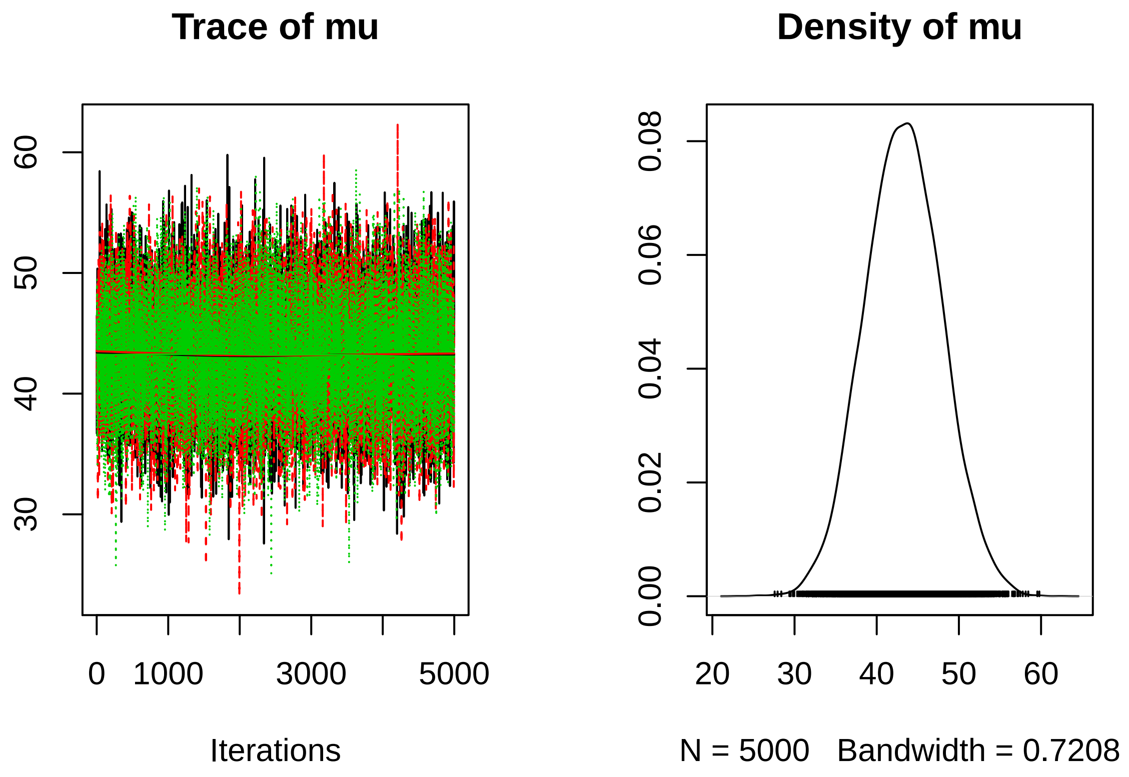
Evaluating MCMC outputs

The output from JAGS is returned in a format called an mcmc.list. The R library *coda* provides a whole set of functions for assessing and visualizing outputs in this format. The coda library can also be used to evaluate saved outputs from other Bayesian software, for example if you decide to use the BUGS stand-alone graphical interface.

MCMC convergence

When running an MCMC analysis the first question to ask with any output is “has it converged yet?” This usually starts with a visual assessment of the output:

```
plot(jags.out)
```



For every variable you are tracking this will spit out two plots, a trace plot and a density plot. The traceplot looks like a time-series with the parameter value on the Y-axis and the iterations on the X-axis. Each chain is plotted in a different color. In a model that has converged these chains will be overlapping and will bounce around at random, preferably looking like white noise but sometimes showing longer term trends.

