Applied Bayesian Modeling
Using JAGS and BUGS via R

Johannes Karreth
Ursinus College
jkarreth@ursinus.edu

ICPSR Summer Program 2017

All code used in this tutorial can also be found on my github page at `https://github.com/jkarreth/Bayes`. Updated code will be posted there.

If you find any errors in this document or have suggestions for improvement, please email me.

Contents

1 How to use this tutorial

This tutorial focuses on using JAGS and WinBUGS/OpenBUGS 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 website, `www.jkarreth.net/bayes-icpsr.html`. Some sections are relevant for Mac or Windows users only as indicated.

In this tutorial, R code that you would enter in your script file or in the command line is preceded by the > character, and by + if the current line of code continues from a previous line. You do not need to type this character in your own code. Note that copying and pasting code from the PDF version of this tutorial may lead to errors when trying to execute code. Please copy code from the R script used to produce this tutorial; this script can be found at `https://www.dropbox.com/s/vkrhzxdrjz1kzry/Lab3-4_JAGS-BUGS.R?raw=1`.

2 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 do all your Bayesian data analysis in R, including:

- importing and preparing the data
- writing the empirical model
- estimate the model using MCMC
- process the output of Bayesian models
- present output in publication-ready form

In this tutorial, I focus on the R2jags and R2WinBUGS/R2OpenBUGS packages that you encounter in Gelman and Hill (2007), as well as a few other options.

3 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. It is a free, open-source program. More information can be found in the excellent JAGS manual at `http://sourceforge.net/projects/mcmc-jags/`.

WinBUGS (Spiegelhalter et al., 2003) is a Windows-only program for Bayesian estimation with a graphical user interface. It has been a very popular option for Bayesian modeling in the past 10–15 years. WinBUGS is available for free at `http://www.mrc-bsu.cam.ac.uk/software/bugs/`, but it is not under active development anymore.

OpenBUGS (Lunn et al., 2009) is an open-source and free version of WinBUGS. The frontend and most practical aspects of the program are virtually identical with WinBUGS. OpenBUGS is available as a Windows program at `http://www.openbugs.net`. Downloading OpenBUGS will give you access to all functionality of WinBUGS as far as our workshop is concerned.

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.

R2WinBUGS/R2OpenBUGS (Sturtz, Ligges, and Gelman, 2005) are R packages similar to **R2jags** that allow controlling WinBUGS/OpenBUGS 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, 2016) allows some additional functionalities, including parallel computing.

4 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 4.2.0 from Martyn Plummer's repository: `http://sourceforge.net/projects/mcmc-jags/files/JAGS/`.
4. Start the Terminal (Mac) or Console (Windows) and type

   ```
   jags
   ```

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

   ```
   Welcome to JAGS 4.2.0 on Wed Jun 22 12:55:34 2017
   JAGS is free software and comes with ABSOLUTELY NO WARRANTY
   Loading module: basemod: ok
   Loading module: bugs: ok
   .
   ```

   You can quit the Terminal/Console now.
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 = "https://cloud.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 = "https://cloud.r-project.org")
   ```

7. Download a scientific text editor for writing R and JAGS code. 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/BUGS model files and other files. I recommend:
   - TextWrangler for Mac (`http://www.barebones.com/products/textwrangler/`)
   - Notepad++ for Windows (`http://notepad-plus-plus.org`).
8. 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.[1] 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.
9. 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
   ```

[1] See `http://martynplummer.wordpress.com/2011/11/04/rjags-3-for-mac-os-x/`.

```
> 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 information:
##    Observed stochastic nodes: 8
##    Unobserved stochastic nodes: 10
##    Total graph size: 50
##
## Initializing model
```

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

5 Fitting Bayesian models using R2jags

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:

5.1 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/Dropbox/Uni/9 - ICPSR/2017/Applied Bayes/Tutorials/2 - JAGS and R")
```

Be sure to provide the file path to your own working directory here when using this code.

The model looks just like the JAGS models shown 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 distribution 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)
```

Note: Here, I did not specify a starting value for the node `tau`. This will lead JAGS (or BUGS) to generate a random number as a starting value for `tau`. In general, any node for which you do not explicitly generate starting values will receive a random starting value. This is not a problem computationally, but undesirable from a reproducibility perspective. (More on this later in this workshop.)

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.

5.2 Fitting the model

Fit the model in JAGS. I'm giving the fit object the unwieldy name `bayes.mod.fit.R2jags` here to distinguish it from other objects later. When you work through this code, be sure to provide the file path to your own `bayes.mod` here.

```
> bayes.mod.fit.R2jags <- 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/Dropbox/Uni/9 - ICPSR/2017/Applied Bayes/Tutorials/2 - JAGS and R/bayes.mod")

## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 100
##    Unobserved stochastic nodes: 4
##    Total graph size: 516
##
## 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.R2jags <- 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 information:
##    Observed stochastic nodes: 100
##    Unobserved stochastic nodes: 4
##    Total graph size: 516
##
## Initializing model
```

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

```
> bayes.mod.fit.R2jags.upd <- update(bayes.mod.fit.R2jags, n.iter = 1000)
> bayes.mod.fit.R2jags.upd <- autojags(bayes.mod.fit.R2jags)
```

This function will auto-update until convergence (as defined by the $\hat{R}$ diagnostic approaching 1).

