CHAPTER

5 FITTING MODELS USING THE BAYESIAN MODELING SOFTWARE BUGS AND JAGS

CHAPTER OUTLINE

5.1 INTRODUCTION

In this chapter, we introduce BUGS software (WinBUGS, OpenBUGS, JAGS) by fitting some very basic models that typically serve as building blocks for more complex hierarchical models (HMs). We briefly outline the salient features of the BUGS language, point you to the use of WinBUGS and OpenBUGS as standalone software, but then run WinBUGS and JAGS exclusively from R to fit a series of basic linear and generalized linear models (GLMs). These include two examples with random effects, one with a Poisson data distribution (Poisson GLMM), and another with a binomial data distribution (binomial GLMM). This emphasizes the conceptual clarity enforced by the BUGS model

Applied Hierarchical Modeling in Ecology. http://dx.doi.org/10.1016/B978-0-12-801378-6.00005-9

definition language about what GLMs and random effects are. The two main groups of topics covered in this chapter are:

- *Bayesian MCMC engines and BUGS software*: WinBUGS, OpenBUGS and JAGS, and Bayesian MCMC-based inference; model specification, posterior inference, predictions, goodness-of-fit (GoF), and missing values (NAs), all shown through R; the great numerical resemblance between Bayesian estimates and maximum likelihood estimates (MLEs) when vague priors are used in a Bayesian analysis.
- *Applied statistics*: Linear models, GLMs, and traditional random-effects, or mixed, models (i.e., the same topics as in Chapter 3, but now illustrated with BUGS) and the things you can do with them—e.g., estimation, testing, predictions, and fit assessment using parametric bootstrap and posterior predictive distributions.

Clearly, the aim of this chapter is *not* to provide a detailed and comprehensive introduction to BUGS for GLMs and traditional mixed models: there are entire books dedicated to those topics (e.g., Gelman and Hill, 2007; McCarthy, 2007; Ntzoufras, 2009; Kéry, 2010). Even less do we strive for an overview of the vast field of linear models and GLMs, about which hundreds of books must have been written. Rather, we want to give a hands-on introduction to the use of BUGS software to fit those types of statistical models that are probably used most widely by ecologists. We also want to illustrate in practice the key topics dealt with in Chapters 2 and 3, such as how parameter estimation, testing, prediction, and model criticism are accomplished in the most widely used Bayesian software BUGS. We do expect you to have at least some applied knowledge of linear models, GLMs, and random effects, as we covered them in Chapter 3.

5.2 INTRODUCTION TO BUGS SOFTWARE: WinBUGS, OpenBUGS, AND JAGS

WinBUGS (Lunn et al., 2000, 2009, 2013), OpenBUGS (Thomas et al., 2006; Lunn et al., 2009, 2013), JAGS (Plummer, 2003) and NIMBLE (NIMBLE Development Team, 2015; de Valpine et al., in review) are generic Bayesian modeling software packages that allow you to describe almost any arbitrarily complex statistical model using the BUGS model definition language and to fit the model using MCMC techniques. The BUGS language is an ingeniously simple language, very similar to R, that lets you specify remarkably complex statistical models in a very concise and easy-to-understand way. Arguably, BUGS is currently the only software that enables ecologists without a formal training in statistics and computation to fit a very large range of fully custom statistical models with confidence. (Unfortunately, the powerful new STAN software (www.mc-stan.org; also see Gelman et al. (2014) and Korner-Nievergelt et al. (2015)) does *not* allow discrete random effects, such as we always have when modeling distribution and abundance using site-occupancy and *N*-mixture models. You can fit such models in STAN, but only when you explicitly specify the integrated likelihood. Arguably, doing this yourself is beyond the level of statistical understanding of most BUGS users.). Note that when we write "BUGS," we typically mean WinBUGS or OpenBUGS, but sometimes we mean JAGS as well and you may even imagine that we include NIMBLE, because this new software also uses a dialect of the BUGS language. We hope that the meaning of "BUGS" will become clear from the context. Functionality is very similar among the three BUGS sisters, with one major exception: WinBUGS and OpenBUGS have a module called `geoBUGS` for fitting spatial models to deal with spatial or temporal autocorrelation; see Chapters 21 and 22.

