Using JAGS via R

Johannes Karreth
University at Albany, SUNY
jkarreth@albany.edu

ICPSR Summer Program 2015

All code used in this tutorial can also be found on my course website at `http://www.jkarreth.net/bayes-icpsr.html`. Updated code will be posted there.

If you find any errors in this document or have suggestions for improvement, please don't hesitate to email me.

Contents

This tutorial focuses on using JAGS for fitting Bayesian models via R. There are other options for fitting Bayesian models that we will briefly discuss during the workshop. You can find more information about them at the end of this tutorial and on my workshop website.

1 Using R as frontend

A convenient way to fit Bayesian models using JAGS (or WinBUGS or OpenBUGS) is to use R packages that function as frontends for JAGS. These packages make it easy to process the output of Bayesian models and present it in publication-ready form. In this tutorial, I focus on the R2jags and runjags packages. rjags is another package for the same purpose.

2 What are JAGS, R2jags, ...?

JAGS (Plummer, 2011) is Just Another Gibbs Sampler 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 (Su and Yajima, 2012) is an R package that allows fitting JAGS models from within R. Almost all examples in Gelman and Hill's *Data Analysis Using Regression and Multilevel/Hierarchical Models* (2007) can be worked through equivalently in JAGS, using R2jags.

rjags (Plummer, 2013) is another R package that allows fitting JAGS models from within R. R2jags depends on it. Simon Jackman's *Bayesian Analysis for the Social Sciences* (2009) provides many examples using rjags, and so does John Kruschke's *Doing Bayesian Data Analysis* (2011).

runjags (Denwood, Under review) allows some additional functionalities, including parallel computing.

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

3 Installing JAGS and R2jags

1. Install the most recent R version from the CRAN website: `http://cran.r-project.org/bin/macosx` or `http://cran.r-project.org/bin/windows`.
2. Mac users: Install the GNU Fortran (`gfortran-4.2.3.dmg`) library from the CRAN tools directory: `http://cran.r-project.org/bin/macosx/tools`.
3. Install JAGS version 3.4.0 from Martyn Plummer's repository: `http://sourceforge.net/projects/mcmc-jags/files/JAGS/3.x/`.
4. Start the Terminal (Mac) or Console (Windows) and type

   ```
   jags
   ```

   If JAGS is installed, you will receive the following message:

   ```
   Welcome to JAGS 3.4.0 on Tue Dec 23 20:11:23 2014
   JAGS is free software and comes with ABSOLUTELY NO WARRANTY
   Loading module: basemod: ok
   Loading module: bugs: ok
   .
   ```

   You can quit the Terminal/Console again.
5. Install the packages R2jags, coda, R2WinBUGS, lattice, and (lastly) rjags from within R or RStudio, via the Package Installer, or by using

   ```
   > install.packages("R2jags", dependencies = TRUE, repos = "http://cran.us.r-project.org")
   ```

   The `dependencies = TRUE` option will install the aforementioned packages automatically.

6. Install the package runjags from within R or RStudio, via the Package Installer, or by using

```
> install.packages("runjags", dependencies = TRUE, repos = "http://cran.us.r-project.org")
```

7. Install the package MCMCpack from within R or RStudio, via the Package Installer, or by using

```
> install.packages("MCMCpack", dependencies = TRUE, repos = "http://cran.us.r-project.org")
```

8. Download a scientific text editor for writing R and JAGS code. I recommend TextWrangler for Mac (`http://www.barebones.com/products/textwrangler/`), Sublime Text (`http://www.sublimetext.com`) or Notepad++ for Windows (`http://notepad-plus-plus.org`). RStudio is a very neat integrated environment for using R, but a separate text editor will be useful from time to time to inspect JAGS model files and other files.

9. 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`. Ask me if you would like help with this.

10. You should now be able to run the following code in R, taken directly from the help file for the `jags` function:

```
> library(R2jags)
>
> # An example model file is given in:
> model.file <- system.file(package = "R2jags", "model", "schools.txt")
>
> # data
> J <- 8.0
> y <- c(28.4,7.9,-2.8,6.8,-0.6,0.6,18.0,12.2)
> sd <- c(14.9,10.2,16.3,11.0,9.4,11.4,10.4,17.6)
>
> jags.data <- list("y","sd","J")
> jags.params <- c("mu","sigma","theta")
> jags.inits <- function(){
+   list("mu"=rnorm(1),"sigma"=runif(1),"theta"=rnorm(J))
+ }
>
> # Fit the model
> jagsfit <- jags(data=list("y","sd","J"), inits = jags.inits,
+   jags.params, n.iter = 10, model.file = model.file)

## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
##    Graph Size: 41
##
## Initializing model
```

This should take less than a second and you should see the output above in your R console.

4 Fitting Bayesian models using R2jags

Just like R2WinBUGS,[1] the purpose of R2jags is to allow fitting 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:

[1] 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 use R2WinBUGS and R.

4.1 Preparing the data and model

For an example dataset, we simulate our own data in R. For this tutorial, we aim to fit a linear model, so we create a continuous outcome variable y as a function of two predictors x_1 and x_2 and a disturbance term e. We simulate a dataset with 100 observations.

- First, we create the predictors:

```
> n.sim <- 100; set.seed(123)
> x1 <- rnorm(n.sim, mean = 5, sd = 2)
> x2 <- rbinom(n.sim, size = 1, prob = 0.3)
> e <- rnorm(n.sim, mean = 0, sd = 1)
```

- Next, we create the outcome y based on coefficients b_1 and b_2 for the respective predictors and an intercept a:

```
> b1 <- 1.2
> b2 <- -3.1
> a <- 1.5
> y <- a + b1 * x1 + b2 * x2 + e
```

- Now, we combine the variables into one dataframe for processing later:

```
> sim.dat <- data.frame(y, x1, x2)
```

- And we create and summarize a (frequentist) linear model fit on these data:

```
>   freq.mod <- lm(y ~ x1 + x2, data = sim.dat)
>   summary(freq.mod)

##
## Call:
## lm(formula = y ~ x1 + x2, data = sim.dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.3432 -0.6797 -0.1112  0.5367  3.2304
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.84949    0.28810    6.42 5.04e-09 ***
## x1           1.13511    0.05158   22.00  < 2e-16 ***
## x2          -3.09361    0.20650  -14.98  < 2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9367 on 97 degrees of freedom
## Multiple R-squared:  0.8772,Adjusted R-squared:  0.8747
## F-statistic: 346.5 on 2 and 97 DF,  p-value: < 2.2e-16
```