5.3 Results

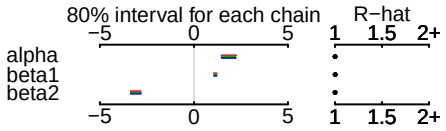
Running JAGS (or BUGS) from R offers seamless, and therefore quick, access to results after fitting a model. See for yourself:

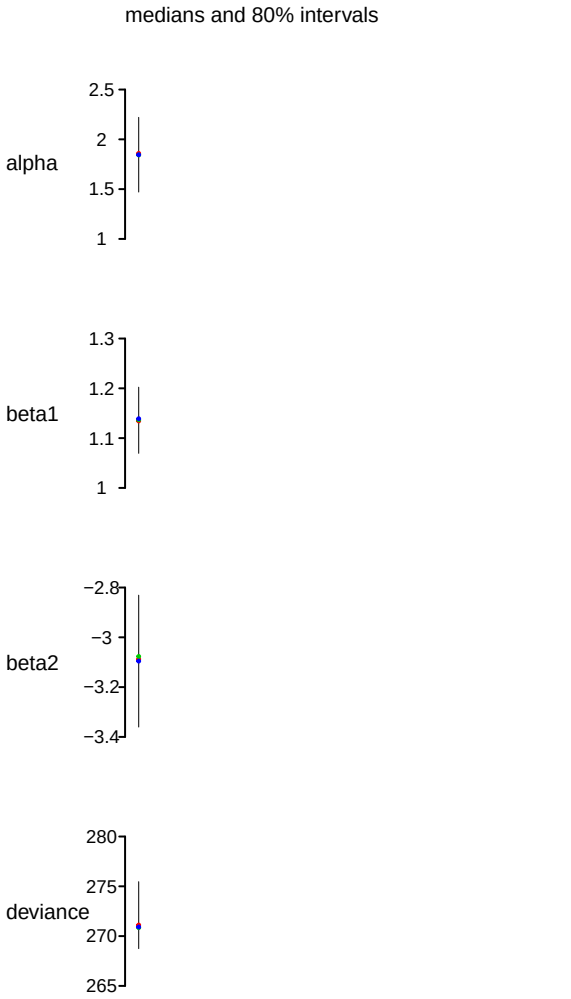
```
> print(bayes.mod.fit.R2jags)

## Inference for Bugs model at "/var/folders/8j/p9nqyxk5295bnlk0n50mm5ch0000gq/T//RtmpBjvkP8/model439112527836.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%   97.5%  Rhat n.eff
## alpha      1.846   0.292   1.279   1.650   1.849   2.046   2.409 1.001  2900
## beta1      1.136   0.052   1.038   1.099   1.137   1.171   1.238 1.002  1100
## beta2     -3.095   0.206  -3.516  -3.235  -3.088  -2.958  -2.699 1.001  3000
## deviance 271.687   2.895 268.155 269.618 270.959 273.031 278.935 1.001  2000
##
## 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 = 4.2 and DIC = 275.9
## DIC is an estimate of expected predictive error (lower deviance is better).

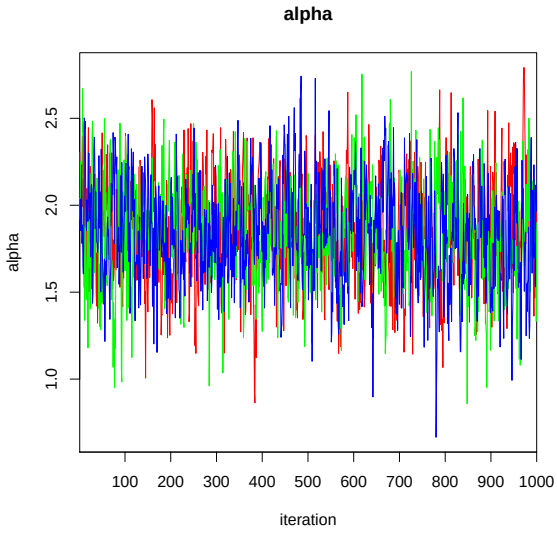
> plot(bayes.mod.fit.R2jags)
```

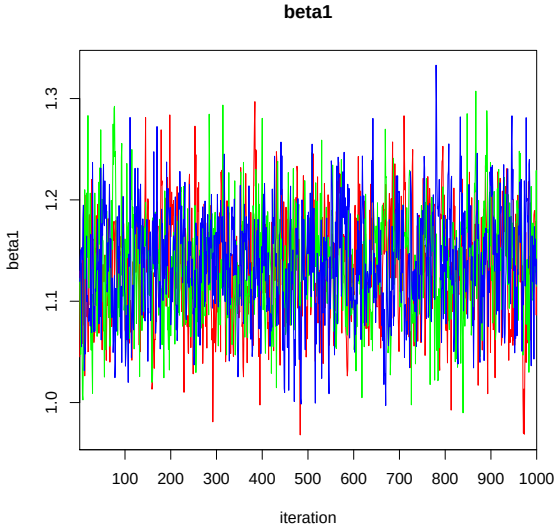
/p9nqyxk5295bnlk0n50mm5ch0000gq/T//RtmpBjvkP8/model439112527836.txt", fit using jags, 3 chains, each with 9000 ite

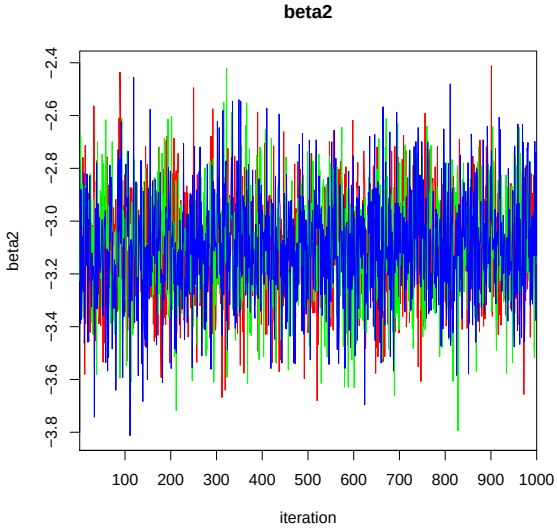


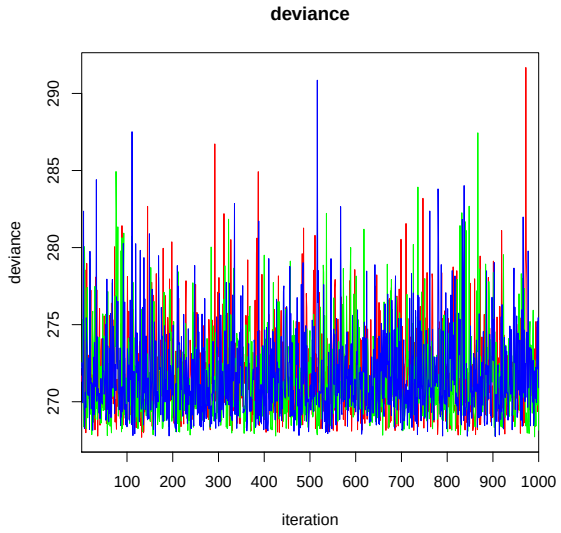



```
> traceplot(bayes.mod.fit.R2jags)
```