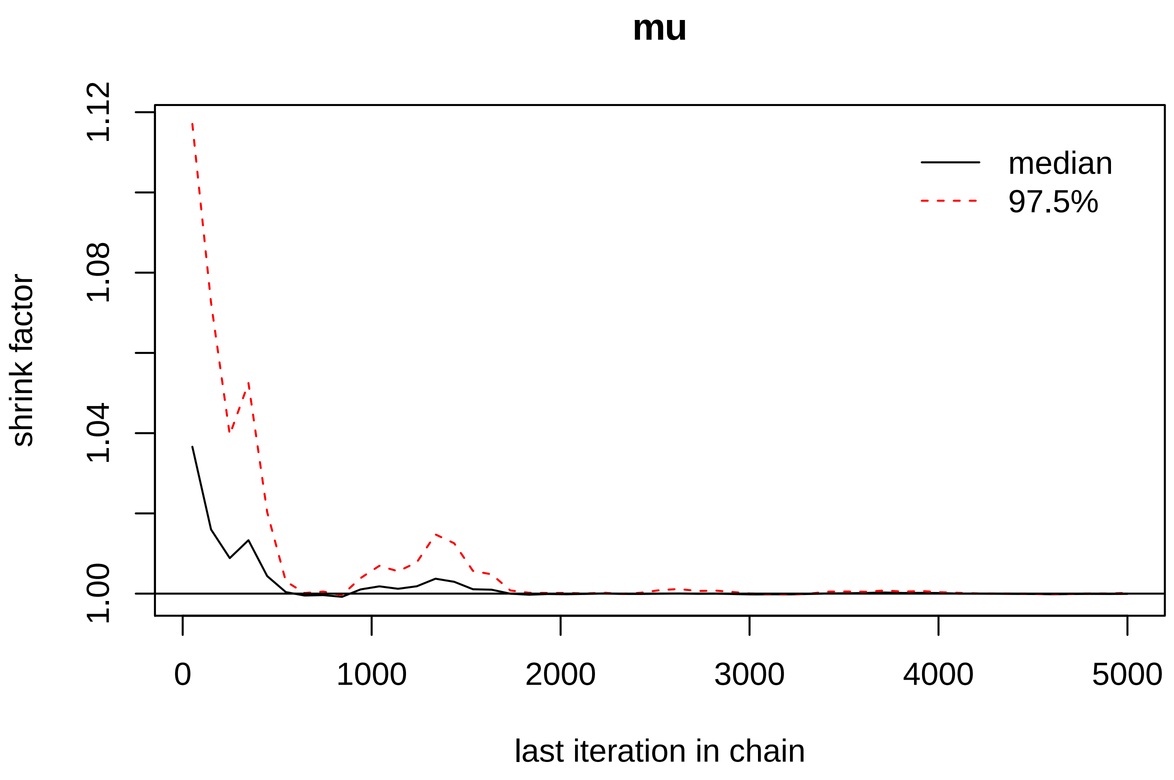
To help make the assessment of convergence more objective, coda offers a number of diagnostics, with the most common being the Brooks-Gelman-Rubin (BGR) statistic. GBR requires that you have run multiple chains because it compares the among chain variance to the within change variance. If a model has converged, then these two should be identical and the GBR should be 1. There's no hard-and-fast rule for the threshold that this statistic needs to hit but in general values less than 1.01 are excellent, 1.05 is good, and >1.1 is generally held to be not yet converged.

```
gelman.diag(jags.out)
```

```
## Potential scale reduction factors:
##
##    Point est. Upper C.I.
## mu          1          1
```