Now, we write a model for JAGS and save it as the text file `"bayes.mod"` in your working directory. (Do not paste this model straight into R yet.) You can set your working directory via:

```
> setwd("~/Documents/Uni/9 - ICPSR/2015/Applied Bayes/Tutorials/2 - JAGS and R")
```

The model looks just like the JAGS models shown in throughout this course:

```
model {
for(i in 1:N){
y[i] ~ dnorm(mu[i], tau)
mu[i] <- alpha + beta1 * x1[i] + beta2 * x2[i]
}

alpha ~ dnorm(0, .01)
beta1 ~ dunif(-100, 100)
beta2 ~ dunif(-100, 100)
tau ~ dgamma(.01, .01)
}
```

Instead of saving the model in your WD, you can also enter it in your R script:

```
> bayes.mod <- function()  {

+ for(i in 1:N){
+ y[i] ~ dnorm(mu[i], tau)
+ mu[i] <- alpha + beta1 * x1[i] + beta2 * x2[i]
+ }

+ alpha ~ dnorm(0, .01)
+ beta1 ~ dunif(-100, 100)
+ beta2 ~ dunif(-100, 100)
+ tau ~ dgamma(.01, .01)

+ }
```

Now define the vectors of the data matrix for JAGS:

```
> y <- sim.dat$y
> x1 <- sim.dat$x1
> x2 <- sim.dat$x2
> N <- nrow(sim.dat)
```

Read in the data frame for JAGS

```
> sim.dat.jags  <- list("y", "x1", "x2", "N")
```

(You could also do this more conveniently using the `as.list` command on your data frame:)

```
> sim.dat.jags  <- as.list(sim.dat)
```

Note, though, that you will also need to specify any other variables not in the data, like in this case N. So here, you would need to add:

```
> sim.dat.jags$N  <- nrow(sim.dat)
```

Define the parameters whose posterior distributions you are interested in summarizing later:

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

Now, we need to define the starting values for JAGS. Per Gelman and Hill (2007, 370), you can use a function to do this. This function creates a list that contains one element for each parameter. Each parameter then gets assigned a random draw from a normal distributio as a starting value. This random draw is created using the `rnorm` function. The first argument of this function is the number of draws. If your parameters are not indexed in the model code, this argument will be `1`. If your `jags` command below then specifies more than one chain, each chain will start at a different random value for each parameter.

```
> bayes.mod.inits <- function(){
+        list("alpha" = rnorm(1), "beta1" = rnorm(1), "beta2" = rnorm(1))
+   }
```

Alternatively, if you want to have control over which starting values are chosen, you can provide specific separate starting values for each chain:

```
> inits1 <- list("alpha" = 0, "beta1" = 0, "beta2" = 0)
> inits2 <- list("alpha" = 1, "beta1" = 1, "beta2" = 1)
> inits3 <- list("alpha" = -1, "beta1" = -1, "beta2" = -1)
> bayes.mod.inits <- list(inits1, inits2, inits3)
```

Before using R2jags the first time, you need to load the package, and you might need to set a random number seed. To do this, type

```
> library(R2jags)
> set.seed(123)
```

directly in R or in your R script. You can choose any not too big number here. Setting a random seed before fitting a model is also good practice for making your estimates replicable. We will discuss replication in more detail in Weeks 3–4.

4.2 Fitting the model

Fit the model in JAGS:

```
> bayes.mod.fit <- jags(data = sim.dat.jags, inits = bayes.mod.inits,
+   parameters.to.save = bayes.mod.params, n.chains = 3, n.iter = 9000,
+   n.burnin = 1000,
+   model.file = "~/Documents/Uni/9 - ICPSR/2015/Applied Bayes/Tutorials/2 - JAGS and R/bayes.mod")

## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
##    Graph Size: 511
##
## Initializing model
```

Note: If you use as your model file the function you gave directly to R above, then remove the quotation marks:

```
> bayes.mod.fit <- jags(data = sim.dat.jags, inits = bayes.mod.inits,
+   parameters.to.save = bayes.mod.params, n.chains = 3, n.iter = 9000,
+   n.burnin = 1000, model.file = bayes.mod)

## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
##    Graph Size: 511
##
## Initializing model
```

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

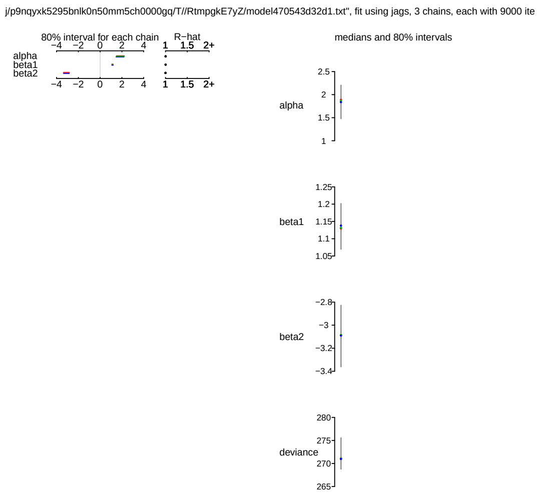
```
> bayes.mod.fit.upd <- update(bayes.mod.fit, n.iter=1000)
> bayes.mod.fit.upd <- autojags(bayes.mod.fit)
```

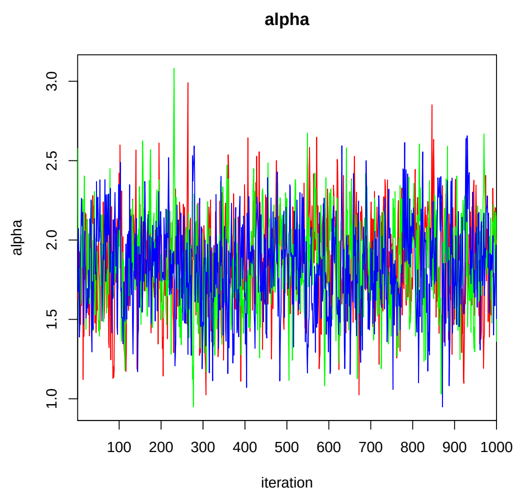
This function will auto-update until convergence.

