JAGS for Rats

Jonathan Rougier[*]
School of Mathematics
University of Bristol UK

Version 2.0, compiled February 23, 2017

1 Introduction

This is a worksheet about JAGS, using rjags. These must both be installed on your computer. Go to `http://mcmc-jags.sourceforge.net/` to find out about and download JAGS. Once installed, you need to install the R package rjags, by doing the following once only:

```
#### install rjags package (once only)

install.packages("rjags")
```

If everything works smoothly then you should be able to execute the following code and get the following output (some of your numbers ought to be slightly different):

```
library(rjags)

## Loading required package:  coda
## Linked to JAGS 4.0.1
## Loaded modules:  basemod,bugs,dic

example(jags.samples)

##
## jgs.sm>   data(LINE)
##
```

[*]`j.c.rougier@bristol.ac.uk`

```
## jgs.sm>   LINE$recompile()
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 5
##    Unobserved stochastic nodes: 3
##    Total graph size: 40
##
## Initializing model
##
##
## jgs.sm>   LINE.samples <- jags.samples(LINE, c("alpha","beta","sigma"),
## jgs.sm+   n.iter=1000)
##
## jgs.sm>   LINE.samples
## $alpha
## mcarray:
## [1] 3.006265
##
## Marginalizing over: iteration(1000),chain(2)
##
## $beta
## mcarray:
## [1] 0.7949662
##
## Marginalizing over: iteration(1000),chain(2)
##
## $sigma
## mcarray:
## [1] 1.006746
##
## Marginalizing over: iteration(1000),chain(2)
```

I suppressed the progress bar for rjags, using `options(jags.pb = "none")`, because it messes up this document, but you might prefer to keep it.

2 Setting up your computer

Don't ignore this section! Unless you get into good habits early on, your computer will rapidly become a graveyard of unattributed files. Put all of your R projects in a common place. For example, in a folder called Rprojects.

Inside this folder create a new folder for each application. We are going to analyse the Rats data, so Rats would be a good name for this folder. When you are in R, make sure you issue the command

```
setwd("~/Rprojects/Rats")
```

or something similar, so that R knows where everything is, and everything goes; this is your 'working directory'. If you ever want to know what your working directory is from inside R, type `getwd()`.

The first thing to do is to put the data-files into your working directory. You can download the two files Rats.xls and Rats.csv from the course webpage at `http://www.maths.bris.ac.uk/~mazjcr/BMB/2016/home.html`. We will be using Rats.csv but I included Rats.xls so you can practice exporting spreadsheet data in CSV format as UTF-8 encoding. To do this you need to open the data in Open Office Calc or Google Sheets, and export it from that—Microsoft do not want you to use a non-proprietary encoding.

A comment on these data. The original digital source of these data is `http://www.openbugs.net/Examples/Rats.html`. Unfortunately these data are wrongly-represented as an R object—see if you can figure out why. I discovered this in the `matplot` below. Thank goodness I always visualize my data once I have loaded it! The original paper source is the Gelfand et al paper listed at the top of the data-file. When I checked the digital source against the original paper source I found another error. And the moral of this story? Data you download from the Internet are not reliable; it is *you* who are at fault if you fail to check them.

Next, you need to open a new file which will contain all of your R commands for the session. *Never* just type stuff at the > prompt, unless you are checking something or trying something out. Create a file called Rats1.R in Rprojects/Rats, open it with a simple text editor (*not* Word for Windows!) and start it off with a helpful summary, such as

```
#### Rats1.R

## first go at analysing the Rats data using rjags.

library(rjags)
setwd("~/Rproject/Rats")
```

All the lines of code below go into Rats1.R, as a permanent record of the steps you have taken in your analysis. This way you can replicate the analysis exactly later on, just by doing `source("Rats1.R")`. And you can come back later on and improve the analysis, by editing the file.

In the meantime, you can cut-and-paste from Rats1.R to the R prompt. Software such as RStudio makes this very easy.

On we go. Let's see what CSV files are in my working directory.

```
list.files(pattern = "csv$")

## [1] "Rats.csv"
```

If you want to know more about how the pattern is specified, have a look at the help-file `?list.files` and then at `help("regex", package = "base")`. You will see there that '`$`' matches the end of the string, so that '`csv$`' matches any name which ends with `csv`.

Below, you will see that I often use `foo` as a temporary variable name; this is a common programmers' convention, with other popular choices being `bar` and `tmp`.