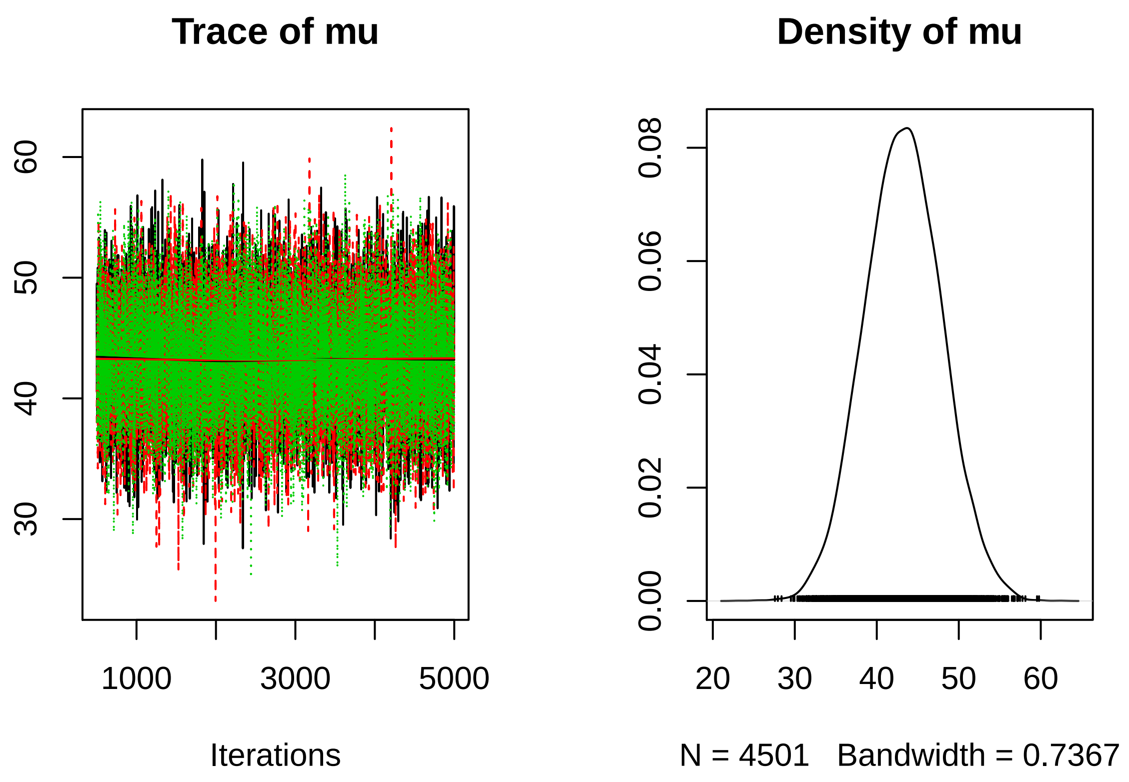
In addition to having checked for overall convergence, it is important to remove any samples from before convergence occurred. Determining when this happened is done both visually and by plotting the GBR statistic versus sample

```
GBR <- gelman.plot(jags.out)
```



The point up to where the GBR drops below 1.05 is termed the "burn in" period and should be discarded. For example, if I determine the burn in to be 500 steps, I could remove the first 500 as:

```r
burnin = 500                                ## determine convergence
jags.burn <- window(jags.out,start=burnin)  ## remove burn-in
plot(jags.burn)                             ## check diagnostics post burn-in
```



After discarding burn-in, you should work with the trimmed object for subsequent analyses

Updating the MCMC

If the MCMC hasn't converged by the end, if you need additional samples, or if you want to add or remove variables from your output, you can take additional samples by simply calling coda.samples again.

```
jags.out2 <- coda.samples(j.model,variable.names = c("mu"),10000)
```

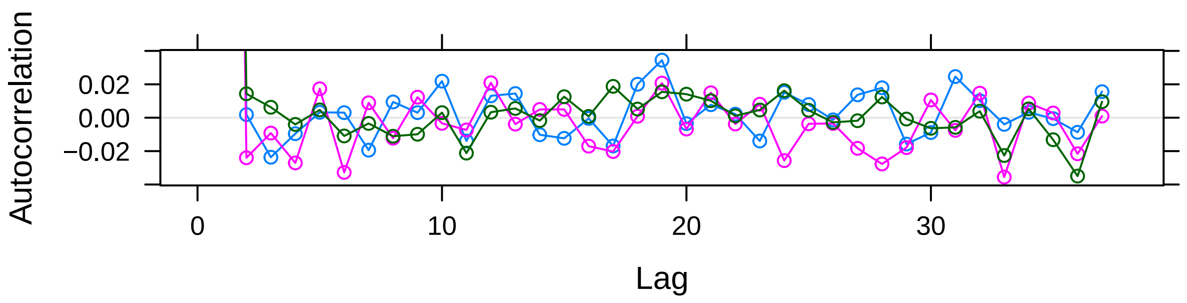
Note: the new samples are written to a new variable, not automatically added to the old. If you're working with a slow model where samples are precious, you'll want to manually splice the old and new samples together.

Sample size

The previous section raises an obvious question, "How many samples is enough?" Unfortunately, there's no single right answer to this question because it depends on how well your model is mixing (i.e. how independent the samples are from one step to the next) and what output statistics you are interested in.