alpha
alpha
2.5
2.0
1.5
1.0
100 200 300 400 500 600 700 800 900 1000
iteration


beta1
beta1
1.3
1.2
1.1
1.0
100 200 300 400 500 600 700 800 900 1000
iteration


beta2
beta2
−2.4
−2.6
−2.8
−3.0
−3.2
−3.4
−3.6
−3.8
100 200 300 400 500 600 700 800 900 1000
iteration



If you want to print and save the plot, you can use the following set of commands; adjust the file path and file name for your desired output:

- `> pdf("~/Documents/Dropbox/Uni/9 - ICPSR/2017/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.R2jags)`

  ... 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"`.

6 Fitting Bayesian models 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:

[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.

```
> 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.fit.runjags <- 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 4.2.0 on Wed Jul  5 20:35:16 2017
## 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 information:
##    Observed stochastic nodes: 100
##    Unobserved stochastic nodes: 4
##    Total graph size: 516
## . 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_3' 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 at `?run.jags` for more information.

Finally, you can summarize the model:

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

##
## JAGS model summary statistics from 15000 samples (chains = 3; adapt+burnin = 2000):
##
##       Lower95  Median Upper95    Mean       SD Mode     MCerr MC%ofSD SSeff
## alpha  1.2656  1.8461  2.4169   1.844  0.29529   --  0.011306     3.8   682
## beta1  1.0362  1.1359  1.2444  1.1362 0.052781   -- 0.0020566     3.9   659
## beta2 -3.5118 -3.0943 -2.7031 -3.0938  0.20594   -- 0.0027224     1.3  5723
##
##            AC.10   psrf
## alpha    0.39505 1.0013
## beta1    0.41196 1.0017
## beta2 -0.0085609 1.0004
##
## Total time taken: 3.8 seconds
```

7 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.txt", chain(1)
inits in "bayes.mod.inits2.txt", chain(2)
inits in "bayes.mod.inits2.txt", 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 *
```

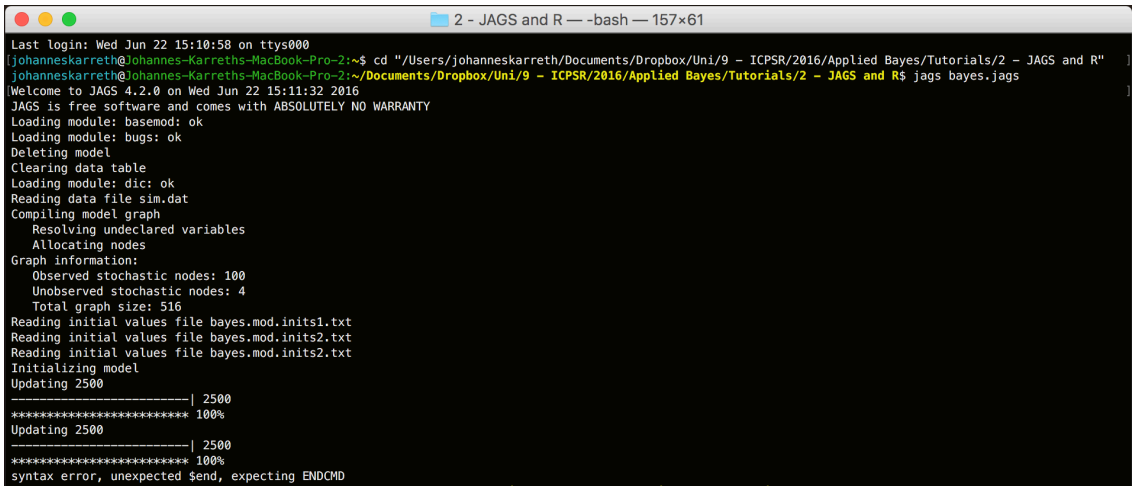
The following instructions are for the Terminal on the Mac, but you can reproduce the same steps on Windows or Linux. You run this script file by opening a Terminal window, changing to the working directory in which all the above files are located (adjust to your own WD)

```
cd "/Users/johanneskarreth/Documents/Dropbox/Uni/9 - ICPSR/2017/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:

```
> 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.txt", chain(1)
  inits in "bayes.mod.inits2.txt", chain(2)
  inits in "bayes.mod.inits.txt3", chain(3)
  ```

Use the starting values you provide in three newly created text files `bayes.mod.inits1.txt`, `bayes.mod.inits2.txt`, and `bayes.mod.inits3.txt`, 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.

