Applied Bayesian Modeling
A brief JAGS and R2jags tutorial

Johannes Karreth
University of Colorado at Boulder

johannes.karreth@colorado.edu

Supplementary **R** and JAGS code:
http://spot.colorado.edu/~joka5204/bayes2012.htm

ICPSR Summer Program 2012
Last updated on July 28, 2012

1 Bayesian modeling for Mac users

As a Mac user, you have the following (and more) options to work through the materials presented in this course:

1. Use WinBUGS on Windows 7 on the computers in the Helen Newberry building.
2. Install a virtual machine or emulator on your Mac and use WinBUGS on the Mac.
3. Use JAGS—a complete BUGS engine for Unix—on your Mac (with very minor code adjustments).

This tutorial focuses on the third option, since it gives you the most mileage if you want to keep working with Bayesian models. If you are interested in the second option, but do not plan on purchasing or using a full virtual machine, it is also possible to use WinBUGS via WINE. Instructions for this option can be found at several sources. Two I have found to be useful are http://idiom.ucsd.edu/~rlevy/winbugsonmacosx.pdf; or http://daniel-stegmueller.com/RWinBugsMac.html. Another program you might want to check out is the **R** package MCMCpack: see http://mcmcpack.wustl.edu/.

2 What are JAGS, R2jags, ...?

JAGS is **J**ust **A**nother **G**ibbs **S**ampler that was mainly written by Martyn Plummer in order to provide a BUGS engine for Unix. More information can be found in the excellent JAGS manual at http://sourceforge.net/projects/mcmc-jags/ .

R2jags is an **R** package that allows running JAGS models from within **R**. It was written by Andrew Gelman et al. with the purpose of providing identical functionality to the R2WinBUGS package on Windows machines. Almost all examples in Gelman and Hill's *Data Analysis Using Regression and Multilevel/Hierarchical Models* can thus be run equivalently in JAGS, using R2jags.

rjags is another **R** package that allows running JAGS models from within **R**. R2jags depends on it. Simon Jackman's *Bayesian Analysis for the Social Sciences* provides many examples using rjags, and so does John Kruschke's *Doing Bayesian Data Analysis*.

In this tutorial, I focus on the use of R2jags, as well as using JAGS directly from the Terminal.

3 Installing JAGS and R2jags on Mac OS X 10.6 and above[1]

1. Install the most recent **R** version from the CRAN website: http://cran.r-project.org/bin/macosx. If you are already running a different version of **R**, you can (but need not) uninstall it by typing
   `> rm -rf /Library/Frameworks/R.framework /Applications/R.app`
   in the Terminal.

2. Install the Tcl/Tk libraries (`tcltk-8.5.5-x11.dmg`) and GNU Fortran (`gfortran-4.2.3.dmg`) from the CRAN tools directory: http://cran.r-project.org/bin/macosx/tools.

3. Install JAGS version 3.1.0 (`JAGSdist-3.1.0.dmg`) from Martyn Plummer's repository: http://sourceforge.net/projects/mcmc-jags/files/JAGS/3.x/Mac%20OS%20X/. Start the Terminal and type
   `> jags`
   to see if JAGS 3.1.0 is installed.

4. Install the packages R2jags, coda, R2WinBUGS, lattice, and (lastly) rjags from within **R**, via the Package Installer or by using
   `> install.packages("packagename")`

5. Consider using a scientific text editor for writing **R** and JAGS code. A list of good editors for Mac OS X is here: http://mac.appstorm.net/roundups/office-roundups/top-10-mac-text-editors/. RStudio is a very neat integrated environment for running **R** on a Mac (and other platforms), and I recommend using it: http://www.rstudio.org.

[1] Note for users of Mac OS X 10.5 (Leopard): Due to a particular behavior of the JAGS installer on Leopard, the JAGS files that rjags requires to run are not located where rjags is looking for them. See http://martynplummer.wordpress.com/2011/11/04/rjags-3-for-mac-os-x/#comments If you would like to use R2jags or rjags on Mac OS X 10.5, you need to manually relocate these files from `/usr` to `/usr/local`. See me if you would like help with this.