To address the first question, the independence of samples is largely a question of autocorrelation. Coda provides a simple autocorrelation plot

```r
acfplot(jags.burn)
```



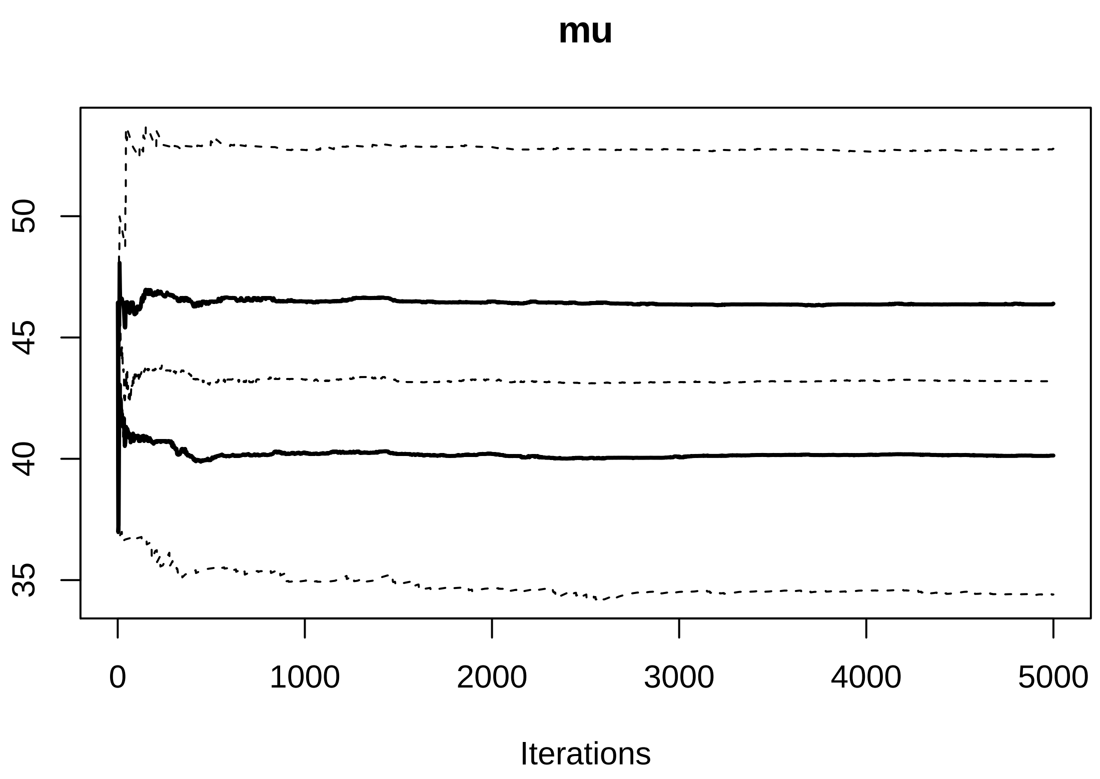
Autocorrelation is always 1 by definition at lag 0 and then typically decays toward 0 roughly exponentially. The faster the decay the greater the independence. One way of approximating the number of independent samples you have is to divide the actual number of samples by the lag distance at which samples are approximately independent. A slightly more analytically rigorous way of achieving the same thing is to perform an **effective sample size calculation**