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

- ```
  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 later in this tutorial.

8 Installing WinBUGS/OpenBUGS (Windows only)

These instructions are for Windows users.[4] Mac users wishing to install WinBUGS or OpenBUGS on a Mac computer: please visit my website `www.jkarreth.net/bayes-icpsr-html` for detailed instructions and come see me in office hours for more information.

8.1 WinBUGS

1. Download and install `WinBUGS14.exe` from `http://www.mrc-bsu.cam.ac.uk/software/bugs/the-bugs-project-winbugs/`. Be sure to install WinBUGS in your user's profile directory (e.g. `/Documents/`) to avoid problems with user rights.

2. Download and install the patch for Version 1.4.3. You can do this simply by opening the file `WinBUGS14_cumulative_patch_No3_06_08_07_RELEASE.txt` from within WinBUGS and following the instructions at the top.

3. Download and install the free key for unrestricted use, again by simply opening the file `WinBUGS14_immortality_key.txt` from within WinBUGS and following the instructions at the top.

8.2 OpenBUGS

Download and install `OpenBUGS323setup.exe.exe` from `http://www.openbugs.net/w/Downloads`. Be sure to install OpenBUGS in your user's profile directory (e.g. `C:/Users/Johannes/`) to avoid problems with user rights.

9 Fitting Bayesian models using R2WinBUGS/R2OpenBUGS (Windows only)

These instructions are for Windows users. Mac users wishing to use R2WinBUGS or R2OpenBUGS on a Mac computer: please visit my website `www.jkarreth.net/bayes-icpsr-html` for detailed instructions and/or come see me in office hours for more information.

R2WinBUGS and R2OpenBUGS are very similar; I discuss them together in this section and point out differences where applicable. You should settle for using either of the two packages; OpenBUGS and R2OpenBUGS are slightly more up to date, and I recommend going with this option.

Once you have installed WinBUGS or OpenBUGS on your computer (previous section), install the respective package from within R or RStudio, via the Package Installer, or by using

```
> install.packages("R2WinBUGS", dependencies = TRUE, repos = "https://cloud.r-project.org")
> install.packages("R2OpenBUGS", dependencies = TRUE, repos = "https://cloud.r-project.org")
```

[4] Some of this content borrows from a tutorial that Christopher Hare, former TA for this workshop, compiled in 2014.
```

9.1 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)
```

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

```
> setwd("C:/Users/Johannes/Bayes/Lab2")
```

The model looks just like the JAGS/BUGS models shown 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 writing the model in a text editor, you can also enter it in your R script:

```
> sink("bayesmod.txt")
> cat("
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)

}
", fill=TRUE)
> sink()
```

Now define the vectors of the data matrix for BUGS:

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

Read in the data frame for BUGS:

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

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

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

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

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

Now, you need to define the starting values for BUGS. 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 distribution 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))
+   }
```

Note: Here, I did not specify a starting value for the node `tau`. This will lead JAGS (or BUGS) to generate a random number as a starting value for `tau`. In general, any node for which you do not explicitly generate starting values will receive a random starting value. This is not a problem computationally, but undesirable from a reproducibility perspective. (More on this later in this workshop.)

Before using R2WinBUGS the first time, you need to load the package, and you might need to set a random number seed. To do this, type

```
> library("R2WinBUGS")
> set.seed(123)
```

or alternatively

```
> library("R2OpenBUGS")
> 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.

9.2 Fitting the model

You are now ready to use the `bugs()` function, which calls WinBUGS or OpenBUGS, depending on which package is loaded.

9.2.1 R2WinBUGS

I'm giving the resulting object the unwieldy name `bayes.mod.fit.R2WinBUGS` here to distinguish it from other objects later. Additionally, you have to specify the location of the model file, the data, the parameters, the initial values, as well as how many chains you want to fit and how long you want to run them. Finally, you need to specify the location of the WinBUGS directory (where you installed WinBUGS):

```
> bayes.mod.fit.R2WinBUGS <- bugs(model.file = "bayes.mod",
+   data = sim.dat.bugs,
+   parameters.to.save = bayes.mod.params,
+   inits = bayes.mod.inits,
+   n.chains = 3,
+   n.iter = 5000,
+   n.burnin = 1000,
+   n.thin = 1,
+   bugs.directory = "C:/Users/Johannes/WinBUGS14/")
```

Running WinBUGS from R offers seamless, and therefore quick, access to results after fitting a model. Try for yourself:

```
> print(bayes.mod.fit.R2WinBUGS)
> plot(bayes.mod.fit.R2WinBUGS)
```

The next section offers details on obtaining convergence diagnostics from this model output.

9.2.2 R2OpenBUGS

I'm giving the resulting object the unwieldy name `bayes.mod.fit.R2OpenBUGS` here to distinguish it from other objects later. Additionally, you have to specify the location of the model file, the data, the parameters, the initial values, as well as how many chains you want to fit and how long you want to run them. Finally, you need to specify the location of the OpenBUGS directory (where you installed OpenBUGS):

```
> bayes.mod.fit.R2OpenBUGS <- bugs(model.file = "bayes.mod",
+   data = sim.dat.bugs,
+   parameters.to.save = bayes.mod.params,
+   inits = bayes.mod.inits,
+   n.chains = 3,
+   n.iter = 5000,
+   n.burnin = 1000,
+   n.thin = 1)
```

Running OpenBUGS from R offers seamless, and therefore quick, access to results after fitting a model. Try for yourself:

```
> print(bayes.mod.fit.R2OpenBUGS)
> plot(bayes.mod.fit.R2OpenBUGS)
```

The next section offers details on obtaining convergence diagnostics from this model output.

10 Convergence diagnostics

R offers a variety of solutions to obtain convergence diagnostics. As a best practice, you should convert your model output (whether it comes from JAGS, R2Jags, runjags, R2WinBUGS, R2OpenBUGS, or other programs) into an MCMC object. MCMC objects are a separate class of R objects that contain one or multiple Markov Chains and the respective information about iterations etc. that is needed to conduct convergence diagnostics.

You can generate most model outputs into an MCMC object for analysis with this command:

- For objects from R2jags, use `as.mcmc()`:

```
> bayes.mod.fit.mcmc <- as.mcmc(bayes.mod.fit.R2jags)
> summary(bayes.mod.fit.mcmc)


##
## 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      1.846 0.29211 0.0053333       0.009263
```

```
## beta1      1.136 0.05219 0.0009529       0.001646
## beta2     -3.095 0.20611 0.0037631       0.003764
## deviance 271.687 2.89458 0.0528476       0.056665
##
## 2. Quantiles for each variable:
##
##             2.5%     25%     50%     75%   97.5%
## alpha      1.279   1.650   1.849   2.046   2.409
## beta1      1.038   1.099   1.137   1.171   1.238
## beta2     -3.516  -3.235  -3.088  -2.958  -2.699
## deviance 268.155 269.618 270.959 273.031 278.935
```

- For objects from runjags, R2WinBUGS, R2OpenBUGS, use `as.mcmc.list()`:

```
> bayes.mod.fit.mcmc <- as.mcmc.list(bayes.mod.fit.runjags)
> summary(bayes.mod.fit.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  1.844 0.29529 0.002411       0.011374
## beta1  1.136 0.05278 0.000431       0.002058
## beta2 -3.094 0.20594 0.001682       0.002725
##
## 2. Quantiles for each variable:
##
##         2.5%    25%    50%    75%  97.5%
## alpha  1.264  1.649  1.846  2.044  2.415
## beta1  1.033  1.101  1.136  1.171  1.242
## beta2 -3.501 -3.229 -3.094 -2.957 -2.690
```

- If you used JAGS or WinBUGS/OpenBUGS outside of R, the MCMC output is stored in text files and you need to read these text files into R.
  - 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 and what their names are.
  - In R, load the coda package:

    ```
    > library("coda")
    ```

  - Read in your JAGS output. This requires that the chain and index files (see above) are in your working directory (be sure to use your own WD in the code below). Above, I gave these file names the stem `bayes_out`.

    ```
    > setwd("~/Documents/Dropbox/Uni/9 - ICPSR/2017/Applied Bayes/Tutorials/2 - JAGS and R/")
    > chain1 <- read.coda(output.file = "bayes_outchain1.txt",
    +   index.file = "bayes_outindex.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 = "bayes_outchain2.txt",
    +   index.file = "bayes_outindex.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 = "bayes_outchain3.txt",
    +   index.file = "bayes_outindex.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 proceed as before:

```
> 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      1.837 0.2782 0.0045424       0.014366
## beta1      1.137 0.0498 0.0008132       0.002529
## beta2     -3.093 0.2121 0.0034641       0.004357
## deviance 271.723 2.8246 0.0461252       0.063986
##
## 2. Quantiles for each variable:
##
##             2.5%     25%     50%     75%   97.5%
## alpha      1.304   1.644   1.837   2.020   2.388
## beta1      1.036   1.105   1.136   1.171   1.234
## beta2     -3.507  -3.240  -3.098  -2.947  -2.670
## deviance 268.162 269.649 271.123 273.126 278.630
```

With an MCMC object, you can use a variety of commands for diagnostics and presentation using the coda package (Plummer et al., 2006):

- Plot:

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

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

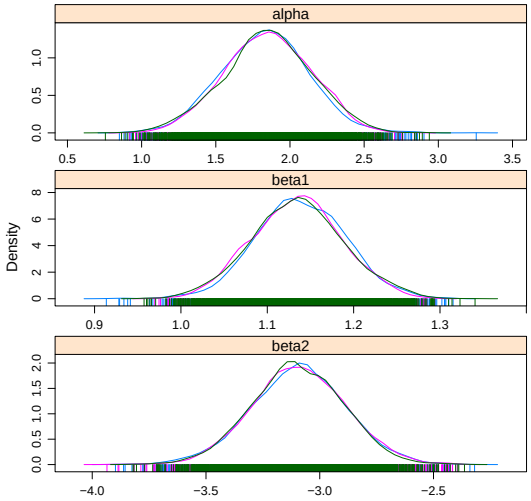
```
> xyplot(bayes.mod.fit.mcmc, layout = c(1, 3), aspect = "fill")
```

- Density plot:

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

... with minor modifications for more effective display:

```
> densityplot(bayes.mod.fit.mcmc, layout = c(1, 3), as.table = TRUE, aspect = "fill")
```



- Trace- and density in one plot, printed directly to your working directory (adjust to your WD when using this code):

```
> pdf("~/Documents/Dropbox/Uni/9 - ICPSR/2017/Applied Bayes/Tutorials/2 - JAGS and R/bayes_fit_mcmc_plot.pdf")
> plot(bayes.mod.fit.mcmc)
> dev.off()
```

- Autocorrelation plot, printed directly to your working directory (adjust to your WD when using this code):

```
> pdf("~/Documents/Dropbox/Uni/9 - ICPSR/2017/Applied Bayes/Tutorials/2 - JAGS and R/bayes_fit_mcmc_autocorr.pdf")
> autocorr.plot(bayes.mod.fit.mcmc, ask = FALSE)
> 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
##  1.987 -1.737 -1.136
##
##
## [[2]]
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##    alpha    beta1    beta2
## -0.44397  0.44147  0.01291
##
##
## [[3]]
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##   alpha   beta1   beta2
## -1.4186  1.6946 -0.9482

> raftery.diag(bayes.mod.fit.mcmc)
```

```
## [[1]]
##
## Quantile (q) = 0.025
## Accuracy (r) = +/- 0.005
## Probability (s) = 0.95
##
##        Burn-in  Total Lower bound  Dependence
##        (M)      (N)   (Nmin)       factor (I)
##  alpha 22       26606 3746         7.10
##  beta1 24       27140 3746         7.25
##  beta2 6        7131  3746         1.90
##
##
## [[2]]
##
## Quantile (q) = 0.025
## Accuracy (r) = +/- 0.005
## Probability (s) = 0.95
##
##        Burn-in  Total Lower bound  Dependence
##        (M)      (N)   (Nmin)       factor (I)
##  alpha 12       14188 3746         3.79
##  beta1 16       19516 3746         5.21
##  beta2 6        6878  3746         1.84
##
##
## [[3]]
##
## Quantile (q) = 0.025
## Accuracy (r) = +/- 0.005
## Probability (s) = 0.95
##
##        Burn-in  Total Lower bound  Dependence
##        (M)      (N)   (Nmin)       factor (I)
##  alpha 22       21906 3746         5.85
##  beta1 24       25338 3746         6.76
##  beta2 6        6637  3746         1.77
```

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