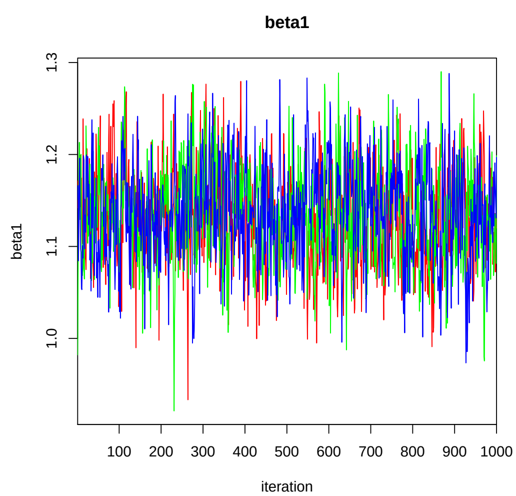
4.3 Diagnostics

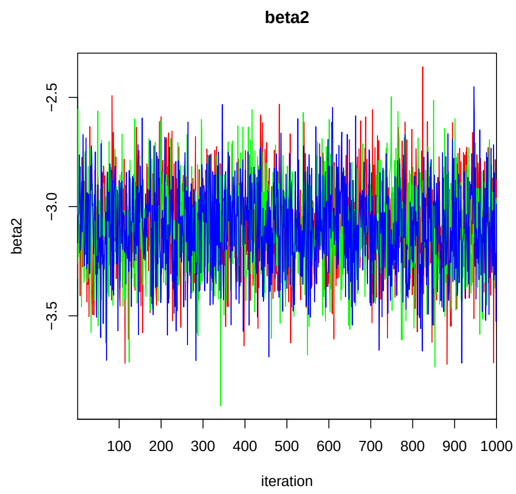
One of the neat things about running JAGS or BUGS from R is that this offers seamless, and therefore quick, access to convergence diagnostics after fitting a model. See for yourself:

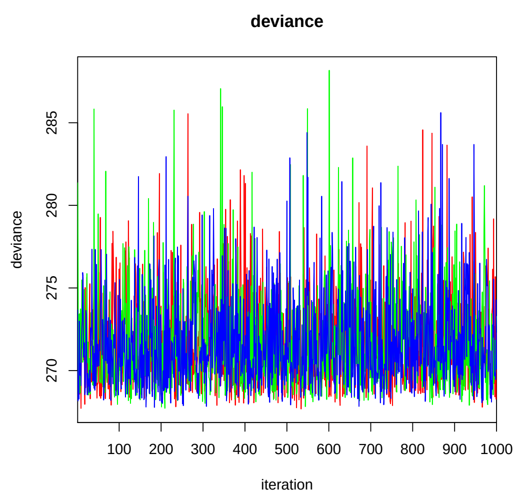
```
> print(bayes.mod.fit)

## Inference for Bugs model at "/var/folders/8j/p9nqyxk5295bnlk0n50mm5ch0000gq/T//RtmpgkE7yZ/model470543d32d1.txt", fit using jags,
##  3 chains, each with 9000 iterations (first 1000 discarded), n.thin = 8
##  n.sims = 3000 iterations saved
##          mu.vect sd.vect    2.5%     25%     50%     75%
## alpha      1.854   0.289   1.281   1.660   1.863   2.051
## beta1      1.134   0.052   1.036   1.097   1.134   1.168
## beta2     -3.092   0.210  -3.496  -3.239  -3.087  -2.943
## deviance 271.766   2.871 268.240 269.647 271.079 273.257
##            97.5%  Rhat n.eff
## alpha      2.396 1.003   820
## beta1      1.237 1.003   810
## beta2     -2.692 1.001  3000
## deviance 278.737 1.001  3000
##
## For each parameter, n.eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 4.1 and DIC = 275.9
## DIC is an estimate of expected predictive error (lower deviance is better).

> plot(bayes.mod.fit)
> traceplot(bayes.mod.fit)
```


j/p9nqyxk5295bnlk0n50mm5ch0000gq/T//RtmpgkE7yZ/model470543d32d1.txt", fit using jags, 3 chains, each with 9000 ite
80% interval for each chain
R−hat
alpha
beta1
beta2
medians and 80% intervals
alpha
beta1
beta2
deviance


alpha
alpha
iteration


beta1
beta1
iteration





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

- ```
  > pdf("~/Documents/Uni/9 - ICPSR/2015/Applied Bayes/Tutorials/2 - JAGS and R/bayes_trace.pdf")
  ```

  ... defines that the plot will be saved as a PDF file with the name "bayes_trace.pdf" in your working directory. [2]

- ```
  > traceplot(bayes.mod.fit)
  ```

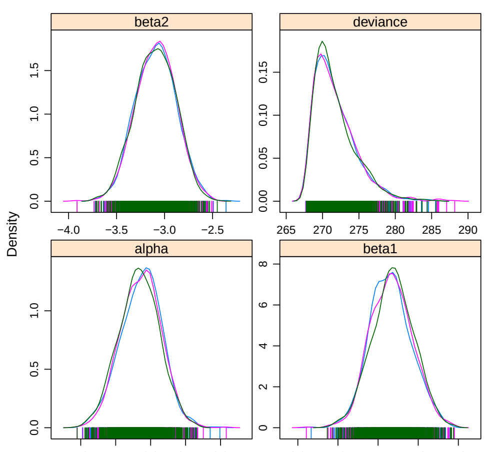
  creates the plot in the background (you will not see it).

- ```
  > 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:

```
> bayes.mod.fit.mcmc <- as.mcmc(bayes.mod.fit)
> summary(bayes.mod.fit.mcmc)
```

[2] LaTeX cannot process file names with periods, so if you use LaTeX and try to include the graphics file angell.trace, LaTeX will not compile your document.
```

```
##
## Iterations = 1001:8993
## Thinning interval = 8
## Number of chains = 3
## Sample size per chain = 1000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##              Mean      SD  Naive SE Time-series SE
## alpha      1.854 0.28863 0.0052697       0.008836
## beta1      1.134 0.05185 0.0009466       0.001616
## beta2     -3.092 0.20974 0.0038294       0.003890
## deviance 271.766 2.87084 0.0524142       0.057127
##
## 2. Quantiles for each variable:
##
##             2.5%     25%     50%     75%   97.5%
## alpha      1.281   1.660   1.863   2.051   2.396
## beta1      1.036   1.097   1.134   1.168   1.237
## beta2     -3.496  -3.239  -3.087  -2.943  -2.692
## deviance 268.240 269.647 271.079 273.257 278.737
```

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

- Plot:

```
> library(lattice)
> xyplot(bayes.mod.fit.mcmc)
```

- You can customize the plot layout (you can use other Lattice options here as well):

```
> xyplot(bayes.mod.fit.mcmc, layout=c(2,2), aspect="fill")
```

- Density plot:

```
> densityplot(bayes.mod.fit.mcmc)
```

```
> densityplot(bayes.mod.fit.mcmc, layout=c(2,2), aspect="fill")
```