Now you need to inspect Rats.csv. I would not usually do this within R, but in this case it is straightforward:

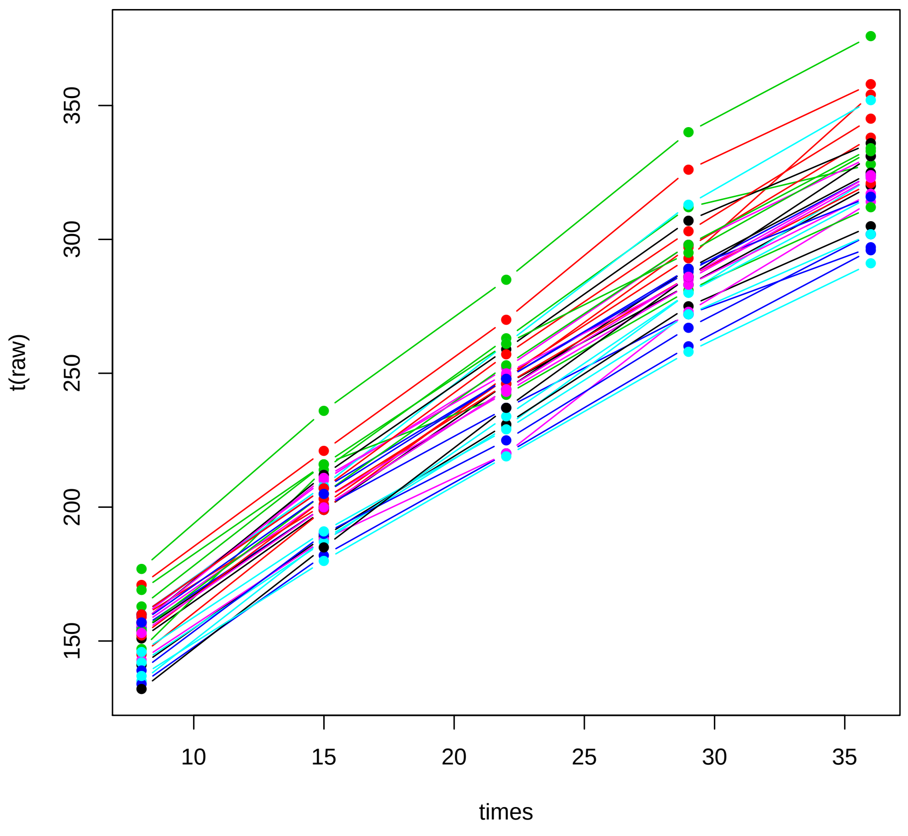
```
foo <- readLines("Rats.csv", encoding  = "UTF-8")
head(foo, 15)

##  [1] "Rats data, originally from Table 3 (p978) of"
##  [2] ""
##  [3] "A.E. Gelfand, S.E. Hills, A. Racine-Poon, and A. Smith, 1990, Illustration
##  [4] "of Bayesian inference in Normal data models using Gibbs sampling, Journal"
##  [5] "of the American Statistical Association, 85, 972-985."
##  [6] ""
##  [7] "Source is http://www.openbugs.net/Examples/Ratsdata.html, with corrections
##  [8] ""
##  [9] "Time t is days from birth, and weight is grams, one row per rat"
## [10] ""
## [11] "t = 8,t = 15,t = 22,t = 29,t = 36"
## [12] "151,199,246,283,320"
## [13] "145,199,249,293,354"
## [14] "147,214,263,312,328"
## [15] "155,200,237,272,297"
```

So the data starts at line 11, with the time values, and then the weights start at line 12. Let's read these both in:

```
## get the times off foo

times <- foo[11]
times <- strsplit(times, ",")[[1]]
times <- gsub("t = ", "", times)
times <- as.numeric(times)
times # ta dah!  But you could have typed them in more easily!

## [1]  8 15 22 29 36

## get the weights out of Rats.csv

raw <- read.csv("Rats.csv", header = FALSE,
  skip = 11, fill = FALSE, encoding  = "UTF-8")
# str(raw)
head(raw)

##    V1  V2  V3  V4  V5
## 1 151 199 246 283 320
## 2 145 199 249 293 354
## 3 147 214 263 312 328
## 4 155 200 237 272 297
## 5 135 188 230 280 323
## 6 159 210 252 298 331
```

And don't forget to check the data visually. I'll plot each rat as one line.

```
## visual check on the results

matplot(times, t(raw), type = "b", pch = 16, lty = 1)
```



That looks OK, but the first time I did it, I could see immediately that there was a problem with the data (see above).

3 Specifying the model

rjags needs the model, expressed in BUGS notation, written in a file. We can do this from within within your Rats1.R file, by wrapping the model in a '`cat("...", file = "Rats.bug")`'.

```
#### write the model out to a file

cat("
#### model for Rats

model {
  for (i in 1:n) {
    for (j in 1:k) {
      Y[i, j] ~ dnorm(alpha + beta[i] * (x[j] - mean(x)), tauy)
    }
    beta[i] ~ dnorm(mub, taub)
  }

  ## flat prior for (alpha, tauy, mub, taub)

  alpha ~ dnorm(0, 0.001^2)
  tauy ~ dgamma(0.001, 0.001)
  mub ~ dnorm(0, 0.001^2)
  taub ~ dgamma(0.001, 0.001)
}
", file = "Rats.bug")
```

If you have a look in your working directory you will now find a file called Rats.bug containing exactly the lines given above.