```
## [[1]]
##
##       Stationarity start     p-value
##       test         iteration
## alpha passed       501       0.339
## beta1 passed       501       0.331
## beta2 passed         1       0.168
##
##       Halfwidth Mean  Halfwidth
##       test
## alpha passed     1.81 0.03522
## beta1 passed     1.14 0.00597
## beta2 passed    -3.09 0.00904
##
## [[2]]
##
##       Stationarity start     p-value
##       test         iteration
## alpha passed       1         0.175
## beta1 passed       1         0.232
## beta2 passed       1         0.458
##
##       Halfwidth Mean  Halfwidth
##       test
## alpha passed     1.85 0.03632
## beta1 passed     1.13 0.00686
## beta2 passed    -3.09 0.00955
##
## [[3]]
##
##       Stationarity start     p-value
##       test         iteration
## alpha passed       1         0.301
```

```
## beta1 passed       1         0.190
## beta2 passed       1         0.310
##
##       Halfwidth Mean  Halfwidth
##       test
## alpha passed     1.85 0.04211
## beta1 passed     1.14 0.00734
## beta2 passed    -3.10 0.00916
```

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 = "https://cloud.r-project.org")
```

- Then, load it:

```
> library("superdiag")
```

- Next, call the `superdiag` function and specify how many iterations of the chain you want to discard before analyzing for convergence:

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


## Number of chains = 3
## Number of iterations = 5000 per chain before discarding the burn-in period
## The burn-in period = 1000 per chain
## Sample size in total = 12000
##
## ********** The Geweke diagnostic: **********
## Z-scores:
##                       chain1    chain 2    chain 3
## alpha              1.1646279  7.6021292 -0.1277970
## beta1             -1.1916455 -9.4937646  0.2442938
## beta2             -0.2395671  0.8658903 -1.0377795
## Window From Start  0.1000000  0.9286500  0.4517600
## Window From Stop   0.5000000  0.0008600  0.4218100
##
## ********** The Gelman-Rubin diagnostic: **********
## Potential scale reduction factors:
##
##       Point est. Upper C.I.
## alpha       1.02       1.07
## beta1       1.02       1.08
## beta2       1.00       1.00
##
## Multivariate psrf
##
## 1.02
##
## ********** The Heidelberger-Welch diagnostic: **********
##
## Chain 1, epsilon=0.1, alpha=0.05
##       Stationarity start     p-value
##       test         iteration
## alpha passed       1         0.469
## beta1 passed       1         0.555
## beta2 passed       1         0.284
##
##       Halfwidth Mean  Halfwidth
##       test
## alpha passed     1.81 0.03682
## beta1 passed     1.14 0.00647
## beta2 passed    -3.09 0.01060
##
```

```
## Chain 2, epsilon=0.142, alpha=0.025
##       Stationarity start     p-value
##       test         iteration
## alpha passed       1         0.0591
## beta1 passed       1         0.0942
## beta2 passed       1         0.3974
##
##       Halfwidth Mean  Halfwidth
##       test
## alpha passed     1.85 0.04001
## beta1 passed     1.13 0.00777
## beta2 passed    -3.09 0.01064
##
## Chain 3, epsilon=0.121, alpha=0.025
##       Stationarity start     p-value
##       test         iteration
## alpha passed       1         0.395
## beta1 passed       1         0.328
## beta2 passed       1         0.351
##
##       Halfwidth Mean  Halfwidth
##       test
## alpha passed     1.86 0.04759
## beta1 passed     1.13 0.00838
## beta2 passed    -3.10 0.01020
##
## ********** The Raftery-Lewis diagnostic: **********
##
## Chain 1, converge.eps = 0.001
## Quantile (q) = 0.025
## Accuracy (r) = +/- 0.005
## Probability (s) = 0.95
##
##        Burn-in  Total Lower bound  Dependence
##        (M)      (N)   (Nmin)       factor (I)
##  alpha 22       26986 3746         7.20
##  beta1 18       18688 3746         4.99
##  beta2 6        6848  3746         1.83
##
##
## Chain 2, converge.eps = 2e-04
## Quantile (q) = 0.05
## Accuracy (r) = +/- 5e-04
## Probability (s) = 0.975
##
## You need a sample size of at least 954539 with these values of q, r and s
##
## Chain 3, converge.eps = 0.001
## Quantile (q) = 0.001
## Accuracy (r) = +/- 0.0025
## Probability (s) = 0.975
##
##        Burn-in  Total Lower bound  Dependence
##        (M)      (N)   (Nmin)       factor (I)
##  alpha 10       2417  804          3.01
##  beta1 10       2417  804          3.01
##  beta2 5        1342  804          1.67
```

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 = "https://cloud.r-project.org")
```

- Then, load it:

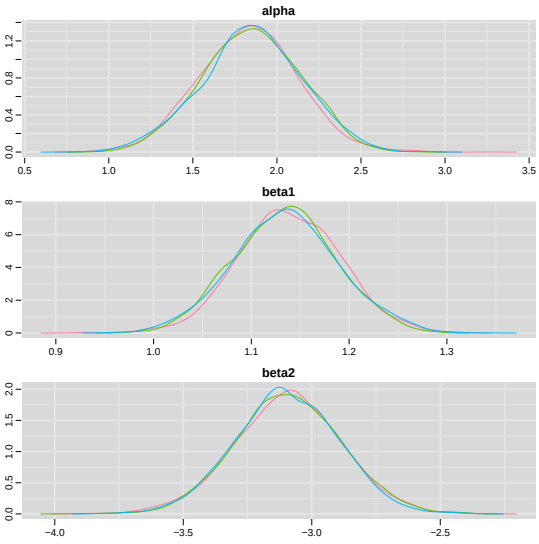
```
> library("mcmcplots")
```

- 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.