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

```
> pdf("~/Documents/Uni/9 - ICPSR/2015/Applied Bayes/Tutorials/2 - JAGS and R/bayes_fit_mcmc_plot.pdf")
> plot(bayes.mod.fit.mcmc)
> dev.off()
```

- Autocorrelation plot, print directly to your working directory:

```
> pdf("~/Documents/Uni/9 - ICPSR/2015/Applied Bayes/Tutorials/2 - JAGS and R/bayes_fit_mcmc_autocorr.pdf")
> autocorr.plot(bayes.mod.fit.mcmc)
> dev.off()
```

- Other diagnostics using CODA:

```
> gelman.plot(bayes.mod.fit.mcmc)
> geweke.diag(bayes.mod.fit.mcmc)

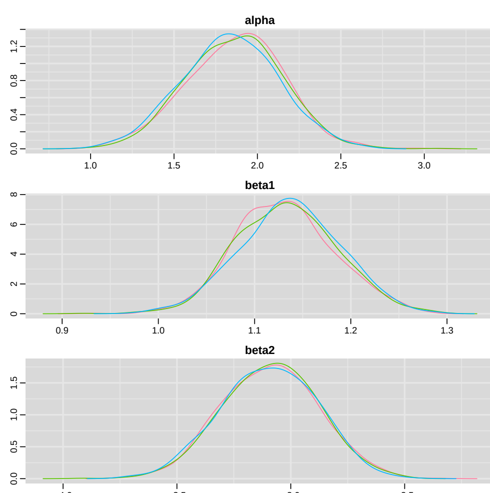
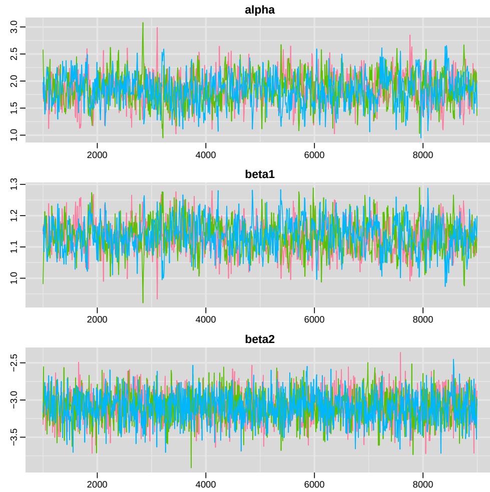
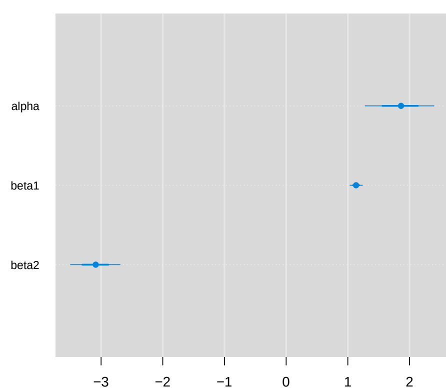
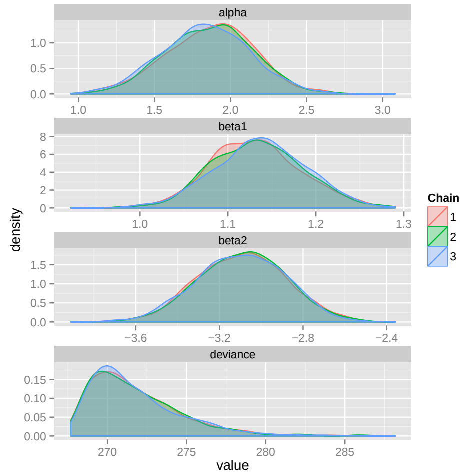
## [[1]]
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##    alpha    beta1    beta2 deviance
## -1.65474  1.71754 -0.03899 -0.30234
##
##
## [[2]]
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##    alpha    beta1    beta2 deviance
##  -0.3987   0.1764   0.8337  -1.0954
##
##
## [[3]]
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##    alpha    beta1    beta2 deviance
##    1.477   -1.651    1.003   -2.182

> geweke.plot(bayes.mod.fit.mcmc)
> raftery.diag(bayes.mod.fit.mcmc)

## [[1]]
##
## Quantile (q) = 0.025
## Accuracy (r) = +/- 0.005
## Probability (s) = 0.95
##
## You need a sample size of at least 3746 with these values of q, r and s
##
## [[2]]
##
## Quantile (q) = 0.025
## Accuracy (r) = +/- 0.005
## Probability (s) = 0.95
##
## You need a sample size of at least 3746 with these values of q, r and s
##
## [[3]]
##
## Quantile (q) = 0.025
## Accuracy (r) = +/- 0.005
## Probability (s) = 0.95
##
## You need a sample size of at least 3746 with these values of q, r and s

> heidel.diag(bayes.mod.fit.mcmc)

## [[1]]
##
##          Stationarity start     p-value
##          test         iteration
## alpha    passed       1         0.185
## beta1    passed       1         0.139
```

```
## beta2    passed       1         0.589
## deviance passed       1         0.827
##
##          Halfwidth Mean   Halfwidth
##          test
## alpha    passed      1.87 0.02971
## beta1    passed      1.13 0.00527
## beta2    passed     -3.09 0.01308
## deviance passed    271.74 0.18831
##
## [[2]]
##
##          Stationarity start     p-value
##          test         iteration
## alpha    passed       1         0.157
## beta1    passed       1         0.393
## beta2    passed       1         0.982
## deviance passed       1         0.604
##
##          Halfwidth Mean   Halfwidth
##          test
## alpha    passed      1.86 0.0296
## beta1    passed      1.13 0.0055
## beta2    passed     -3.09 0.0129
## deviance passed    271.80 0.2042
##
## [[3]]
##
##          Stationarity start     p-value
##          test         iteration
## alpha    passed         1       0.4468
## beta1    passed         1       0.3556
## beta2    passed         1       0.0964
## deviance passed       101       0.0839
##
##          Halfwidth Mean   Halfwidth
##          test
## alpha    passed      1.83 0.03066
## beta1    passed      1.14 0.00568
## beta2    passed     -3.10 0.01360
## deviance passed    271.81 0.20054
```

Quick convergence diagnostics: superdiag

A very convenient function to analyze numerical representations of diagnostics in one sweep is the superdiag package (Tsai, Gill, and Rapkin, 2012).

