CHAPTER

5

Running WinBUGS from R via R2WinBUGS

OUTLINE

5.1 INTRODUCTION

In Chapter 4, it has become clear that as soon as you want to use WinBUGS more frequently, pointing and clicking is a pain and it is much more convenient to run the analyses by using a programming language. In program R, we can use the add-on package R2WinBUGS (Sturtz et al., 2005) to communicate with WinBUGS, so WinBUGS analyses can be run directly from R (also see package BRugs, not featured here, in connection with OpenBugs). The ingredients of an entire analysis are specified in R and sent to WinBUGS, then WinBUGS runs the desired number of updates, and the results, essentially just a long stream of random draws from the posterior distribution for each monitored parameter, along with an index showing which value of that stream belongs to which parameter, are sent back to R for convenient summary and further analysis. This is how we will be using WinBUGS from now on (so I will assume that you have R installed on your computer).

Any WinBUGS run produces a text file called "`codaX.txt`" for each chain X requested along with another text file called "`codaIndex.txt`."

DOI: 10.1016/B978-0-12-378605-0.00005-3

When requesting three chains, we get `coda1.txt`, `coda2.txt`, `coda3.txt`, and the index file. They contain all sampled values of the Markov chains (MCs) and can be read into R using facilities provided by the R packages coda or boa and then analyzed further. Indeed, sometimes, for big models, the R object produced by R2WinBUGS may be too big for one's computer to swallow, although it may still be possible to read in the coda files. Doing this and using coda or boa for analyzing, the Markov chain Monte Carlo (MCMC) output may then be one's only way of obtaining inferences.

As an example for how to use WinBUGS in combination with R through R2WinBUGS, we rerun the model of the mean, this time for a custom-drawn sample of male peregrines. This is the first time where we use R code to simulate our data set and then analyze it in WinBUGS as well as by using the conventional facilities in R.

First, we assemble our data set, i.e., simulate a sample of male peregrine body mass measurements. Second, we use R to analyze the model of the mean for this sample in the frequentist mode of inference. Third, we use WinBUGS called from R to conduct the same analysis in a Bayesian mode of inference. Remember, analyzing data is like repairing motorcycles. Thus, analyzing the data set means breaking it apart into those pieces that we had used earlier to build it up. In real life, of course, nature assembles the data sets for us and we have to infer truth (i.e., nature's assembly rules).

5.2 DATA GENERATION

First, we create our data:

```
# Generate two samples of body mass measurements of male peregrines
y10 <- rnorm(n = 10, mean = 600, sd = 30)       # Sample of 10 birds
y1000 <- rnorm(n = 1000, mean = 600, sd = 30)   # Sample of 1000 birds

# Plot data
xlim = c(450, 750)
par(mfrow = c(2,1))
hist(y10, col = 'grey ', xlim = xlim, main = 'Body mass (g) of 10 male peregrines')
hist(y1000, col = 'grey', xlim = xlim, main = ' Body mass (g) of
1000 male peregrines')
```

Here, and indeed in all further examples, it is extremely instructive to execute the previous set of statements repeatedly to experience sampling error—the variation in one's data that stems from the fact that only part but not the whole of a variable population has been measured. Sampling error, or sampling variance, is something absolutely central to statistics, yet it is among the most difficult concepts for ecologists to grasp,

especially, since in practice we only ever observe a single sample from the distribution that characterizes that variation! It is astonishing to observe how different repeated realizations from the exact same random process can be, here, the sampling of 10 or 1000 male peregrines from an assumed infinite population of male peregrines. Also surprising is, how far from normal the distribution in the smaller sample may look.

After playing around a while, we keep one pair of samples and go on. Note that unless you set a so-called seed for the random-number generator in R (for details, type `?set.seed`), you will have a different sample from me (i.e., this book) and therefore also get slightly different output in the ensuing analyses (although you can download the exact data sets analyzed for the book from the Web site).