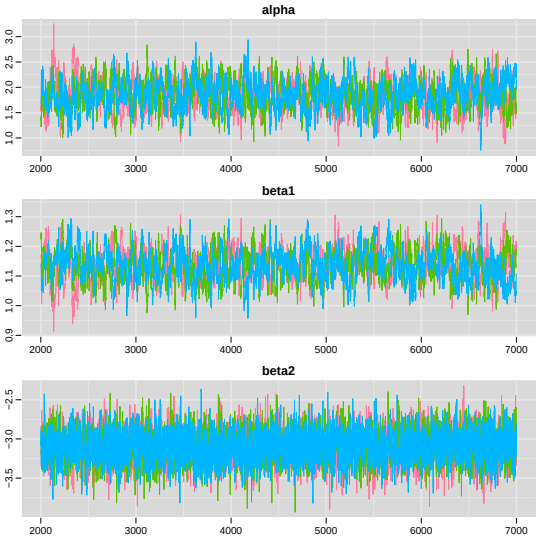
```
> mcmcplot(bayes.mod.fit.mcmc)
```

- Some commands for individual plots:

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



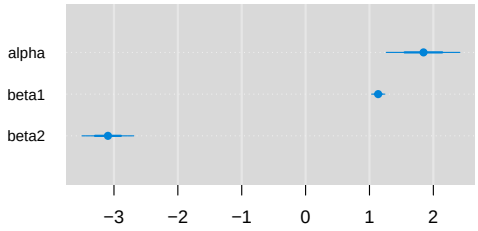

```
> traplot(bayes.mod.fit.mcmc, parms = c("alpha", "beta1", "beta2"))
```



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

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

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



More plots: ggmcmc

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

- First, install the package:

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

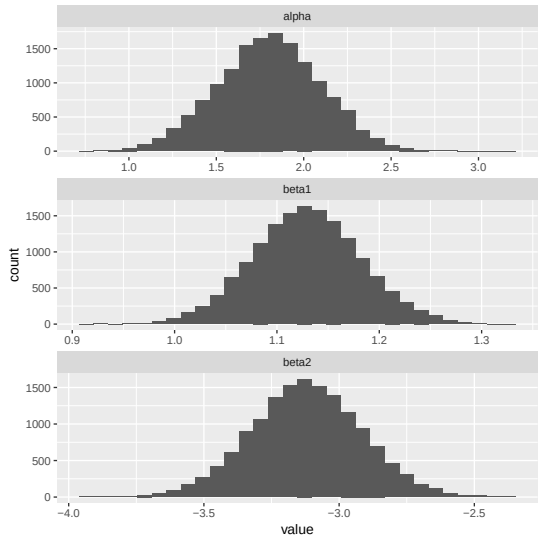
- Then, load it:

```
> library("ggmcmc")
```

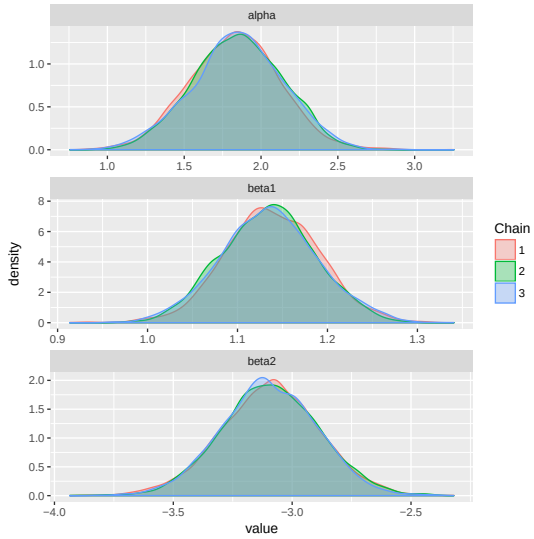
- To use this package, you need to first convert your MCMC object into a ggs object, using the ggs function. This object can then be passed on to the various ggmcmc plotting commands:

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

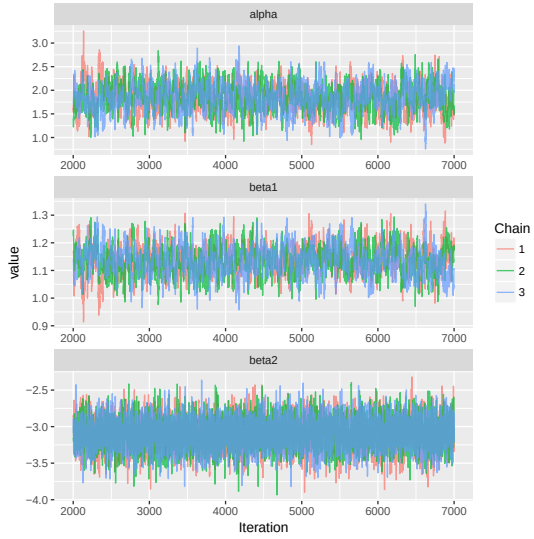
```
> ggs_histogram(bayes.mod.fit.gg)
```



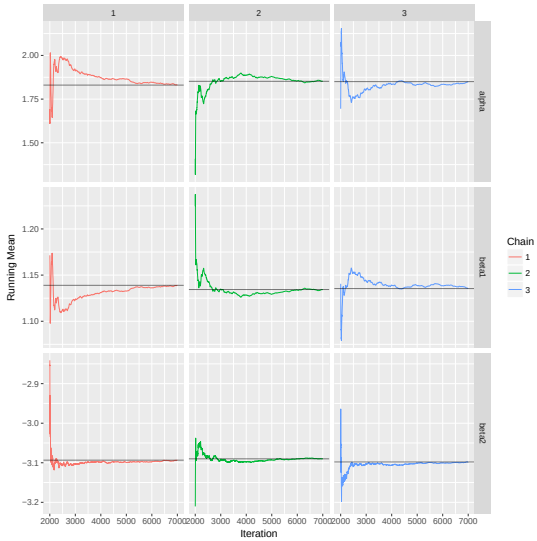

```
> ggs_density(bayes.mod.fit.gg)
```