- First, install the package:

```
> install.packages("superdiag", dependencies = TRUE, repos = "http://cran.us.r-project.org")
```

- Then, load it:

```
> library(superdiag)
```

- Next, all you need to do is:

```
> superdiag(bayes.mod.fit.mcmc, burnin = 100)

## Number of chains = 3
## Number of iterations = 1000 per chain before discarding the burn-in period
## The burn-in period = 100 per chain
## Sample size in total = 2700
##
## ********** The Geweke diagnostic: **********
## Z-scores:
##                          chain1    chain 2   chain 3
## alpha              -0.32702322 -0.8681148 -1.804617
```

```
## beta1              0.47904271  0.5804246  1.541857
## beta2             -1.89980599 -0.2904349  1.577197
## deviance          -0.05770752  0.4806812 -1.169611
## Window From Start  0.10000000  0.4542000  0.719760
## Window From Stop   0.50000000  0.0827600  0.226630
##
## ********** The Gelman-Rubin diagnostic: **********
## Potential scale reduction factors:
##
##          Point est. Upper C.I.
## alpha          1.01       1.03
## beta1          1.01       1.04
## beta2          1.00       1.00
## deviance       1.00       1.00
##
## Multivariate psrf
##
## 1.01
##
## ********** The Heidelberger-Welch diagnostic: **********
##
## Chain 1, epsilon=0.1, alpha=0.05
##          Stationarity start     p-value
##          test         iteration
## alpha    passed       1         0.558
## beta1    passed       1         0.441
## beta2    passed       1         0.491
## deviance passed       1         0.477
##
##          Halfwidth Mean   Halfwidth
##          test
## alpha    passed      1.87 0.03087
## beta1    passed      1.13 0.00553
## beta2    passed     -3.09 0.01380
## deviance passed    271.76 0.18102
##
## Chain 2, epsilon=0.062, alpha=0.1
##          Stationarity start     p-value
##          test         iteration
## alpha    passed       1         0.178
## beta1    passed       1         0.300
## beta2    passed       1         0.907
## deviance passed       1         0.752
##
##          Halfwidth Mean   Halfwidth
##          test
## alpha    passed      1.86 0.03183
## beta1    passed      1.13 0.00595
## beta2    passed     -3.09 0.01363
## deviance passed    271.84 0.21074
##
## Chain 3, epsilon=0.111, alpha=0.05
##          Stationarity start     p-value
##          test         iteration
## alpha    passed       1         0.219
## beta1    passed       1         0.377
## beta2    passed       1         0.122
## deviance passed       1         0.145
##
##          Halfwidth Mean   Halfwidth
##          test
## alpha    passed      1.83 0.0323
## beta1    passed      1.14 0.0060
## beta2    passed     -3.10 0.0137
## deviance passed    271.81 0.2005
##
## ********** The Raftery-Lewis diagnostic: **********
##
## Chain 1, converge.eps = 0.001
## Quantile (q) = 0.025
## Accuracy (r) = +/- 0.005
## Probability (s) = 0.95
##
```

```
## You need a sample size of at least 3746 with these values of q, r and s
##
## Chain 2, converge.eps = 5e-04
## Quantile (q) = 0.001
## Accuracy (r) = +/- 0.005
## Probability (s) = 0.95
##
##           Burn-in  Total Lower bound  Dependence
##           (M)      (N)   (Nmin)       factor (I)
##  alpha    2        173   154          1.12
##  beta1    2        173   154          1.12
##  beta2    2        173   154          1.12
##  deviance 2        173   154          1.12
##
##
## Chain 3, converge.eps = 0.0025
## Quantile (q) = 0.25
## Accuracy (r) = +/- 0.001
## Probability (s) = 0.99
##
## You need a sample size of at least 1244044 with these values of q, r and s
```

Note that the R2jags object by default retains 1000 iterations (through thinning), hence the burn-in period you provide for superdiag must be less than 1000.

Quick diagnostic plots: mcmcplots

A convenient way to obtain graphical diagnostics and results is using the mcmcplots package (Curtis, 2012):

- First, install the package:

```
> install.packages("mcmcplots", dependencies = TRUE, repos = "http://cran.us.r-project.org")
```

- Then, load it:

```
> library(mcmcplots)
```

- Some commands for plots:

```
> denplot(bayes.mod.fit.mcmc)
```

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





As always, check the help files for options to customize these plots.

- 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 browser.

```
> mcmcplot(bayes.mod.fit.mcmc)
```

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

```
> caterplot(bayes.mod.fit.mcmc)
```

```
> caterplot(bayes.mod.fit.mcmc, parms = c("alpha", "beta1", "beta2"),
+   labels = c("alpha", "beta1", "beta2"))
```



More plots: ggmcmc

Yet another option for plotting output is the ggmcmc package (Fernández i Marín, 2013):

- First, install the package:

```
> install.packages("ggmcmc", dependencies = TRUE, repos = "http://cran.us.r-project.org")
```

- Then, load it:

```
> library(ggmcmc)
```

- To use this package, you need to first convert your MCMC object into a ggs object, using the ggs function. This object can then be passed on to the various ggmcmc plotting commands:

```
> bayes.mod.fit.gg <- ggs(bayes.mod.fit.mcmc)
> ggs_density(bayes.mod.fit.gg)
```



- Or, you can use the ggmcmc command to create a PDF file containing a variety of diagnostic plots:

```
> ggmcmc(bayes.mod.fit.gg,
+        file = "~/Documents/Uni/9 - ICPSR/2015/Applied Bayes/Tutorials/2 - JAGS and R/bayes_fit_ggmcmc.pdf")

## Plotting histograms
## Plotting density plots
## Plotting traceplots
## Plotting running means
## Plotting comparison of partial and full chain
## Plotting autocorrelation plots
## Plotting crosscorrelation plot
## Plotting Potential Scale Reduction Factors
## Plotting Geweke Diagnostic
## Plotting caterpillar plot
## pdf
##   2
```

5 Using runjags

Instead of R2jags, you can also use the runjags package to command JAGS via R. Here is an abbreviated version of the workflow for runjags. The workflow mimics what you saw for R2jags above:

```
> library(runjags)
```

For an example dataset, we simulate our own data in R. For this tutorial, we aim to fit a linear model, so we create a continuous outcome variable y as a function of two predictors x_1 and x_2 and a disturbance term e. We simulate 100 observations.