You can see from above that the BUGS notation is quite simple, and conforms closely to the way that we would express a statistical model in extensive form. One thing to watch out for, though. The scale parameter in a Normal distribution is usually the variance, but in BUGS it is the precision which is the reciprocal of the variance. So in the lines for `alpha` and `mub`, the second parameter indicates a variance of $1/0.001^2$, or 1000^2.

4 Using **rjags**

If you are unclear about how to use rjags, type `?rjags` to get a very helpful message. Here are the steps listed in the message:

1. *Define the model using the BUGS language in a separate file.*

   Done that.

2. *Read in the model file using the 'jags.model' function.*

   But this gives an error:

   ```
   #### set up the JAGS sampler

   ## this causes an error
   ```

```
myrats <- jags.model("Rats.bug")

## Compiling model graph
##    Resolving undeclared variables
## Deleting model

## Error in jags.model("Rats.bug"):  RUNTIME ERROR:
## Compilation error on line 6.
## Unknown variable n
## Either supply values for this variable with the data
## or define it  on the left hand side of a relation.
```

This is because there are objects in Rats.bug which are not random variables: `n`, `k`, and `x`. So these must be supplied through the `data` argument.

```
## this works OK

mydata <- list(n = nrow(raw), k = ncol(raw), x = times)
myrats <- jags.model("Rats.bug", data = mydata)

## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 0
##    Unobserved stochastic nodes: 184
##    Total graph size: 509
##
## Initializing model
```

But what we really want to do is to supply the `Y` values as well. We use `Y = as.matrix(raw)` because `raw` is a dataframe.

```
## this is what we want to do, including the observations

mydata <- list(n = nrow(raw), k = ncol(raw), x = times,
  Y = as.matrix(raw))
myrats <- jags.model("Rats.bug", data = mydata)

## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 150
##    Unobserved stochastic nodes: 34
##    Total graph size: 509
##
## Initializing model
```

Below, I'll suppress the jags commentary using the argument `quiet = TRUE`.

3. *Update the model using the 'update' method for 'jags' objects.*

   I'll cover this below, in the subsection on Assessing convergence (sec. 5.1).

4. *Extract samples from the model object using the 'coda.samples' function.*

   Here is the very simplest example of that, where we just extract α and some of the β quantities.

```
## pull out a sample of the betas

rsam <- coda.samples(myrats,
  c("alpha", "beta[1:3]", "beta[7]"), n.iter = 1000)
head(rsam)

## [[1]]
## Markov Chain Monte Carlo (MCMC) output:
## Start = 1
## End = 7
## Thinning interval = 1
##         alpha  beta[1]  beta[2]  beta[3]  beta[7]
## [1,] 242.3745 5.969256 7.301887 6.601351 5.864765
## [2,] 242.7985 6.141207 7.291924 6.947651 6.036773
## [3,] 243.2772 6.614218 6.763668 5.763482 6.070928
## [4,] 242.0239 6.018950 6.220971 5.929867 6.406937
## [5,] 243.9434 5.960167 6.282002 5.298714 5.470378
## [6,] 241.7870 5.923740 6.283835 5.846657 5.719476
## [7,] 241.4971 6.055372 5.691364 6.125876 5.726264
##
## attr(,"class")
## [1] "mcmc.list"
```

rsam is an mcmc.list object, see ?mcmc.list for more details about how to extract elements from it. Note that the column names identify the random variables, which may be in a different order to the argument.

To see a summary of the random sample do

```
summary(rsam)
```

which I have not included here because there is quite a lot of information (but type it for yourself). See the examples in the next section.