4 Fitting Bayesian models in JAGS

4.1 Using R2jags

Just like R2WinBUGS[2], the purpose of R2jags is to allow running JAGS models from within **R**, and to analyze convergence and perform other diagnostics right within **R**. A typical sequence of using R2jags could look like this:

- `> library(R2jags)`

  calls:

  ```
  > library(coda)
  > library(lattice)
  > library(R2WinBUGS)
  > library(rjags)
  ```

Preparing the data and model

- Read the data in from the car package:

  ```
  > library(car)
  > data(Angell)
  > angell.1 <- Angell[, -4] ## take off the fourth
  > ## column (remember, the order is (row, column))
  ```

- Save the model as "angell.model.jags" in your working directory. (Do not run this model from within **R**.) You can set your working directory in the **R** preferences, or via:

  ```
  > setwd("/Users/johanneskarreth/R/Bayes/angell")
  ```

  The model looks just like the BUGS models shown in class:

  ```
  model {
  for(i in 1:N){
  moral[i]~dnorm(mu[i], tau)
  mu[i]<-alpha + beta1*hetero[i] + beta2*mobility[i]
  }

  alpha~dnorm(0, .01)
  beta1~dunif(-100000,100000)
  beta2~dunif(-100000,100000)
  tau~dgamma(.01,.01)
  }
  ```

[2] See Appendix C in Gelman and Hill (2007) or their online appendix http://www.stat.columbia.edu/~gelman/bugsR/runningbugs.html for more info on how to run R2WinBUGS and **R**.

- Now define the vectors of the data matrix for JAGS:

```
> moral <- angell.1$moral
> hetero <- angell.1$hetero
> mobility <- angell.1$mobility
> N <- length(angell.1$moral)
```

- Read in the Angell data for JAGS

```
> angell.data  <- list("moral", "hetero", "mobility", "N")
```

- Define the parameters you're interested in:

```
> angell.params <- c("alpha", "beta1", "beta2")
```

- Define the starting values for JAGS

```
> angell.inits <- function(){
+         list("alpha"=c(20), "beta1"=c(-0.1), "beta2" =c(-.02))
+   }
```

  Alternatively, you can specify separate starting values for each chain:

```
> inits1 <- list("alpha"=0, "beta1"=0, "beta2"=0)
> inits2 <- list("alpha"=1, "beta1"=1, "beta2"=1)
> angell.inits <- list(inits1, inits2)
```

- Before using R2jags the first time, you might need to set a seed. To do this, type

```
> set.seed(123)
```

  directly in **R**. (You can choose any not too big number here.)

Fitting the model

- Fit the model in JAGS:

```
> angellfit <- jags(data=angell.data, inits=angell.inits,
+   angell.params, n.chains=2, n.iter=9000, n.burnin=1000,
+   model.file="angell.model.jags")
```

- Update your model if necessary - e.g. if there is no/little convergence:

```
> angellfit.upd <- update(angellfit, n.iter=1000)
> angellfit.upd <- autojags(angellfit)
```

  This function will auto-update until convergence - I believe it uses the Gelman-Rubin statistic to assess convergence.

Diagnostics

One of the neat things about using R2jags is that it offers seamless, and therefore quick, access to convergence diagnostics after fitting a model. See for yourself:

- ```
  > print(angellfit)
  > plot(angellfit)
  ```

- ```
  > traceplot(angellfit)
  ```

- If you want to print and save the plot, you can use the following set of commands:

  ```
  > pdf("angell.trace.pdf")
  ```

  ... defines that the plot will be saved as a PDF file with the name "angell.trace.pdf" in your working directory.

  ```
  > traceplot(angellfit)
  ```

  creates the plot in the background (you will not see it).

  ```
  > dev.off()
  ```

  finishes the printing process and creates the PDF file of the plot. If successful, **R** will display the message `"null device 1"`.

More diagnostics are available when you convert your model output into an MCMC object. You can generate an MCMC object for analysis with this command:

```
> angellfit.mcmc <- as.mcmc(angellfit)
> summary(angellfit.mcmc)
```

With an MCMC object, you can use a variety of commands for diagnostics and presentation. First, using CODA:

- Plot:

  ```
  > xyplot(angellfit.mcmc)
  ```

  Maybe better display via (you can use other Lattice options here as well):

  ```
  > xyplot(angellfit.mcmc, layout=c(2,6), aspect="fill")
  ```

- Density plot:

  ```
  > densityplot(angellfit.mcmc)
  > densityplot(angellfit.mcmc, layout=c(2,6), aspect="fill")
  ```

- Trace- and density in one plot, print directly to your working directory:

```
> pdf("angellfit.mcmc.plot.pdf")
> plot(angellfit.mcmc)
> dev.off()
```

- Autocorrelation plot, print directly to your working directory:

```
> pdf("angellfit.mcmc.autocor\textsf{\textbf{R}}.pdf")
> autocor\textsf{\textbf{R}}.plot(angellfit.mcmc)
> dev.off()
```

- Other diagnostics using CODA:

```
> gelman.plot(angellfit.mcmc)
> geweke.diag(angellfit.mcmc)
> geweke.plot(angellfit.mcmc)
> raftery.diag(angellfit.mcmc)
> heidel.diag(angellfit.mcmc)
```

A very convenient function to analyze numerical representations of diagnostics in one sweep is the superdiag package written by Tsung-han Tsai and Jeff Gill.