- First, we create the predictors:

```
> n.sim <- 100; set.seed(123)
> x1 <- rnorm(n.sim, mean = 5, sd = 2)
> x2 <- rbinom(n.sim, size = 1, prob = 0.3)
> e <- rnorm(n.sim, mean = 0, sd = 1)
```

- Next, we create the outcome y based on coefficients b_1 and b_2 for the respective predictors and an intercept a:

```
> b1 <- 1.2
> b2 <- -3.1
> a <- 1.5
> y <- a + b1 * x1 + b2 * x2 + e
```

- Now, we combine the variables into one dataframe for processing later:

```
> sim.dat <- data.frame(y, x1, x2)
```

- And we summarize a (frequentist) linear model fit on these data:

```
>   freq.mod <- lm(y ~ x1 + x2, data = sim.dat)
>   summary(freq.mod)

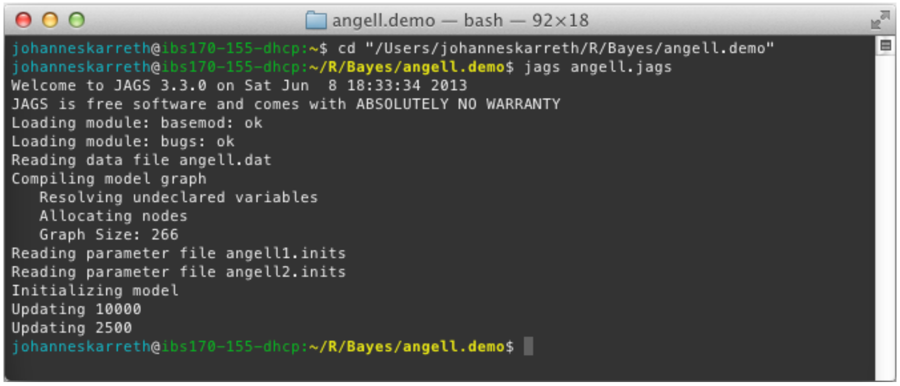
##
## Call:
## lm(formula = y ~ x1 + x2, data = sim.dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.3432 -0.6797 -0.1112  0.5367  3.2304
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.84949    0.28810    6.42 5.04e-09 ***
## x1           1.13511    0.05158   22.00  < 2e-16 ***
## x2          -3.09361    0.20650  -14.98  < 2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9367 on 97 degrees of freedom
## Multiple R-squared:  0.8772,Adjusted R-squared:  0.8747
## F-statistic: 346.5 on 2 and 97 DF,  p-value: < 2.2e-16
```

Create the model object:

```
> bayes.mod <- "model{

for(i in 1:N){
y[i] ~ dnorm(mu[i], tau)
mu[i] <- alpha + beta1 * x1[i] + beta2 * x2[i]
}

alpha ~ dnorm(0, .01)
beta1 ~ dunif(-100, 100)
beta2 ~ dunif(-100, 100)
tau ~ dgamma(.01, .01)

}"
```

Convert the data frame to a list:

```
> sim.list  <- as.list(sim.dat)
```

and add the number of observations:

```
> sim.list$N  <- nrow(sim.dat)
```

Convert the data for runjags:

```
> sim.dat.runjags <- dump.format(sim.list)
```

Specify initial values:

```
> inits1 <- list(alpha = 1, beta1 = 1, beta2 = 1)
> inits2 <- list(alpha = 0, beta1 = 0, beta2 = 0)
> inits3 <- list(alpha = -1, beta1 = -1, beta2 = -1)
```

Fit the model in using `run.jags`:

```
> bayes.mod.fit2 <- run.jags(model = bayes.mod, monitor = c("alpha", "beta1", "beta2"),
+   data = sim.dat.runjags, n.chains = 3, inits = list(inits1, inits2, inits3),
+   burnin = 1000, sample = 5000, keep.jags.files = TRUE)
```

```
## Calling the simulation...
## Welcome to JAGS 3.4.0 on Wed May 27 19:07:26 2015
## JAGS is free software and comes with ABSOLUTELY NO WARRANTY
## Loading module: basemod: ok
## Loading module: bugs: ok
## . . Reading data file data.txt
## . Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
##    Graph Size: 511
## . Reading parameter file inits1.txt
## . Reading parameter file inits2.txt
## . Reading parameter file inits3.txt
## . Initializing model
## . Adapting 1000
## -------------------------------------------------| 1000
## ++++++++++++++++++++++++++++++++++++++++++++++++++ 100%
## Adaptation successful
## . Updating 1000
## -------------------------------------------------| 1000
## ************************************************** 100%
## . . . . Updating 5000
## -------------------------------------------------| 5000
## ************************************************** 100%
## . . . . . . Updating 0
## . Deleting model
## .
## Simulation complete.  Reading coda files...
## Coda files loaded successfully
## Calculating summary statistics...
## Calculating the Gelman-Rubin statistic for 3
## variables....
## Finished running the simulation
## JAGS files were saved to the 'runjagsfiles_2' folder
## in your current working directory
```

Note a few things:

- The model is read into R within quotation marks.
- By setting `keep.jags.files = TRUE`, we get `run.jags` to create a folder "runjagsfiles" in your WD. This folder will contain the same files you obtain from JAGS in the next step below. This can be useful for further processing.
- The option `method` allows for parallel chains on separate cores of your computer and other higher-end computing options. See the help file for `run.jags` for more information.

5.1 Output and diagnostics

`runjags` also allows you to use the same set of commands for diagnostics and accessing the fitted objects as you explored above.

- Summarize the model object

```
> print(bayes.mod.fit)

## Inference for Bugs model at "/var/folders/8j/p9nqyxk5295bnlk0n50mm5ch0000gq/T//RtmpgkE7yZ/model470543d32d1.txt", fit using
##  3 chains, each with 9000 iterations (first 1000 discarded), n.thin = 8
##  n.sims = 3000 iterations saved
##          mu.vect sd.vect    2.5%     25%     50%     75%
## alpha      1.854   0.289   1.281   1.660   1.863   2.051
## beta1      1.134   0.052   1.036   1.097   1.134   1.168
## beta2     -3.092   0.210  -3.496  -3.239  -3.087  -2.943
## deviance 271.766   2.871 268.240 269.647 271.079 273.257
##            97.5%  Rhat n.eff
## alpha      2.396 1.003   820
## beta1      1.237 1.003   810
## beta2     -2.692 1.001  3000
## deviance 278.737 1.001  3000
##
```

```
## For each parameter, n.eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 4.1 and DIC = 275.9
## DIC is an estimate of expected predictive error (lower deviance is better).
```