5 Important details

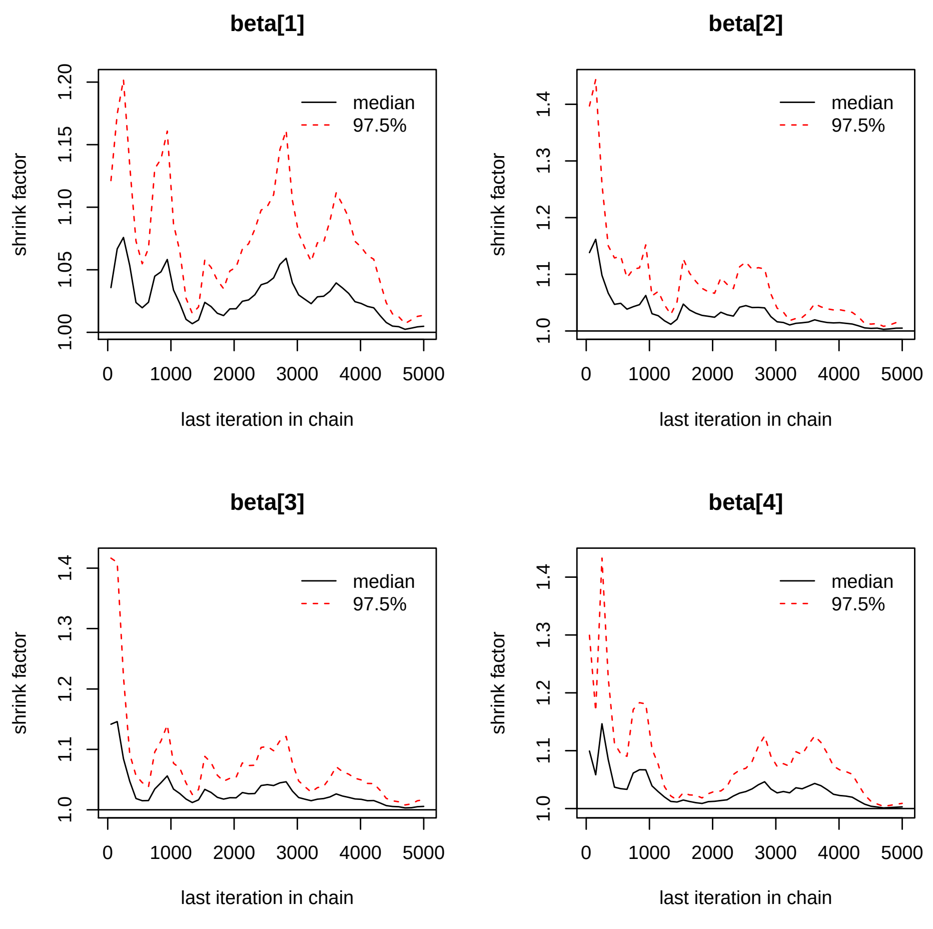
5.1 Assessing convergence

The handout describes the Brooks and Gelman (1998) method for assessing convergence. It is based on running multiple chains, which is specified in the jags.model function using the n.chains argument. My hunch is that the beta variables will converge slowly, so these are the ones I will check (I'll just do the first 4). Here is roughly how we will proceed, subject to one adjustment to be made below.

```
## this is what we want to do, including the observations,
## but this will not work properly quite yet ...

mydata <- list(n = nrow(raw), k = ncol(raw), x = times,
  Y = as.matrix(raw))
myrats <- jags.model("Rats.bug", data = mydata, n.chains = 4,
  quiet = TRUE)
rsam <- coda.samples(myrats, c("beta[1:4]"), n.iter = 2000)

## this test is similar to the test in the handout,
## we want values only a little larger than 1

gelman.diag(rsam)

## Potential scale reduction factors:
##
##          Point est. Upper C.I.
## beta[1]        1.04       1.10
## beta[2]       1.04       1.11
## beta[3]       1.04       1.10
## beta[4]       1.04       1.11
##
## Multivariate psrf
##
## 1.05
```

The reason this is not quite right is because the test requires the initial values for the chain to be over-dispersed with respect to the posterior (target) distribution; JAGS does not do this automatically. One approach is to generate over-dispersed values by sampling from the prior distribution. But because the prior is *very* dispersed, we'll use a slightly concentrated prior by including a few observations.

```
## sample from over-dispersed distribution: 'bar' this has
## only a few observations

bar <- as.matrix(raw)
bar[-(1:3), ] <- NA # just include first three mice

mydata <- list(n = nrow(raw), k = ncol(raw), x = times, Y = bar)
myrats <- jags.model("Rats.bug", data = mydata, n.chains = 4,
  quiet = TRUE)
rsam <- coda.samples(myrats, c("alpha", "tauy", "mub", "taub"),
  n.iter = 10000) # do long run
rsam <- rsam[10000, ] # just keep final value

## now initialise the sampler with these values

inits <- lapply(rsam, as.list) # wants a list of lists
mydata$Y <- as.matrix(raw) # now all the observations
myrats <- jags.model("Rats.bug", data = mydata, n.chains = 4,
  inits = inits, quiet = TRUE)
rsam <- coda.samples(myrats, c("beta[1:4]"), n.iter = 5000)

gelman.diag(rsam)

## Potential scale reduction factors:
##
##         Point est. Upper C.I.
## beta[1]       1.00       1.01
## beta[2]        1.01       1.01
## beta[3]        1.01       1.02
## beta[4]        1.00       1.01
##
## Multivariate psrf
##
## 1.01

gelman.plot(rsam)
```



So it seems as though the chains have converged after 3000 iterations. In future, after each call to `jags.model` we will include the call

```
update(myrats, n.iter = 3000)
```

to do the burn-in.