- First, install the package:

```
> install.packages("superdiag")
> library(superdiag)
```

- Next, all you need to do is:

```
> superdiag(angellfit.mcmc, burnin = 1000)
```

A convenient way to obtain graphical diagnostics and results is using the mcmcplots package:

- First, install the package:

```
> install.packages("mcmcplots")
> library(mcmcplots)
```

- Some commands for plots:

```
> denplot(angellfit.mcmc))
> denplot(angellfit.mcmc, parms = c("alpha", "beta1", "beta2"))
> traplot(angellfit.mcmc, parms = c("alpha", "beta1", "beta2"))
```

  As always, check the help files for options to customize these plots if you are so inclined.

- Or, for quick diagnostics, you can produce html files with trace, density, and autocorrelation plots all on one page. The files will be displayed in your default internet browse**R**.

```
> mcmcplot(angellfit.mcmc)
```

- If you want to produce a coefficient dot plot with credible intervals, use `caterplot`:

```
> caterplot(angellfit.mcmc)
> caterplot(angellfit.mcmc, parms = c("alpha", "beta1", "beta2"),
+   labels = c("Intercept", "Heterogeneity", "Mobility"))
```

If you want to write out the output from your model, you could use the function jags2:

```
> angell.params <- c("alpha", "beta1", "beta2", "deviance")
> angellfit <- jags2(data=angell.data, inits=angell.inits,
+   angell.params, n.iter=5000, model.file="angell.model.jags",
+   clearWD = FALSE)
```

- This will create the following files in your working directory (if `clearWD` is set to `FALSE`):

```
/Users/johanneskarreth/R/CODAchain1.txt
/Users/johanneskarreth/R/CODAchain2.txt
/Users/johanneskarreth/R/CODAindex.txt
/Users/johanneskarreth/R/jagsdata.txt
/Users/johanneskarreth/R/jagsinits1.txt
/Users/johanneskarreth/R/jagsinits2.txt
/Users/johanneskarreth/R/jagsscript.txt
```

- ... which can then be analyzed using BOA

```
> library(boa)
> boa.menu()
```

- or CODA.

```
> library(coda)
> codamenu()
```

- In both cases (using boa or coda), remember to rename the index file's extension to .ind, and the chain files to .out. See section **??** for code on how to read these files into R via the command line.

Good resources to find more information about R2jags:

- Yu-Sung Su (Columbia) co-wrote the R2jags package:
  http://yusung.blogspot.com/2008/05/r2jags-package-for-running-jags-from-r.html
- Gelman and Hill's book website with code:
  http://www.stat.columbia.edu/~gelman/arm/software/
- Rebecca Steorts' presentation on JAGS:
  http://www.stat.ufl.edu/~rsteorts/jags_present.pdf

4.2 Using JAGS via Terminal

JAGS can also be run straight from the command line - on Windows and Unix systems alike. Probably the most feasible way to do this is to write a script file with the following parts, and save it as `angell.jags`:

```
model clear
data clear
load dic
model in "angell.mod"
data in "angell.dat"
compile, nchains(2)
inits in "angell1.inits", chain(1)
inits in "angell2.inits", chain(2)
initialize
update 2500, by(100)
monitor alpha, thin(2)
monitor beta1, thin(2)
monitor beta2, thin(2)
monitor deviance, thin(2)
update 2500, by(100)
coda *
```

You can run this script file by opening a Terminal window, changing to the working directory (WD) in which all the above files are located -

```
cd /Users/johanneskarreth/R/Bayes/angell
```

and then simply telling JAGS to run the script:

```
jags angell.jags
```

In more detail:

- ```
  model clear
  data clear
  load dic
  ```

  Remove previous data and models (if applicable), load the `dic` module so you can monitor the model deviance late**R**.