5.3 ANALYSIS USING R

We can conduct a quick classic analysis of this model using the linear regression facilities in R on the larger sample.

```
summary(lm(y1000 ~ 1))
> summary(lm(y1000 ~ 1))

[ ...]

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  599.664      1.008   594.6   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 31.89 on 999 degrees of freedom
```

We recognize well the estimates of the population mean (599.664) and population standard deviation (SD), which is called the residual standard error (31.89). Now let's do the same analysis in WinBUGS.

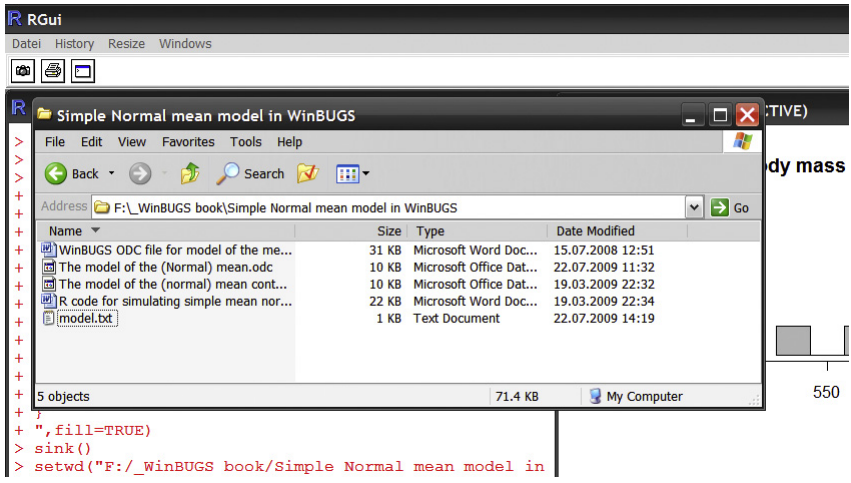
5.4 ANALYSIS USING WinBUGS

Remember that you always need to load the R2WinBUGS package first, although we won't show this again from now on. Also, we need to set the R working directory.

```
library(R2WinBUGS)        # Load the R2WinBUGS library
setwd("F:/_WinBUGS book/Simple Normal mean model in WinBUGS") # wd

# Save BUGS description of the model to working directory
sink("model.txt")
```

```
cat("
model {

# Priors
 population.mean ~ dunif(0,5000)          # Normal parameterized by precision
 precision <- 1 / population.variance     # Precision = 1/variance
 population.variance <- population.sd * population.sd
 population.sd ~ dunif(0,100)

# Likelihood
 for(i in 1:nobs){
    mass[i] ~ dnorm(population.mean, precision)
 }
}
",fill=TRUE)
sink()
```

This last bit of code writes into the R working directory a text file named "`model.txt`" containing the WinBUGS description of the model. We can see this when we look at the Windows Explorer after execution of this set of statements.



Next, we need to package the data that WinBUGS uses in the analysis. We do this by creating a bundle that contains both the data themselves and a count of the number of data points.

```
# Package all the stuff to be handed over to WinBUGS
# Bundle data
win.data <- list(mass = y1000, nobs = length(y1000))
```

Then, we define a function that creates random starting values, the inits function. We could also explicitly supply these initial values for each Markov chain requested, as we did in the previous chapter. However, this is less flexible, since we would need to explicitly specify one set of inits for each chain. If we wanted five chains instead of three, we would have to add two sets of inits. In contrast, a function will simply be executed two more times.

```
# Function to generate starting values
inits <- function()
  list(population.mean = rnorm(1,600), population.sd = runif(1,1,30))
```

As a reminder, if you are unsure what an R function such as `rnorm()` or `runif()` means, just type `?rnorm` or `?runif` in the R console. We also have to tell WinBUGS for which parameters it should save the posterior draws. Let's say we want to keep the variance also.

```
# Parameters to be monitored (= to estimate)
params <- c("population.mean", "population.sd", "population.variance")
```

Then, the MCMC settings need to be selected.