If you have single or multiple chains, a trace plot is another way assess convergence, by eye. This ought to look like a ‘hairy centipede’. Execute the following command (not run here).

```
## trace plot for convergence by eye

traceplot(rsam)
```

5.2 Monte Carlo Standard Errors (MCSEs)

These are produced automatically by the `summary` method.

```
## run the chain some more and look at the estimates and MCSEs

rsam <- coda.samples(myrats, c("alpha", "beta[1:4]"), n.iter = 1000)
summary(rsam)

##
## Iterations = 5001:6000
## Thinning interval = 1
## Number of chains = 4
## 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    242.604 1.2910 0.020412        0.02084
## beta[1]    6.197 0.1961 0.003101        0.01290
## beta[2]    6.253 0.1969 0.003113        0.01278
## beta[3]    6.213 0.1920 0.003036        0.01157
## beta[4]    6.152 0.2101 0.003321        0.01711
##
## 2. Quantiles for each variable:
##
##             2.5%     25%     50%     75%   97.5%
## alpha    240.038 241.752 242.614 243.462 245.081
## beta[1]    5.792   6.092   6.205   6.316   6.572
## beta[2]    5.874   6.138   6.244   6.359   6.689
## beta[3]    5.830   6.106   6.213   6.329   6.606
## beta[4]    5.671   6.047   6.179   6.285   6.499
```

The ‘Time-series SE’ is similar to the batch means standard error. The latter can be found explicitly using the `batchSE` function:

```
## Estimate MCSEs using batch means

batchSE(rsam)

##      alpha    beta[1]    beta[2]    beta[3]    beta[4]
## 0.01932646 0.01435182 0.01351773 0.01342804 0.01672872
```

If these standard errors are too large for your application, the you need to increase the value of the `n.iter` argument. Usually, `n.iter = 1000` would be a bit low, `n.iter = 1e5` would be more common, or even larger, but that takes too long for a demonstration.

5.3 Monitoring other random quantities

You can only monitor random quantites which are named in the model file. Suppose we want to monitor the mean weight at birth,

$$\bar{y}_0 := \frac{1}{n}\sum_{i=1}^{n}\bigl(\alpha - \beta_i \cdot \bar{x}\bigr).$$

We have to add this to the model. Unfortunately `rjags` does not allow us to specify more than one `model` block, so we have to add the monitor to the original file. Modify the previous code as follows.

```
#### write the model out to a file

cat("
#### model for Rats

model {

  for (i in 1:n) {
    for (j in 1:k) {
      Y[i, j] ~ dnorm(alpha + beta[i] * (x[j] - mean(x)), tauy)
    }
    beta[i] ~ dnorm(mub, taub)
  }

  ## flat prior for (alpha, tauy, mub, taub)
  alpha ~ dnorm(0, 0.001^2)
  tauy ~ dgamma(0.001, 0.001)
  mub ~ dnorm(0, 0.001^2)
  taub ~ dgamma(0.001, 0.001)

  ## add a monitor

  ybar0 <- mean(alpha - beta * mean(x))
}
", file = "Rats.bug")
```

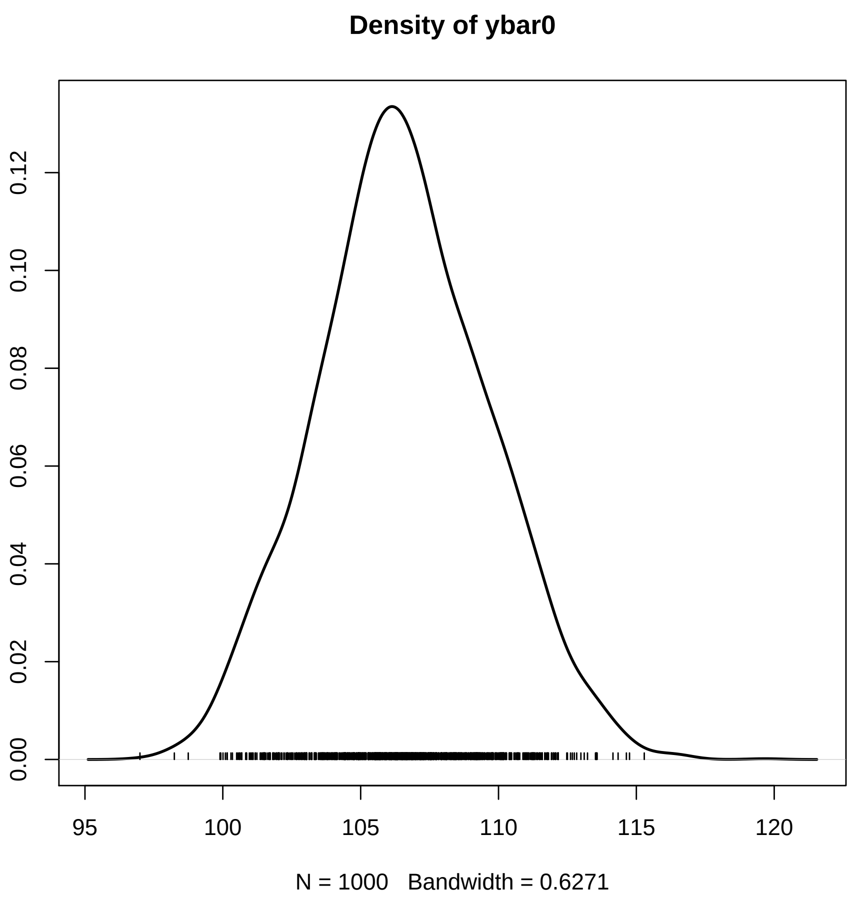
```
## here we re-do the steps above, note the burn-in

myrats <- jags.model("Rats.bug", data = mydata, n.chains = 4,
  quiet = TRUE)
update(myrats, n.iter = 3000) # burn-in
rsam <- coda.samples(myrats, "ybar0", n.iter = 1000)
foo <- summary(rsam)
print(foo)

##
## Iterations = 3001:4000
## Thinning interval = 1
## Number of chains = 4
## 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
##      106.52607        3.11107        0.04919        0.31070
##
## 2. Quantiles for each variable:
##
##  2.5%   25%   50%   75% 97.5%
## 100.6 104.5 106.4 108.6 112.8
```