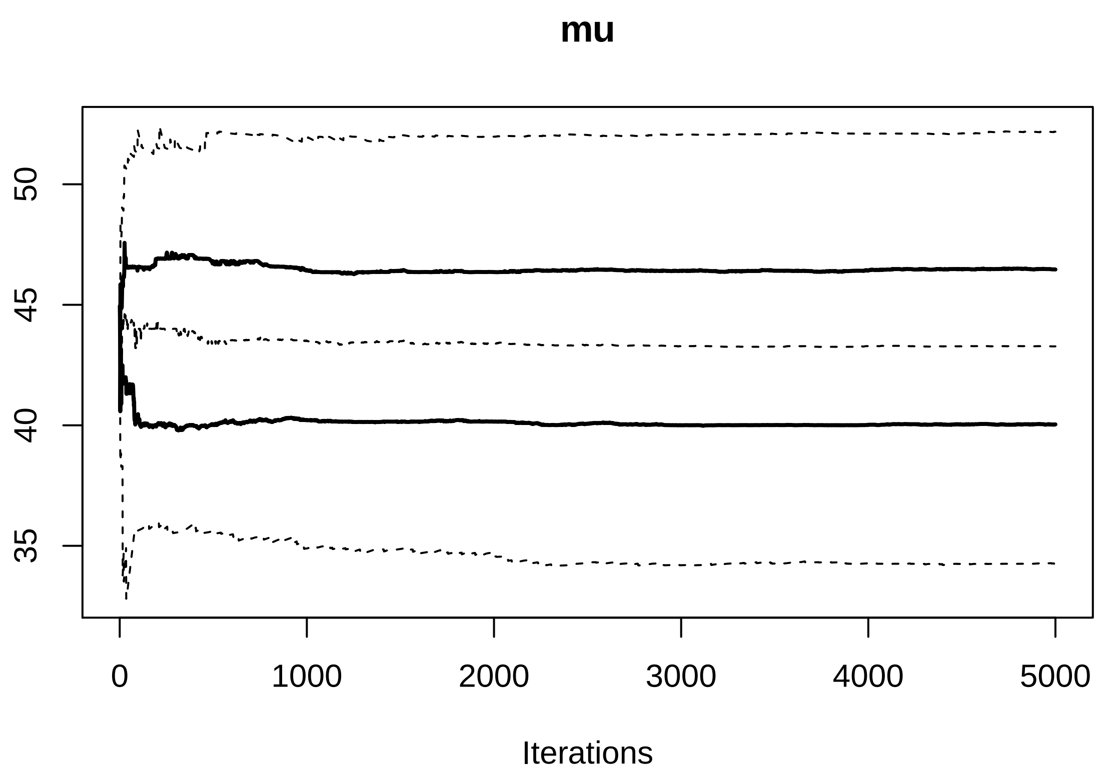
```r
effectiveSize(jags.burn)
```

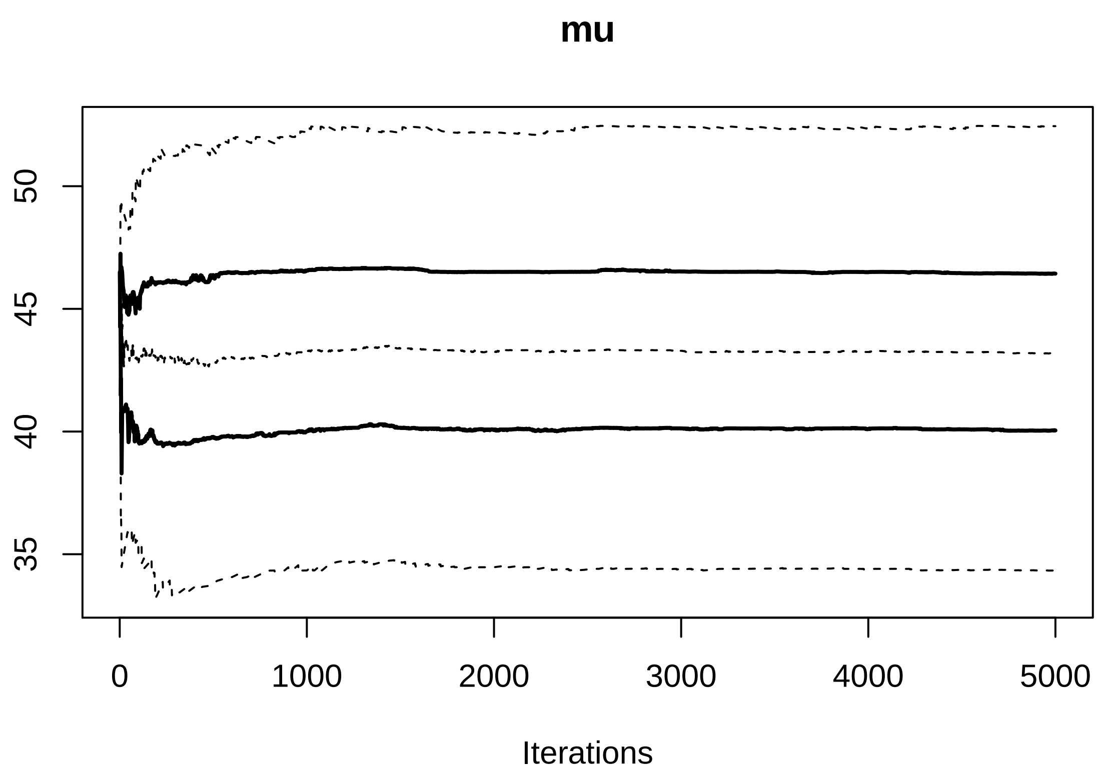
```
##       mu
## 14256.64
```

For the second question, the number of samples required increases with the tails of the posterior distribution. For example, if you only care about the posterior mean or median, then you can get a stable estimate with an effective sample size of only a few hundred points. The standard deviation and interquartile range are more extreme and thus require more samples to estimate, while the 95% CI requires even more – **a common rule of thumb would be an effective size of 5000 samples**. Recall that the CI is determined by the most exteme values, so at n=5000 it is only the 125 largest and 125 smallest values that determine the