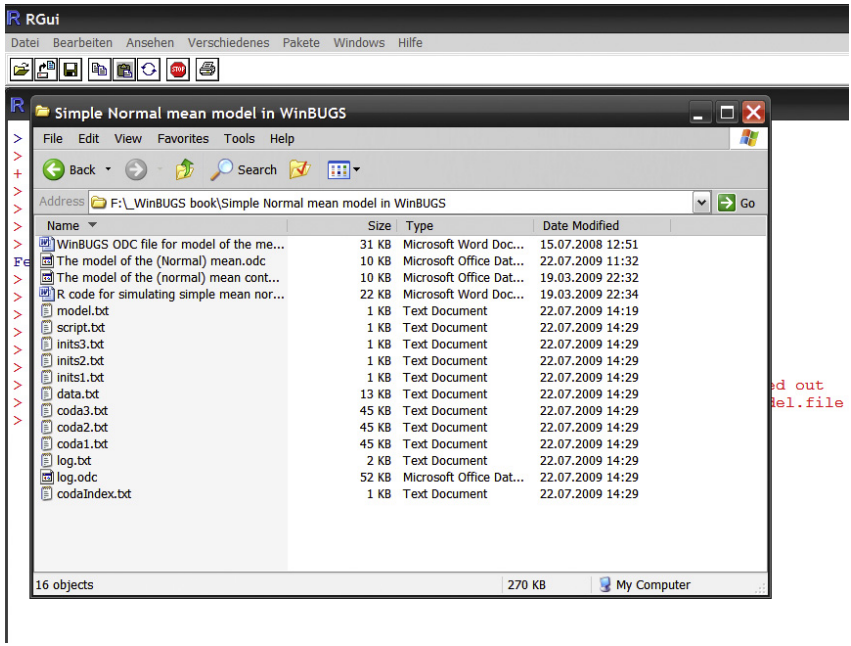
```
# MCMC settings
nc <- 3              # Number of chains
ni <- 1000           # Number of draws from posterior (for each chain)
nb <- 1              # Number of draws to discard as burn-in
nt <- 1              # Thinning rate
```

Finally, the function `bugs()` is called to perform the analysis in WinBUGS and put its results into an R object called `out`:

```
# Start Gibbs sampler: Run model in WinBUGS and save results in object called out
out <- bugs(data = win.data, inits = inits, parameters.to.save = params, model.file
= "model.txt", n.thin = nt, n.chains = nc, n.burnin = nb, n.iter = ni, debug =
TRUE, DIC = TRUE, working.directory = getwd())
```

During execution of the program in WinBUGS, the R window is frozen. Once the requested number of draws from the posterior has been produced, WinBUGS presents a graphical (the "history") and numerical summary of those parameters for which monitoring (i.e., estimation) was requested. On exiting WinBUGS, we have the results in various formats in our working directory (e.g., `coda1.txt`, `coda2.txt`, `coda3.txt`, `log.odc`, and `log.txt`) as well as in the R workspace a new object named `out`, a summary of which can be obtained by just typing its name (i.e., `out`). Note that setting `debug = FALSE` would cause WinBUGS to exit automatically after completion of the requested number of draws. This is important for instance when running repeated simulations. Otherwise, keeping WinBUGS open after the required

number of draws have been taken from the posterior (i.e., setting `debug = TRUE`) allows you to use WinBUGS for additional analyses, for instance, to make plots.



Now look at the R workspace and note the new object called *out*:

```
ls()
> ls()
 [1] "inits"     "nb"     "nc"    "ni"     "nt"   "out"   "params"
 [8] "win.data"  "xlim"   "y10"   "y1000"
```

We look at a summary of the Bayesian analysis:

```
out       # Produces a summary of the object
> out
Inference for Bugs model at "model.txt", fit using WinBUGS,
  3 chains, each with 1000 iterations (first 1 discarded)
  n.sims = 2997 iterations saved
                      mean    sd   2.5%    25%    50%    75%  97.5%  Rhat  n.eff
population.mean      599.7   1.0  597.7  598.9  599.6  600.4  601.7   1.0   3000
population.sd         32.0   1.8   30.6   31.4   31.9   32.4   33.4   1.1   3000
population.variance 1027.0 173.7  934.6  988.7 1019.0 1050.0 1116.0   1.1   3000
deviance            9765.1  33.4 9762.0 9762.0 9763.0 9764.0 9769.0   1.1   1500
```

```
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 = Dbar-Dhat)
pD = 3.5 and DIC = 9768.6
DIC is an estimate of expected predictive error (lower deviance is better).
```