So the expected mean birthweight is $106.5 \pm 2 \times 0.31$ grams.

6 Other fun stuff

6.1 Visualizing 1D margins

```
#### plot the marginal posterior PDF for ybar0

densplot(rsam, lwd = 2)
```



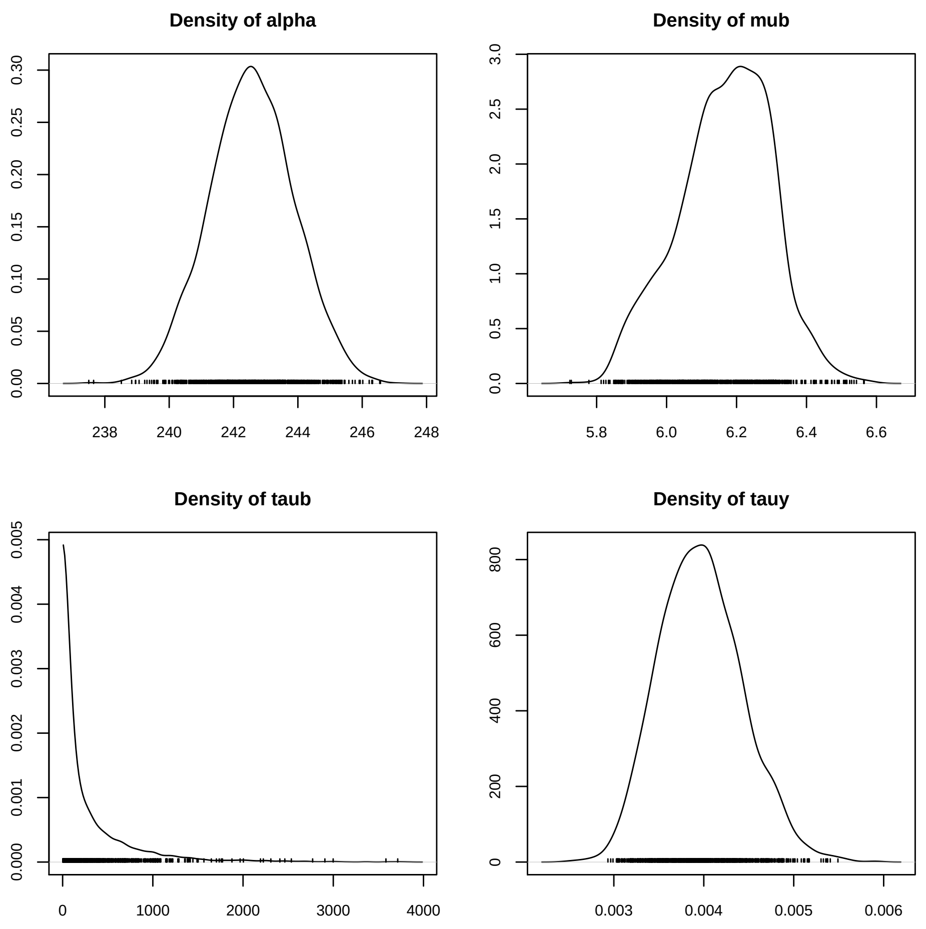
Using `densplot` rather than `plot(density(ybar0))` is convenient because it works directly from objects of class `mcmc.list`, and also it makes a sensible choice of bandwidth. `traceplot` and `densplot` are called together if you type `plot(rsam)`, although you will get a lot of figures, so setting `par(ask = TRUE)` might be a good idea.