BUGS and JAGS software is freely available and does four things for you:

1. Lets you describe almost any kind of statistical model in the simple and powerful BUGS **language**
2. Translates your BUGS language description of a statistical model into an **MCMC algorithm**
3. **Runs the algorithm** for as long as you wish (or have time to wait), and thereby accumulates samples of the desired joint posterior distribution for all unknown quantities in your model
4. Allows some processing of results, such as graphical or tabular **posterior summaries** and convergence assessment (not for JAGS)

WinBUGS and OpenBUGS exist as Windows standalone applications, and OpenBUGS is also available for Unix/Linux and Mac operating systems. They are available through each program's website (WinBUGS: www.mrc-bsu.cam.ac.uk/software/bugs/; OpenBUGS: www.openbugs.net/w/FrontPage). They come with a comprehensive hypertext manual and several volumes of worked and richly commented example analyses. Both manuals and example volumes are extremely rich in information, but it is fair to say that this wealth of information is not easy to navigate, most of all, because there is no index available. Development of WinBUGS ceased about 10 years ago and the developmental branch of the BUGS project has moved over to OpenBUGS. However, at the time of writing, OpenBUGS does not seem to have evolved very much beyond the capabilities of the original WinBUGS, in terms of the capabilities of the programming language or the speed of the MCMC algorithms. JAGS is developed by Martyn Plummer at the International Agency for Research on Cancer in Lyon, France. It does not have a Windows user interface, and its manual and other relevant things can be downloaded from sourceforge.net/projects/mcmc-jags/files/. NIMBLE is another project using the BUGS language see r-nimble.org/). For all three, there is a tremendous amount of documentation freely available on the web (try googling "BUGS and X," where X is your favorite model); also, there is a user group with a discussion list.

In this book, we use WinBUGS and JAGS, but note (and show in one example) that OpenBUGS can be run from R in exactly the same way as WinBUGS. We do not show how to run BUGS as a standalone program, because most people use it from R and because use of the standalone is explained in many books, including those by Woodworth (2004), McCarthy (2007), Ntzoufras (2009), Kéry (2010), and Woodward (2011). To run BUGS from R, you may use a number of packages including `R2WinBUGS` or `R2OpenBUGS` (Sturtz et al., 2005), `BRugs` (Thomas et al., 2006), `R2jags` (Su and Yajima, 2014), `rjags` (Plummer, 2015), `dclone` (Solymos, 2010), `jagsUI` (Kellner, 2015), or `runjags` (Denwood, 2015). We use `R2WinBUGS` for WinBUGS and `jagsUI` for JAGS, though we illustrate the use of `rjags` in Section 5.4. The new package `jagsUI` has similar capabilities as `R2jags`, its output is very similar to that of `R2WinBUGS`, and it avoids some problems in `R2jags` (at least as of mid-2014, the latter did not adequately separate the MCMC adaptation and burn-in phases). Package `runjags` has extensive facilities for parallel processing with JAGS, something that can be attractive given the sometimes extremely long run times of MCMC analyses. The packages `R2jags`, `jagsUI`, and `dclone` also have some facilities for running JAGS and/or WinBUGS in parallel.

Even though the three BUGS engines and also NIMBLE are totally separate programs and JAGS and NIMBLE are two entirely different enterprises, they use mere dialects of the BUGS language. So, what is the BUGS language? In a nutshell:

- The BUGS language serves to specify statistical models.
- It very much resembles R, but is not identical.
- There are "nodes" in statistical models; broadly, these are model quantities such as parameters, latent variables, predictions, and missing or observed data.