- `model in "angell.mod"`

  Use the model angell.mod, which is saved in your WD, and looks like a regular BUGS model. Make sure you use the exact and full name of the model file as it is in your working directory, otherwise JAGS will not find it. Look out for hidden file name extensions...
  ```

```
model {
for(i in 1:N){
moral[i]~dnorm(mu[i], tau)
mu[i]<-alpha + beta1*hetero[i] + beta2*mobility[i]
}

alpha~dnorm(0, .01)
beta1~dunif(-100000,100000)
beta2~dunif(-100000,100000)
tau~dgamma(.01,.01)
}
```

- `model in "angell.mod"`

  Use the data angell.dat. These data can be saved as vectors in one file that you name `angell.dat`:

  ```
  "moral" <- c( 19, 17, ...)
  "hetero" <- c( 20.6, 15.6, ...)
  "mobility" <- c( 15, 20.2, ...)
  "N" <- 43
  ```

  You can create this data file by hand (inconvenient), or you can work with some **R** commands to do this automatically. If `angell` is a data frame object in **R**, you may do this:

  ```
  > angell.list <- as.list(angell)
  > dump("angell.list", file = "angell.dump")
  > bugs2jags("angell.dump", "angell.dat")
  ```

  The first command converts the data frame into a list; the second command writes out the data in BUGS format as a file to your working directory; the third command (part of the coda package) translates it into JAGS format. Both angell.dump and angell.dat are written to your working directory, so be sure that you have specified that in the beginning. Also be sure to visually inspect your data file was created correctly.

- `compile, nchains(2)`

  Compile the models and run two Markov chains.

- `inits in "angell1.inits", chain(1)`
  `inits in "angell2.inits", chain(2)`

  Use the starting values you provide in angell1.inits and angell2.inits, each of which could look like this:

  ```
  "alpha" <- c(0)
  "beta1" <- c(0)
  "beta2" <- c(0)
  ```

This way, you can specify different starting values for each chain.

- `initialize`

  Initialize and run the model.

- `update 2500, by(100)`

  Specify 2500 updates before you start monitoring your parameters of interest.

- ```
  monitor alpha, thin(2)
  monitor beta1, thin(2)
  monitor beta2, thin(2)
  monitor deviance, thin(2)
  ```

  Monitor the values for these parameters, in this case the three regression coefficients and the model deviance. `thin(2)` specifies that only each second draw from the chains will be used for your model output (compare this to the thinning command in WinBUGS).

- `update 2500, by(100)`

- `coda *, stem(angell_out)`

  Tell JAGS to produce coda files that all begin with the stem `angell_out` and are put in your WD. These can then be analyzed with the tools described in this tutorial.

5 Using the boa and coda packages in R

You can use either of the two packages boa or coda in **R** to analyze your BUGS/JAGS output, regardless of the BUGS software you used to fit your model (i.e. WinBUGS/OpenBUGS/JAGS). These are the key steps to get your data into **R** to use boa or coda:

- Fit your model in WinBUGS/OpenBUGS/JAGS, and identify where your software saved the chains and index files – most likely in the working directory where the other components of your model (data, model, inits) are.
- Give these files a proper name, for instance `angell_out_chain1.txt` and `angell_out_chain2.txt`.
- In **R**, load the boa or coda package, whichever you prefer:

  ```
  > library(boa)
  > library(coda)
  ```

- Now, read your chain and index files into **R**, via the commands below.
- If you prefer the usual **R** command-line behavior to the boa/coda menu option, both packages also can be used via the command line.
  ```

- In coda, it should work like this:
  - Set your working directory:

    ```
    > setwd("/Users/johanneskarreth/R/Bayes/angell")
    ```
  - Read in your BUGS/JAGS output. This requires that the chains and index files (see above) are in your working directory.

    ```
    > chain1 <- read.coda("angell_out_chain1.txt", "angell_out_index.txt")
    > chain2 <- read.coda("angell_out_chain2.txt", "angell_out_index.txt")
    > angell.chains <- as.mcmc.list(list(chain1, chain2))
    ```
  - Now you can analyze using some of the commands listed in

    ```
    > help(package=coda)
    ```

    for instance:

    ```
    > summary(angell.chains)
    > traceplot(angell.chains)
    ```

    You should be able to play around with graphical parameters and different printing devices this way.
- In boa, try something like:
  - Start the boa session:

    ```
    > boa.init()
    ```
  - `> my.model.chains <- boa.importBUGS("angell_out", "~/Bayes/angell")`

    where `angell_out` is the stem of your chains and index files; and `"~/Bayes/angell/"` is the folder where your chains and index files are saved.
  - Now you can analyze using some of the commands listed in

    ```
    help(package=boa)
    ```

    for instance:

    ```
    summary(angell.chains)
    boa.plot(angell.chains)
    ```

    You should be able to play around with graphical parameters and different printing devices this way.
- Both the superdiag and mcmcplots packages also should work like described above once you have declared your model output an MCMC object.