We can see from the PDF that the marginal posterior PDF of $\bar{y}_0$ is approximately bell-shaped (very approximately!), and hence approximately

Normal. This is not unusual; in fact, it is a Theorem that, under certain conditons, including lots of observations, the posterior PDF is approximately Multinormal, which implies that the margins are approximately Normal. Note that often these conditions do not hold, but it is always gratifying to see a bell-shaped posterior PDF.

Let's have a look at the posterior densities of the parameters with "flat" prior PDFs:

```r
## densities of the flat parameters

rsam <- coda.samples(myrats, c("alpha", "mub", "taub", "tauy"),
  n.iter = 1000)

op <- par(no.readonly = TRUE)
par(mfrow = c(2, 2), mar = c(3, 3, 3, 1), cex.axis = 0.8, cex.main = 1)
densplot(rsam)
```



```r
par(op) # reset graphics parameters
```

6.2 Choosing between models

I'm just going to show how to compare two models using the Deviance Information Criterion (DIC), not enter into a debate about whether this is a good idea!

First, we need to score the current model. This needs multiple chains.

```r
#### model comparison for taub --> infty

## collect DIC values for the current model

myrats <- jags.model("Rats.bug", data = mydata, n.chains = 4,
  quiet = TRUE)
update(myrats, n.iter = 3000)
dic1 <- dic.samples(myrats, n.iter = 1000, type = "pD")
dic1

## Mean deviance:  1256
## penalty 4.351
## Penalized deviance: 1260
```

Now we need to create a second model in which $\tau_\beta \to \infty$, which means that each β_i is replaced by μ_β.

```
## write the new model out to a file

cat("
#### new model for Rats has taub --> infty

model {

  for (i in 1:n) {
    for (j in 1:k) {
      Y[i, j] ~ dnorm(alpha + mub * (x[j] - mean(x)), tauy)
    }
  }

  ## flat prior for (alpha, tauy, mub)

  alpha ~ dnorm(0, 0.001^2)
  tauy ~ dgamma(0.001, 0.001)
  mub ~ dnorm(0, 0.001^2)
}
", file = "Rats0.bug")
```

```
## collect DIC from new model

myrats0 <- jags.model("Rats0.bug", data = mydata, n.chains = 4,
  quiet = TRUE)
update(myrats0, n.iter = 3000)
dic0 <- dic.samples(myrats0, n.iter = 1000, type = "pD")
dic0

## Mean deviance:  1256
## penalty 3.027
## Penalized deviance: 1259
```

Now we compare the two DICs. A small value is better, so if `dic1 - dic0` is negative, then the original model with a β_i for each rat is better, and if it is positive then the new model with a common β for all rats is better.

```
## compare the DICs

diffdic(dic1, dic0) # could also do dic1 - dic0

## Difference: 0.9707562
## Sample standard error: 0.6583179
```

This is positive, and so it looks as though the model with a common β is better. Looking at the DIC values, the common-β model is achieving the same quality of fit but with a smaller effective number of parameters. This gives it a better (lower) score because overfitting is a source of prediction error.

6.3 Controlling the initialisation values

For fully-reproducible research it is necessary to specify the initial value of the Markov chain and the random seed used to initialise the random nunbers used to simulate the Markov chain. Otherwise these are selected by the software, with the random seed being selected in a pseudo-random manner, and varying from run to run. You do this using the `inits` argument to `jags.model`, see 'Initialization' in `?jags.model` for more details. (We already did some initialization in the convergence test.)

Initialisation values for the random quantities can be collected directly from the model. These do not have to be 'correct', merely not-invalid. The `update` function takes care of moving the sequence into the correct part of the state space. Having said that, better initalisation values mean better-tuned proposals in the non-Gibbs proposals, so perhaps one could do better than my choices below.

I'll use the original model (`Rats.bug`), to show how to initialise a vector of random quantities. This just has one chain.

```
#### control the initialisation values

inits <- list(
  alpha = 0,
  tauy = 1,
  mub = 0,
  taub = 50,
  .RNG.name = "base::Wichmann-Hill", # this is my usual choice,
  .RNG.seed = 101                    # any integer will do
)

myrats <- jags.model("Rats.bug", data = mydata, inits = inits,
  quiet = TRUE)
update(myrats, n.iter = 3000)
rsam <- coda.samples(myrats, "ybar0", n.iter = 1000)
head(rsam)

## [[1]]
## Markov Chain Monte Carlo (MCMC) output:
## Start = 3001
## End = 3007
## Thinning interval = 1
##         ybar0
## [1,] 104.8307
## [2,] 103.7027
## [3,] 103.9726
## [4,] 103.6066
## [5,] 104.8607
## [6,] 105.7734
## [7,] 104.4688
##
## attr(,"class")
## [1] "mcmc.list"
```

Your `head(rsam)` should now be identical to mine.