value. If you need to work with even larger CI or even lower probability events you'll require more samples. This phenomenon can be shown graphically (and diagnosed) by looking at plots of how different statistics change as a function of sample size:

```
cumuplot(jags.out,probs=c(0.025,0.25,0.5,0.75,0.975))
```



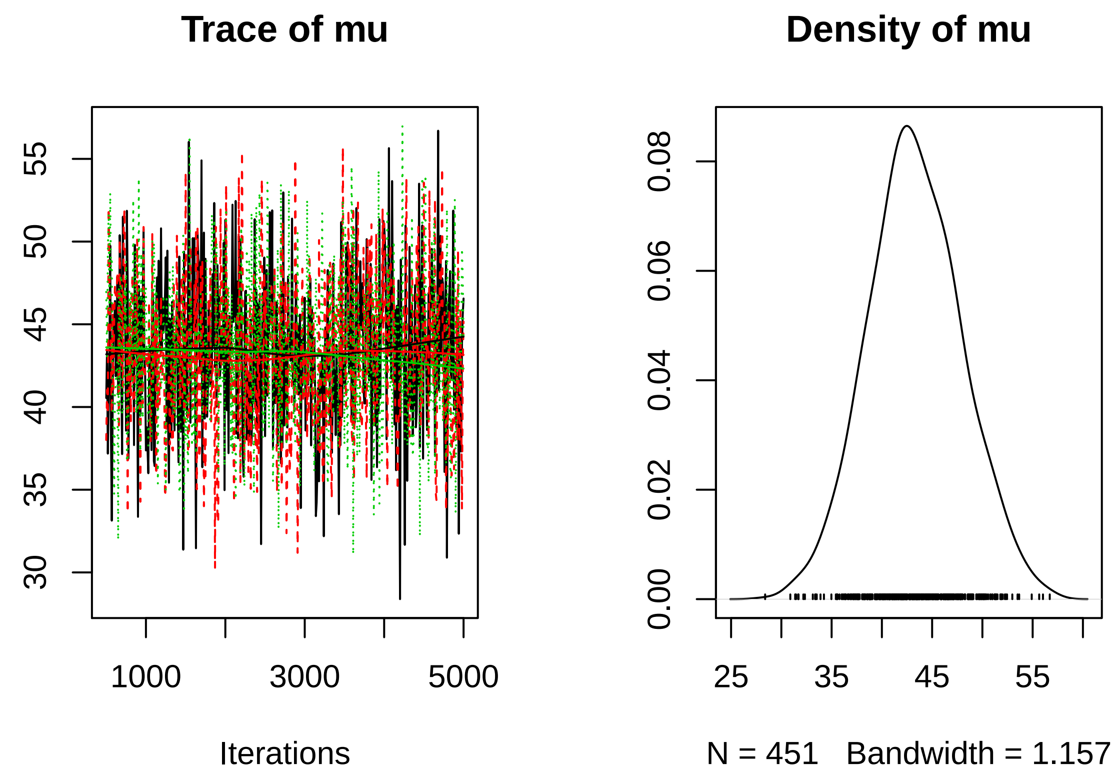

mu
50
45
40
35
0
1000
2000
3000
4000
5000
Iterations



Thinning

Older texts will advise that you thin your posterior samples to account for autocorrelation. Current thinking is that this is unnecessary unless you need to reduce the size of the output you save. Intuitively, setting "thin" to 10 retains only every 10th sample for further analysis, etc.

```
jags.thin = window(jags.burn,thin=10)
plot(jags.thin)
```