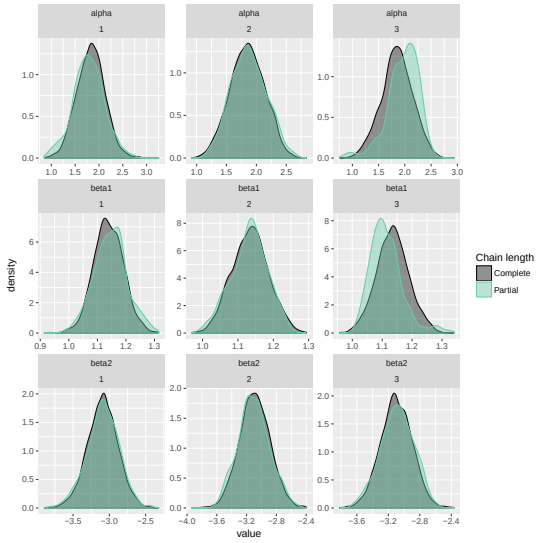

```
> ggs_traceplot(bayes.mod.fit.gg)
```



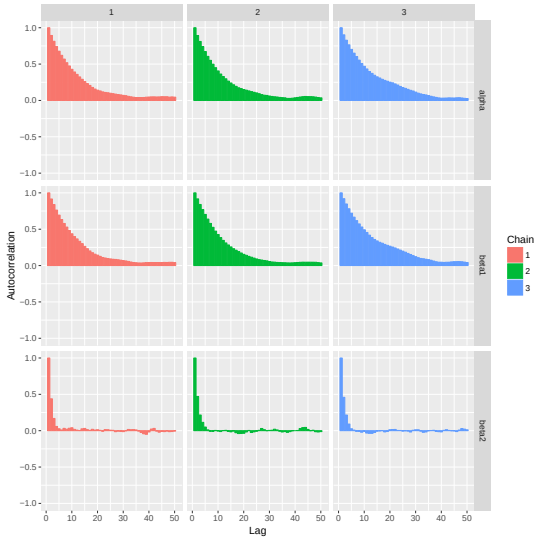

```
> ggs_running(bayes.mod.fit.gg)
```



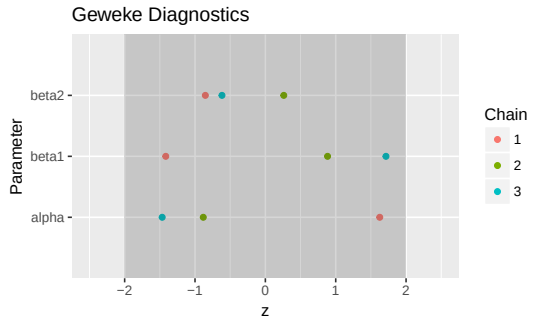
```
> ggs_compare_partial(bayes.mod.fit.gg)
```



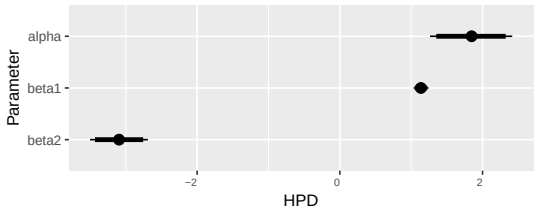

```
> ggs_autocorrelation(bayes.mod.fit.gg)
```




```
> ggs_geweke(bayes.mod.fit.gg)
```




```
> ggs_caterpillar(bayes.mod.fit.gg)
```



- Or, you can use the ggmcmc command to create a PDF file containing a variety of diagnostic plots (adjust to your WD and desired file names when using this code):

```
> ggmcmc(bayes.mod.fit.gg,
+        file = "~/Documents/Dropbox/Uni/9 - ICPSR/2017/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
## Time taken to generate the report: 7.8 seconds.
```

11 MCMCpack

MCMCpack (Martin, Quinn, and Park, 2011) is an R package that we will also briefly use in this workshop It is as easy to use as the `lm()` command in R and produces the same MCMC output you analyzed above, and it is quite fast. One major downside of MCMCpack is that it does not offer many options for model specification as writing your full model in JAGS or BUGS. For more information on MCMCpack, see `http://mcmcpack.wustl.edu/`. Because MCMCpack uses an R-like model 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 you 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.fit.MCMCpack <- 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.MCMCpack)

##
## Iterations = 1001:6000
## Thinning interval = 1
## Number of chains = 1
## 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
## (Intercept)  1.8366 0.28842 0.0040789       0.003941
## x1           1.1367 0.05165 0.0007305       0.000705
## x2          -3.0790 0.21073 0.0029801       0.002980
## sigma2       0.8969 0.13020 0.0018413       0.001853
##
## 2. Quantiles for each variable:
##
##                2.5%     25%     50%     75%  97.5%
## (Intercept)  1.2756  1.6407  1.8364  2.0274  2.414
## x1           1.0343  1.1022  1.1373  1.1722  1.236
## x2          -3.4835 -3.2204 -3.0802 -2.9389 -2.657
## sigma2       0.6789  0.8026  0.8859  0.9805  1.186
```

The resulting object is already an MCMC object. 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.fit.MCMCpack)
> superdiag(bayes.mod.fit.MCMCpack, burnin = 1000)
```

12 Following this course using JAGS

JAGS and BUGS course are very similar, with some minor exceptions. 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/bayes-icpsr.html`.

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

13 Other software solutions for Bayesian estimation

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. 2016. "runjags: An R package providing interface utilities, parallel computing methods and additional distributions for MCMC models in JAGS." *Journal of Statistical Software* 71 (9): 1–25.

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.

Lunn, David, David Spiegelhalter, Andrew Thomas, and Nicky Best. 2009. "The BUGS project: Evolution, critique and future directions." *Statistics in Medicine* 28 (25): 3049–3067.

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*

Plummer, Martyn, Nicky Best, Kate Cowles, and Karen Vines. 2006. "CODA: Convergence Diagnosis and Output Analysis for MCMC." *R News* 6 (1): 7–11.

Spiegelhalter, David J., Andrew Thomas, Nicky G. Best, and Dave Lunn. 2003. "WinBUGS Version 1.4 User Manual.".

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*

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*