6.4 Missing observations

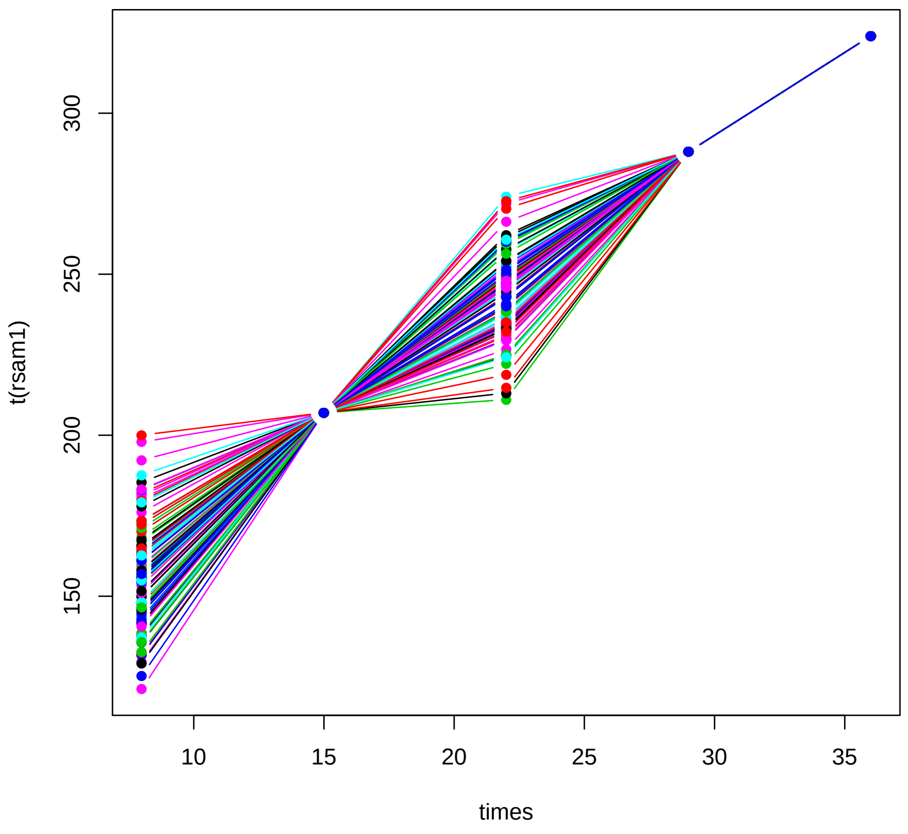
Missing observations are easy to handle. Suppose that we were missing the first and third observations on Rat 16. These just become `NA` in the `Y` matrix (we already did this in the convergence test). We can collect imputed values from the Markov chain.

I'll do this one with the new model, with a common β for all rats.

```
#### pretend that some obervations are missing

Y <- raw
Y[16, c(1, 3)] <- NA # Rat 16 missing a couple of values
mydata$Y <- Y

myrats0 <- jags.model("Rats0.bug", data = mydata, quiet = TRUE)
update(myrats0, n.iter = 3000)
rsam <- coda.samples(myrats0, "Y[16, 1:5]", n.iter = 1000)
head(rsam)

## [[1]]
## Markov Chain Monte Carlo (MCMC) output:
## Start = 3001
## End = 3007
## Thinning interval = 1
##       Y[16,1] Y[16,2]  Y[16,3] Y[16,4] Y[16,5]
## [1,] 128.7586     207 226.4811     288     324
## [2,] 153.3167     207 231.2442     288     324
## [3,] 156.2232     207 225.5468     288     324
## [4,] 173.7728     207 231.5943     288     324
## [5,] 170.1133     207 246.6527     288     324
## [6,] 154.0430     207 255.4892     288     324
## [7,] 160.9726     207 245.9760     288     324
##
## attr(,"class")
## [1] "mcmc.list"

# summary(rsam) # to get estimates for the missing values
```

Let's have a look at the imputed growth curves for this rat.

```
## plot a random sample of 100 of the growth curves

rsam1 <- sample.int(nrow(rsam[[1]]), 100)
rsam1 <- rsam[[1]][rsam1, ]
matplot(times, t(rsam1), type = "b", pch = 16, lty = 1)
```



There is too much variation in the imputed values for my liking.

7 Afterword

A final word on the Rats model. I have used the standard model, in which the coefficients α and β_i are conditionally Normal. But if this were my inference, I would use Gamma distributions for the β_i, imposing the constraint that the rats gain weight over time; to allow, in the prior PDF, that the rats might shrink in their first five weeks seems bizarre. It would be better still to use a functional form for weight as a function of time which imposed the obvious constraint that weight can never be negative, or zero.

References

Brooks, S. and Gelman, A. (1998). General methods for monitoring convergence of iterative simulations. *Journal of Computational and Graphical Statistics*, 7(4):434–455.