- Convert to an MCMC object. Here, we use `as.mcmc.list` to keep the 3 chains separate in the resulting MCMC object:

```
> bayes.mod.fit2.mcmc <- as.mcmc.list(bayes.mod.fit2)
> summary(bayes.mod.fit2.mcmc)


##
## Iterations = 2001:7000
## 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
## alpha  1.829 0.2994 0.0024447       0.012116
## beta1  1.138 0.0536 0.0004377       0.002189
## beta2 -3.092 0.2075 0.0016942       0.002772
##
## 2. Quantiles for each variable:
##
##         2.5%    25%    50%    75%  97.5%
## alpha  1.242  1.635  1.830  2.022  2.420
## beta1  1.032  1.104  1.139  1.173  1.244
## beta2 -3.496 -3.233 -3.090 -2.950 -2.687
```

- Use `mcmcplot` and `superdiag` for diagnostics:

```
> mcmcplot(bayes.mod.fit2.mcmc)
> superdiag(bayes.mod.fit2.mcmc, burnin = 1000)
```

6 Using JAGS via the command line

JAGS can also be operated straight from the command line—on Windows and Unix systems alike. This can be useful if you don't want to have R busy fitting models that take a longer time. The most feasible way to do this is to write a script file with the following parts, and save it, for instance as `bayes.jags`. Before running this script, you will need to create some files (see below).

```
model clear
data clear
load dic
model in "bayes.mod"
data in "sim.dat"
compile, nchains(3)
inits in "bayes.mod.inits1", chain(1)
inits in "bayes.mod.inits2", chain(2)
inits in "bayes.mod.inits2", chain(3)
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 *
```

The following instructions are for the Terminal on the Mac, but you can reproduce the same steps on Windows or Linux. You run this script file by opening a Terminal window, changing to the working directory in which all the above files are located -

```
cd "~/Documents/Uni/9 - ICPSR/2015/Applied Bayes/Tutorials/2 - JAGS and R"
```

and then simply telling JAGS to run the script:

```
jags bayes.jags
```

In your Terminal, you will see something like this:



In more detail, this is what each line of the script does:

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

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

- ```
  model in "bayes.mod"
  ```

  Use the model bayes.mod, which is saved in your WD, and looks like a regular JAGS 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. [3] You wrote this model earlier and saved it in your WD as `bayes.mod`:

  ```
  model {
  for(i in 1:N){
  y[i] ~ dnorm(mu[i], tau)
  mu[i] <- alpha + beta1 * x1[i] + beta2 * x2[i]
  }

  alpha ~ dnorm(0, .01)
  beta1 ~ dunif(-100, 100)
  beta2 ~ dunif(-100, 100)
  tau ~ dgamma(.01, .01)
  }
  ```

- ```
  data in "sim.dat"
  ```

  Use the data sim.dat. These data can be saved as vectors in one file that you name `sim.dat`. This file will look something like this:

  ```
  "y" <- c(6.94259729574987, 4.61661626624104, ...
  "x1" <- c(3.87904870689558, 4.53964502103344, ...
  "x2" <- c(0, 1, ...
  "N" <- 100
  ```

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

[3] On the Mac, you can set the Finder to display all file extensions by going to Finder > Preferences > check "Show all filename extensions."
  ```

```
> sim.dat.list <- as.list(sim.dat)
> sim.dat.list$N  <- nrow(sim.dat)
> dump("sim.dat.list", file = "sim.dat.dump")
> bugs2jags("sim.dat.dump", "sim.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 whether your data file was created correctly.

- `compile, nchains(2)`

  Compile the models and run two Markov chains.

- ```
  inits in "bayes.mod.inits1", chain(1)
  inits in "bayes.mod.inits2", chain(2)
  inits in "bayes.mod.inits3", chain(3)
  ```

  Use the starting values you provide in bayes.mod.inits1, bayes.mod.inits2, and bayes.mod.inits3, 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.

- `update 2500, by(100)`

- `coda *, stem(bayes_out)`

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

7 Using the coda package

You can use the coda package in R to analyze your JAGS output. These are the key steps to get your data into R to use CODA:

- Fit your model in 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.
- In R, load the coda package:

```
> library(coda)
```

- Read in your JAGS output. This requires that the chains and index files (see above) are in your working directory.

```
> setwd("~/Documents/Uni/9 - ICPSR/2015/Applied Bayes/Tutorials/2 - JAGS and R/")
> chain1 <- read.coda(output.file = "CODAchain1.txt",
+   index.file = "CODAindex.txt")

## Abstracting alpha ... 1250 valid values
## Abstracting beta1 ... 1250 valid values
## Abstracting beta2 ... 1250 valid values
## Abstracting deviance ... 1250 valid values

> chain2 <- read.coda(output.file = "CODAchain2.txt",
+   index.file = "CODAindex.txt")

## Abstracting alpha ... 1250 valid values
## Abstracting beta1 ... 1250 valid values
## Abstracting beta2 ... 1250 valid values
## Abstracting deviance ... 1250 valid values

> chain3 <- read.coda(output.file = "CODAchain3.txt",
+   index.file = "CODAindex.txt")

## Abstracting alpha ... 1250 valid values
## Abstracting beta1 ... 1250 valid values
## Abstracting beta2 ... 1250 valid values
## Abstracting deviance ... 1250 valid values

> bayes.chains <- as.mcmc.list(list(chain1, chain2, chain3))
```

- Now you can analyze using some of the commands listed in

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

for instance:

```
> summary(bayes.chains)

##
## Iterations = 2501:4999
## Thinning interval = 2
## Number of chains = 3
## Sample size per chain = 1250
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##             Mean      SD  Naive SE Time-series SE
## alpha      1.803 0.28359 0.0046310       0.014504
## beta1      1.143 0.05142 0.0008397       0.002786
## beta2     -3.088 0.20954 0.0034218       0.004091
## deviance 271.768 2.93772 0.0479728       0.071076
##
## 2. Quantiles for each variable:
##
##             2.5%     25%     50%     75%   97.5%
## alpha      1.238   1.614   1.808   1.992   2.355
## beta1      1.041   1.109   1.143   1.179   1.244
## beta2     -3.496  -3.228  -3.089  -2.949  -2.678
## deviance 268.138 269.671 271.082 273.063 279.708

> traceplot(bayes.chains)
```

You should be able to play around with graphical parameters and different printing devices this way.

- Convenient: the superdiag, mcmcplots, and ggmcmc packages also work like described above once you have declared your JAGS output an MCMC object.

8 MCMCpack

MCMCpack (Martin, Quinn, and Park, 2011) is an R package that we will also use in this course. It is as easy to use as the `lm()` command in R and produces the same MCMC output you analyzed above. One potential downside of MCMCpack is that it does not offer as much options for customization as writing your full model in JAGS. For more information on MCMCpack, see `http://mcmcpack.wustl.edu/`.