Actually, object *out* contains a lot more information, which we can see by listing all the objects contained within *out* by typing `names(out)`.

```
> names(out)
 [1] "n.chains"      "n.iter"      "n.burnin"    "n.thin"
 [5] "n.keep"        "n.sims"      "sims.array" "sims.list"
 [9] "sims.matrix"   "summary"     "mean"        "sd"
[13] "median"        "root.short"  "long.short" "dimension.short"
[17] "indexes.short" "last.values" "isDIC"       "DICbyR"
[21] "pD"            "DIC"         "model.file" "program"
```

You can look into any of these objects by typing their name, which will usually print their full content, or by applying some summarizing function such as `names()` or `str()` on them. (Again, when not sure what an R function does, just type `?names` or `?str` into your R console.) You can also look deeper; note some degree of redundancy among `sims.array`, `sims.list`, and `sims.matrix`:

```
str(out)
> str(out)
List of 24
 $ n.chains      :  num 3
 $ n.iter        :  num 1000
 $ n.burnin      :  num 1
 $ n.thin        :  num 1
 $ n.keep        :  num 999
 $ n.sims        :  num 2997
 $ sims.array    :  num [1:999, 1:3, 1:4] 600 601 600 599 599 ...
  ..- attr(*, "dimnames")=List of 3
  .. ..$ : NULL
  .. ..$ : NULL
  .. ..$ : chr [1:4] "population.mean" "population.sd" "population.variance"
"deviance"
 $ sims.list     :List of 4
  ..$ population.mean     : num [1:2997] 600 600 601 601 600 ...
  ..$ population.sd       : num [1:2997] 31.8 32.8 31.6 32 31.8 ...
  ..$ population.variance: num [1:2997] 1013 1076 1000 1027 1011 ...
  ..$ deviance            : num [1:2997] 9762 9763 9764 9763 9762 ...
 $ sims.matrix   :  num [1:2997, 1:4] 600 600 601 601 600 ...
```

```
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : chr [1:4] "population.mean" "population.sd" "population.variance"
"deviance"
 $ summary       :  num [1:4, 1:9] 599.66 32 1027.02 9765.13 1.04 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:4] "population.mean" "population.sd" "population.variance"
"deviance"
  .. ..$ : chr [1:9] "mean" "sd" "2.5%" "25%" ...
 $ mean          :List of 4
  ..$ population.mean    : num 600
  ..$ population.sd      : num 32
  ..$ population.variance: num 1027
  ..$ deviance           : num 9765
 $ sd            :List of 4
  ..$ population.mean    : num 1.04
  ..$ population.sd      : num 1.78
  ..$ population.variance: num 174
  ..$ deviance           : num 33.4
 $ median        :List of 4
  ..$ population.mean    : num 600
  ..$ population.sd      : num 31.9
  ..$ population.variance: num 1019
  ..$ deviance           : num 9763
[ ... ]
```

Object *out* contains all the information contained in the coda files that WinBUGS produced plus some processed items such as summaries like mean values of all monitored parameters, the Brooks–Gelman–Rubin (BGR) convergence diagnostic called `Rhat`, and an effective sample size that corrects for the degree of autocorrelation within the chains (more autocorrelation means smaller effective sample size). We can now apply standard R commands to get what we want from this raw output of our Bayesian analysis of the model of the mean.