- Exactly two kinds of relations exist among nodes in a model: deterministic (represented by the arrow or assignment operator "<–"—*not* the equal sign) or stochastic (represented by the tilde "~"). All models that can be fitted in BUGS may be represented mathematically as so-called directed acyclic graphs, or DAGs (Gilks et al., 1994; Lunn et al., 2000, 2013).
- There is a moderate number of mathematical and statistical functions (see the manuals).
- There is a range of inbuilt statistical distributions (see the manuals); new distributions may be defined using the "zeros trick" or the "ones trick," provided you know how to write the likelihood explicitly (Lunn et al., 2013); see Section 5.8 below.
- The BUGS language is not really vectorized—we need loops to define the model for every element in vectors or multidimensional arrays. Indexing then becomes important—we use one or multiple indices to address elements of vectors or multidimensional arrays. (Note, though, that some vector or matrix operations are available, such as `inprod`).
- BUGS is a declarative language, and hence the order of statements does not matter, with the one main exception being what you put inside or outside of a loop.

This concludes our very brief first introduction to BUGS software and the BUGS language. We hope it suffices as a simple prelude to seeing BUGS "in action" for a series of basic and important statistical models in the remainder of this chapter (and in the rest of the book). We believe that learning by doing (or by watching others do), is the most powerful and quick way of learning a statistical programming language. Thus, in the remainder of this chapter we will fit a range of linear models, GLMs and random-effects (= hierarchical) models that are typical in the applied work of ecologists and indeed most empirical scientists. We illustrate, using BUGS, each type of model described in Chapter 3 using a data set of simulated counts of great tits (from Chapter 4), those being:

1. Multiple linear regression with normal errors (normal GLM): Sections 5.3–5.5, 5.8
2. Analysis of covariance (ANCOVA) model with normal errors (normal GLM): Sections 5.6 and 5.7
3. ANCOVA model with Poisson errors (Poisson GLM): Sections 5.9–5.10
4. ANCOVA model with binomial errors (logistic regression or binomial GLM): Sections 5.11–5.12
5. Poisson mixed model or Poisson generalized linear mixed model (Poisson GLMM): Section 5.13
6. Binomial mixed model or binomial generalized linear mixed model (binomial GLMM): Section 5.14

In this book, the normal distribution takes on a much less prominent role than in many statistics books, because we deal with models for counts. Counts are discrete and positive valued, and hence the normal is arguably an inadequate choice for describing such data, unlike the case for more conventional ecological data such as measurements on a continuous scale. Nevertheless, we will start by briefly illustrating the fitting of a model with normal response for the sake of illustration, and because ecologists are most familiar with a normal linear model. We do not necessarily endorse the application of every model in this chapter to the particular type of simulated data. However, we liked the idea of using a single data set to illustrate all models. Moreover, one can frequently see in the ecological literature all the kinds of analyses shown in this chapter—e.g., for better or worse the normal distribution is still quite often adopted for count data (see, e.g., state-space models for time series of population counts, Buckland et al., 2004b, 2007, and Chapter 5 in Kéry and Schaub, 2012).

The building blocks of much of what we do in hierarchical modeling of abundance, occurrence, and species richness are Poisson and binomial GLMs. We describe in the BUGS language some prototypical components of HMs; normal, Poisson, and binomial GLMs; and the specification of random effects. Almost all HMs in the remainder of this book are composed of these building blocks. We also illustrate the fitting of the same models using maximum likelihood (ML) to show

the numerical similarities of Bayesian estimates and MLEs for reasonable sample sizes and vague priors adopted in Bayesian analyses. (Technically, we use least squares or iteratively reweighted least squares for linear models and GLMs, but the resulting estimates are equivalent to MLEs.)