MCMC Statistics

Once your model has converged, and has enough samples after burn-in, you can calculate any summary statistics you want to describe your posterior output

```
summary(jags.out)
```

```
##
## Iterations = 1:5000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean             SD       Naive SE Time-series SE
##       43.24534        4.65270        0.03799        0.03702
##
## 2. Quantiles for each variable:
##
##  2.5%   25%   50%   75% 97.5%
## 34.29 40.08 43.24 46.43 52.45
```

The first part of this table is the mean, standard deviation, and two different standard errors. Knowing how to interpret the last three of this is important. The first, SD, describes the spread of the posterior distribution – in this case it is the **uncertainty about the parameter** mu (which, in other contexts, we usually refer to as the standard error). The SD doesn't decrease as you run the MCMC longer, it just converges on the value determined by the number and distribution of data points. By contrast, the SE declines asymptotically with MCMC length. In this context, this makes it an **indicator of the numerical precision of your results** – the longer the MCMC the higher the numerical precision of the posterior statistics. Between the two alternative SE values you should always use the Time-series SE, which has been corrected for autocorrelation, rather than the Naive SE, which has not.

The second part of the summary table is the sample quantiles for the posterior – the default is the 95% CI, interquartile range, and median.

In addition, there are a number of stats/diagnostics that you'll want to perform for multivariate models to assess the correlations among parameters. Watch for the *corr* statistic and *pairs* diagnostics in later examples

Finally, if you need to work with the MCMC output itself, either to perform additional analyses (e.g. uncertainty analysis, prediction) or generate additional statistics/diagnostics, the coda mcmc.list format is a bit of a pain to work with. I find the most convenient format is a matrix:

```
out <- as.matrix(jags.burn)
```

Case Study: Forest Stand Characteristics

For the next few examples we'll be using a dataset on the diameters of loblolly pine trees at the Duke FACE experiment. In this example we'll just be looking at the diameter data in order to characterize the stand itself. Let's begin by expanding the model we specified above to account for a large dataset (rather than a single observation), in order to estimate the mean stem diameter at the site. As a first step let's still assume that the variance is known. Our data set has 297 values, so when specifying the model in JAGS we'll need to loop over each value to calculate the likelihood of each data point and use the vector index notation, [i] , to specify which value we're computing.

```
NormalMeanN <- "
model {
  mu ~ dnorm(mu0,T) # prior on the mean
  for(i in 1:N){
    X[i] ~ dnorm(mu,S) # data model
  }
}
"
```

The data for fiting this model are:

```
data = list(
  N = 297,
  mu0=20,
  T=0.01,
  S = 1/27,
  X = c(20.9, 13.6, 15.7, 6.3, 2.7, 25.6, 4, 20.9, 7.8, 27.1, 25.2, 19, 17.8, 22.8, 12.5, 21.1, 22, 22.
  )
```

Activity Task 1

Run the unknown mean/fixed variance model to estimate mean tree diameter. Include the following in your results:

- Table of summary statistics for the posterior estimate of the mean
- Graph of parameter density
- Plot of MCMC “history”
- Length of your MCMC (i.e. the total number of “updates”), the number of chains, the burnin values you used, the effective sample size, and any graphs/statistics/rationale you used to justify those settings

Activity Task 2

Modify the model to account for the uncertainty in the variance. This only requires adding one line — a prior on S outside the loop.

Run the unknown mean/unknown variance model to simultaneously estimate both the mean tree diameter and the standard deviation. Include the following in your results:

- Explanation of your choice of prior on the precision
- Table of summary statistics for the posterior mean and standard deviation
- Graph of parameter densities
- Plot of MCMC “history”
- Length of your MCMC (i.e. the total number of “updates”), the number of chains, burnin, effective sample size, and any graphs/statistics/rationale you used to justify those settings
- Also describe any changes in the distribution of the mean (shape, location, tails) compared to Task 1.