6 Differences between JAGS and WinBUGS

There are just a few minor differences between JAGS and WinBUGS that require small adjustments in the code. Two that are relevant for the assignments in this course are described below. A few others that you might encounter in your work are listed in the JAGS manual, section 7.

6.1 Logit models and the p.bound workaround:

Andrew Gelman's blog entry [3] on specifying p.bound:

> *Aurelien Madouasse writes:*
>
> > *I am currently fitting a multilevel logistic model using WinBUGS. I have adapted the example you provide in 'Data analysis using Regression and Multilevel/Hierarchical Models' p. 381-2:*
> >
> > ```
> > y[i] ~ dbin(p.bound[i], 1)
> > p.bound[i] <- max(0, min(1, p[i]))
> > logit(p[i]) <- Xbeta[i] ...
> > ```
> >
> > *I was wondering what is the aim of the p.bound variable. When I use p[i] instead WinBUGS crashes. I thought that the inverse logit was bounded between 0 and 1 so I don't see the point of constraining it to be in this interval. What do I miss?*
>
> *My reply: Yes, inverse logit is bounded, but I think Bugs sometimes messes up and gets it outside of the bound. The other thing is that when you change the specification (in this case, using p.bound) it changes the sampler that Bugs uses. Maybe it's switching from the buggy adaptive rejection sampler to the foolproof Metropolis or slice sample***R**.

6.2 Ordered logit models in JAGS:

From the JAGS manual (currently p. 34):

> Prior ordering of top-level parameters in the model can be achieved using the sort function, which sorts a vector in ascending orde**R**. Symmetric truncation relations like this
>
> ```
> alpha[1] ~ dnorm(0, 1.0E-3) I(,alpha[2])
> alpha[2] ~ dnorm(0, 1.0E-3) I(alpha[1],alpha[3])
> alpha[3] ~ dnorm(0, 1.0E-3) I(alpha[2],)
> ```
>
> should be replaced by this
>
> ```
> for (i in 1:3) {
> alpha0[i] ~ dnorm(0, 1.0E-3)
> }
> alpha[1:3] <- sort(alpha0)
> ```

Accordingly, we need to adjust the code for a simple ordered logit model like displayed below.

[3] http://www.stat.columbia.edu/~cook/movabletype/archives/2009/04/pbound_in_multi.html

BUGS:

```
model{
for(i in 1:N){
for(j in 1:2){
logit(gamma[i,j]) <- theta[j] - mu[i]
}
quality[i] ~ dcat(p[i,1:3])
p[i,1]<- gamma[i,1]
p[i,2] <- gamma[i,2] - gamma[i,1]
p[i,3] <- 1-gamma[i,2]
mu[i] <-  b[1]*price[i] + b[2]*sodium[i] + b[3]*alcohol[i]+b[4]*calories[i]
}
for(j in 1:3){
pred[i,j] <- equals(p[i,j], ranked(p[i,1:3],3)) ## pred[i,j] is always 0 or 1
}
predcat[i] <- pred[i,1] + 2*pred[i,2] + 3*pred[i,3]
}
for(m in 1:4){
b[m] ~ dnorm(0, .0001)
}
theta[1] ~ dnorm(0,.1)I(0, theta[2])
theta[2] ~ dnorm(0,.1)I(theta[1], )
}
```

JAGS:

```
model{
for (i in 1:N){
for (j in 1:2){
logit(gamma[i,j]) <- theta1[j] - mu[i]
}
quality[i] ~ dcat(p[i,1:3])
p[i,1]<- gamma[i,1]
p[i,2] <- gamma[i,2] - gamma[i,1]
p[i,3] <- 1-gamma[i,2]
mu[i] <-  b1*price[i] + b2*sodium[i] + b3*alcohol[i]+b4*calories[i]
}
for (i in 1:2) {
theta[i] ~ dnorm(0, 1.0E-3)
}
theta1[1:2] <- sort(theta)
b1 ~ dnorm(0, .1)
b2 ~ dnorm(0, .1)
```

```
b3 ~ dnorm(0, .1)
b4 ~ dnorm(0, .1)
}
```

7 Following this course using JAGS

You can find modified (if necessary) JAGS code for all models presented in this course on my website at http://spot.colorado.edu/~joka5204/bayes2012.htm. Aside from the web in general, one good resource for JAGS is reading the discussion board at http://sourceforge.net/projects/mcmc-jags/forums/forum/610037.