Because MCMCpack uses an R-like formula, it is straightforward to fit a Bayesian linear model.

```
> library(MCMCpack)
```

For an example dataset, we again simulate our own data in R. We create a continuous outcome variable y as a function of two predictors x_1 and x_2 and a disturbance term e. We simulate 100 observations.

- First, we create the predictors:

```
> n.sim <- 100; set.seed(123)
> x1 <- rnorm(n.sim, mean = 5, sd = 2)
> x2 <- rbinom(n.sim, size = 1, prob = 0.3)
> e <- rnorm(n.sim, mean = 0, sd = 1)
```

- Next, we create the outcome y based on coefficients b_1 and b_2 for the respective predictors and an intercept a:

```
> b1 <- 1.2
> b2 <- -3.1
> a <- 1.5
> y <- a + b1 * x1 + b2 * x2 + e
```

- Now, we combine the variables into one dataframe for processing later:

```
> sim.dat <- data.frame(y, x1, x2)
```

- And we summarize a (frequentist) linear model fit on these data:

```
>   freq.mod <- lm(y ~ x1 + x2, data = sim.dat)
>   summary(freq.mod)

##
## Call:
## lm(formula = y ~ x1 + x2, data = sim.dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.3432 -0.6797 -0.1112  0.5367  3.2304
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.84949    0.28810    6.42 5.04e-09 ***
## x1           1.13511    0.05158   22.00  < 2e-16 ***
## x2          -3.09361    0.20650  -14.98  < 2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9367 on 97 degrees of freedom
## Multiple R-squared:  0.8772,Adjusted R-squared:  0.8747
## F-statistic: 346.5 on 2 and 97 DF,  p-value: < 2.2e-16
```

MCMCpack only requires us to specify one single model object. For the linear model, we use `MCMCregress`. For other models available in MCMCpack, see `help(package = "MCMCpack")`.

```
> bayes.mod.fit3 <- MCMCregress(y ~ x1 + x2, data = sim.dat, burnin = 1000,
+   mcmc = 5000, seed = 123, beta.start = c(0, 0, 0),
+   b0 = c(0, 0, 0), B0 = c(0.1, 0.1, 0.1))
> summary(bayes.mod.fit)

##                    Length Class  Mode
## model               8     jags   list
## BUGSoutput         24     bugs   list
## parameters.to.save  4     -none- character
## model.file          1     -none- character
## n.iter              1     -none- numeric
## DIC                 1     -none- logical
```

The resulting object is already an MCMC object.

8.1 Output and diagnostics

MCMCpack also allows you to use the same set of commands for diagnostics and accessing the fitted objects as you explored above. For example, you may use `mcmcplot` and `superdiag` for diagnostics:

```
> mcmcplot(bayes.mod.fit3)
> superdiag(bayes.mod.fit3, burnin = 1000)
```

9 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://www.jkarreth.net/bayes-icpsr.html`.

The most recent version of this file is posted at `http://www.jkarreth.net/files/Lab2_JAGS.pdf`.

Many JAGS-related questions are answerd at `http://stackoverflow.com/questions/tagged/jags` and the discussion board at `http://sourceforge.net/projects/mcmc-jags/forums/forum/610037`.

10 Other software solutions for Bayesian estimation

Other options to fit Bayesian models include:

- WinBUGS/OpenBUGS. These programs have a GUI, but can also be accessed from R. Gelman and Hill (2007) provide extensive information on how to integrate BUGS and R. OpenBUGS (`http://www.openbugs.net/w/FrontPage`) is the open-source continuation of WinBUGS (`http://www.mrc-bsu.cam.ac.uk/software/bugs/the-bugs-project-winbugs/`) with similar functionality. A convenient way to fit Bayesian models using WinBUGS or OpenBUGS is to use R packages that function as frontends for WinBUGS or OpenBUGS, such as R2WinBUGS (Sturtz, Ligges, and Gelman, 2005) or BRugs (Thomas et al., 2006). R2WinBUGS works very similar to R2jags, so you can adapt the code in this tutorial easily. Mac users can find setup instructions on my course website.

- Stan (`http://mc-stan.org`) is a new and fast program that can be accessed via R (and other statistical packages). The documentation on the Stan website is very easy to follow, and offers tutorials on fitting some example models in Stan. We will show you how to use Stan in Week 4 of this course.

References

Curtis, S. McKay. 2012. *mcmcplots: Create Plots from MCMC Output*. R package version 0.4.1.
URL: *http://CRAN.R-project.org/package=mcmcplots*

Denwood, Matthew J. Under review. "runjags: An R package providing interface utilities, parallel computing methods and additional distributions for MCMC models in JAGS." *Manuscript, University of Copenhagen* .

Fernández i Marín, Xavier. 2013. *ggmcmc: Graphical tools for analyzing Markov Chain Monte Carlo simulations from Bayesian inference*. R package version 0.5.1.
URL: *http://xavier-fim.net/packages/ggmcmc*

Gelman, Andrew, and Jennifer Hill. 2007. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. New York, NY: Cambridge University Press.

Jackman, Simon. 2009. *Bayesian Analysis for the Social Sciences*. Chichester: Wiley.

Kruschke, John. 2011. *Doing Bayesian Data Analysis: A Tutorial Introduction with R*. Oxford: Academic Press / Elsevier.

Martin, Andrew D., Kevin M. Quinn, and Jong Hee Park. 2011. "MCMCpack: Markov Chain Monte Carlo in R." *Journal of Statistical Software* 42 (9): 22.

Plummer, Martyn. 2011. "JAGS Version 3.1.0 User Manual.".

Plummer, Martyn. 2013. *rjags: Bayesian graphical models using MCMC*. R package version 3-10.
URL: *http://CRAN.R-project.org/package=rjags*

Sturtz, Sibylle, Uwe Ligges, and Andrew Gelman. 2005. "R2WinBUGS: A Package for Running WinBUGS from R." *Journal of Statistical Software* 12 (3): 1–16.

Su, Yu-Sung, and Masanao Yajima. 2012. *R2jags: A Package for Running jags from R*. R package version 0.03-08.
URL: *http://CRAN.R-project.org/package=R2jags*

Thomas, Andrew, Bob O'Hara, Uwe Ligges, and Sibylle Sturtz. 2006. "Making BUGS Open." *R News* 6 (1): 12–17.

Tsai, Tsung-han, Jeff Gill, and Jonathan Rapkin. 2012. *superdiag: R Code for Testing Markov Chain Nonconvergence*. R package version 1.1.
URL: *http://CRAN.R-project.org/package=superdiag*