5.3 LINEAR MODEL WITH NORMAL RESPONSE (NORMAL GLM): MULTIPLE LINEAR REGRESSION

The normal distribution does not take on a prominent role in many analyses of counts, since counts are neither continuous nor can they be negative (though the normal becomes a better approximation when the counts become large). With a data set created using the function in Chapter 4, we model the mean count of great tits assuming that they come from a normal distribution. This model is a special case of a GLM with a normal distribution and identity link, where we directly model the mean tit count as a linear function of elevation and forest cover and there is no link transformation of the expected response.

$$Cmean_i \sim Normal(\mu_i, \sigma^2)$$

$$\mu_i = \alpha_0 + \alpha_1 * elev_i + \alpha_2 * forest_i + \alpha_3 * elev_i * forest_i$$

```
# Generate data with data.fn from chapter 4
set.seed(24)
data <- data.fn()
str(data)
attach(data)

# Summarize data by taking mean at each site and plot
Cmean <- apply(C, 1, mean)
par(mfrow = c(1,3))
hist(Cmean, 50)                # Very skewed
plot(elev, Cmean)
plot(forest, Cmean)
```

To fit a model in WinBUGS or JAGS run from R and using packages `R2WinBUGS` (Sturtz et al., 2005) and `jagsUI` (Kellner, 2015), we must first prepare all the "ingredients" of the analysis. Then, we use the function `bugs` or `jags` to ship them over to BUGS and thereby instruct the BUGS engines on how they should run the analysis. After the desired number of MCMC draws has been produced by BUGS, a long list of samples from the joint posterior distribution is imported back into R and summarized in various convenient ways. This list of MCMC samples is also contained in the `coda.txt` files produced by WinBUGS. We strive for a consistent layout of all of these steps that lead to an analysis in BUGS. We do this here for the first time and hence give much more detail than in later examples.

We need to prepare the following objects: data set, model, a function for initial values for the Markov chains, a list of parameters that we want to estimate, and the MCMC settings (how many chains, for how long, etc.). The first thing we prepare is a list containing the data to be analyzed. Importantly, for WinBUGS we must include only data that are used in the model definition, while for JAGS the data package may include data not used in the model.

```
# Package the data needed in a bundle
win.data <- list(Cmean = Cmean, M = length(Cmean), elev = elev, forest = forest)
str(win.data)                       # Check what's in win.data
```

Next, we use the `cat` function to write into the R working directory a named text file containing the model description in the BUGS language. The name of this text file is up to you (though ideally it should be shorter than the one we use here…).

```
# Write text file with model description in BUGS language
cat(file = "multiple_linear_regression_model.txt",
"    # --- Code in BUGS language starts with this quotation mark ---
model {

# Priors
alpha0 ~ dnorm(0, 1.0E-06)          # Prior for intercept
alpha1 ~ dnorm(0, 1.0E-06)          # Prior for slope of elev
alpha2 ~ dnorm(0, 1.0E-06)          # Prior for slope of forest
alpha3 ~ dnorm(0, 1.0E-06)          # Prior for slope of interaction
tau <- pow(sd, -2)                  # Precision tau = 1/(sd^2)
sd ~ dunif(0, 1000)                 # Prior for dispersion on sd scale

# Likelihood
for (i in 1:M){
  Cmean[i] ~ dnorm(mu[i], tau)      # dispersion tau is precision (1/variance)
  mu[i] <- alpha0 + alpha1*elev[i] + alpha2*forest[i] + alpha3*elev[i]*forest[i]
}

# Derived quantities
for (i in 1:M){
  resi[i] <- Cmean[i] - mu[i]
}
}"# --- Code in BUGS language ends on this line ---
)
```

This is the first place we meet a full model written in the BUGS language; hence, a couple comments are in order. First, you must be absolutely clear about which code is R and which is BUGS. The way we write our code, everything between the quotes as indicated by the comments is in the BUGS language, and everything outside is R. The purpose of the R code is simply to write the BUGS model code to a text file. As far as R is concerned, the BUGS model file is simply a large character vector. Shortly, we will ask WinBUGS or JAGS to try to understand and do something useful with the character vector that is output by R.