For a quick check whether any of the parameters has a BGR diagnostic greater than 1.1 (i.e., has Markov chains that have not converged), you can type this:

```
hist(out$summary[,8])        # Rhat values in the eighth column of the summary
which(out$summary[,8] > 1.1) # None in this case
```

For trace plots for the entire chains, do

```
par(mfrow = c(3,1))
matplot(out$sims.array[1:999,1:3,1], type = "l")
matplot(out$sims.array[,,2] , type = "l")
matplot(out$sims.array[,,3] , type = "l")
```

… or just for the start of the chains to see how rapidly they converge …

```
par(mfrow = c(3,1))
matplot(out$sims.array[1:20,1:3,1], type = "l")
matplot(out$sims.array[1:20,,2] , type = "l")
matplot(out$sims.array[1:20,,3] , type = "l")
```

We can also produce graphical summaries, e.g., histograms of the posterior distributions for each parameter:

```
par(mfrow = c(3,1))
hist(out$sims.list$population.mean, col = "grey")
hist(out$sims.list$population.sd, col = "blue")
hist(out$sims.list$population.variance, col = "green")
```

… or plot the (lack of) correlation between two parameters:

```
par(mfrow = c(1,1))
plot(out$sims.list$population.mean, out$sims.list$population.sd)
```

or

```
pairs(cbind(out$sims.list$population.mean, out$sims.list$population.sd,
out$sims.list$population.variance))
```

Numerical summaries of the posterior distribution can also be obtained, with the standard deviation requested separately:

```
summary(out$sims.list$population.mean)
summary(out$sims.list$population.sd)
sd(out$sims.list$population.mean)
sd(out$sims.list$population.sd)
```

Now compare this again with the classical analysis using maximum likelihood and the R function `lm()`. The results are almost indistinguishable.

```
summary(lm(y1000 ~ 1))
```

To summarize, after using R2WinBUGS, the R workspace contains all the results from your Bayesian analysis (the coda files contain them as well but in a less accessible format). If you want to keep these results, you must save the R workspace or save them in an external file.

5.5 SUMMARY

We have repeated the analysis of the model of the mean from the previous chapter by calling WinBUGS from within program R using the R2WinBUGS interface package. This is virtually always the most efficient mode of running analyses in WinBUGS and is the way in which we will use WinBUGS in the remainder of this book. We have also seen that a

Bayesian analysis with vague priors yields almost identical estimates as a classical analysis, something that we will see many more times.

EXERCISES

1. *Informative priors*: The analysis just presented uses vague priors, i.e., the effect of the prior on the posterior distribution is minimal. As we have just seen, Bayesian analyses with such priors typically yield estimates very similar numerically to those obtained using maximum likelihood.

   To see how the posterior distribution is influenced by both prior and likelihood, assume that we knew that the average body mass of male peregrines in populations like the one we sampled lay between 500 and 590 g. We can then formally incorporate this information into the analysis by means of the prior distribution. Hint: Change some code bits as follows, and let WinBUGS generate initial values automatically:

   ```
   # Priors
   population.mean ~ dunif(500, 590) # Mean mass must lie between 500 and 590
...
   ```

   Rerun the analysis (again, with the prior for the `population.sd` slightly changed) and see how the parameter estimates and their uncertainty change under this prior. Is this a bad thing? You may want to experiment with other limits for the uniform prior on the sd, for instance, set it at (0,10). Run the model and inspect the posterior distributions.
2. Run the model for the small and large data set and compare posterior means and SDs. Explain the differences.
3. Run the analysis for the larger data set for a long time, and from time to time, draw a density plot (in WinBUGS) or inspect the stats, for instance, after 10, 100, 1000, and 10 000 iterations: see how increasing length of Markov chains leads to improved estimates. Look at the MC error (in the WinBUGS output, hence you have to set `default = TRUE`) and at the values of the BGR statistic (=Rhat). Explain what you see.
4. Compare MC error and posterior SD and distinguish these things conceptually. See how they change in Markov chains of different lengths (as in Exercise 3).
5. *Derived quantities*: Estimate the coefficient of variation (CV = SD/mean) of body mass in that peregrine population. Report a point estimate (with SE) of that CV and also draw a picture of its posterior distribution.
6. *Swiss hare data*: Fit the model of the mean to *mean.density* and report the mean, the SE of the mean, and a 95% credible interval for mean hare density.