In the BUGS model, you see the following quantities (nodes): `alpha0`, `alpha1`, `alpha2`, `alpha3`, `tau`, `sd`, and `mu`, as well as `Cmean`, `elev`, and `forest`. The relationship between the response `Cmean` (the observed mean count of great tits), and the mean and dispersion of the normal random variable, `mu` and `tau`, is a stochastic one; therefore, they are connected with a tilde (~)—that is, `Cmean` is drawn from a normal distribution with mean `mu` and dispersion `tau`. In contrast, the expected mean count (`mu`) is connected with covariates `elev` and `forest` in a deterministic manner, and therefore we use the arrow or assignment operator (`<-`), which is equivalent to an equal sign in the BUGS language.

We also see three functions: the power and the normal and the uniform distribution functions (a list of the functions available in BUGS and JAGS can be found in the manuals). Finally, we use a loop to describe the statistical model for the relationship between the covariates and the mean counts of great tits for each of the M sites in turn, with each data point indexed by `i`. We could have defined the priors after the loop over M (which is the likelihood), but not inside of that loop. Finally, in BUGS the normal distribution is defined in terms of the precision, which is the reciprocal of the variance. Hence, a prior with very small precision, such as one over one million, has a large standard deviation of 1000 and is pretty flat over a very large range of possible values of a parameter or where the likelihood has support (see also Section 5.5.2).

We try to keep the two main parts of the Bayesian analysis separate, namely the likelihood, which describes the relationship between the observed data and the unknown parameters, and the priors, which describe our knowledge about these parameters using a probability distribution. (Note that this distinction becomes blurred in HMs, as you will see later.) Priors must be specified for the primary unknown quantities in a Bayesian analysis, here, the four regression parameters, `alpha0` through `alpha3`, and the variance parameter, `tau` or `sd`. We typically use suitably wide uniform distributions or "flat" normal distributions with a small precision (corresponding to a large variance) to specify our wish to let only (or mainly) the data influence the parameter estimates. To be vague, a prior needs to be approximately flat only over the range of values where the likelihood has support—i.e., where the latter has values that are effectively nonzero. The question of whether a prior is vague for a given data set and model must be ascertained by trial and error for every new data set or model anew in a kind of informal sensitivity analysis—you make a guess about what is a vague prior, fit the model, and then make the priors more diffuse and refit the model, and if the posterior distributions do not change, then you have specified vague priors.

Next, we define a function that generates random initial values with some dispersion for at least some of the parameters. We do not need to give initial values for each parameter, since BUGS randomly initializes parameters for which no initial values are given based on the prior defined in the model. At the beginning, you may find the distinction between priors and initial values confusing; however, the two are fundamentally different. Priors are part of the model when you analyze it using Bayesian methods and they always affect the estimates (the posterior distributions), even if only very slightly. In contrast, initial values are *not* part of the model; they simply represent the starting points for the Markov chains. After chain convergence, initial values no longer have any effect on the properties of the posterior samples, and they, along with all chain values until convergence, are discarded as an MCMC "burn-in."

Even though initial values do not affect the posterior distribution, they can be of great practical importance especially for complex models. If we start the chains at too wildly improbable values for a parameter, the Markov chains may never find their way to the region in the parameter space where the likelihood does have some support—i.e., we may never get convergence. Or the MCMC algorithm may not start when unsuitable initial values are chosen. On the other hand, to gauge convergence we want the initial values of multiple chains to be "overdispersed"—i.e., to start at reasonably different places. If the chains move to the same region in the parameter space, we have much greater confidence in chain convergence. Therefore, we must strike a balance in the generation of initial values between not enough and too much "overdispersion."

Finally, initial values must not contradict the priors, the model, or the data. For instance, initializing a variance at a negative value would make BUGS crash. Similarly, in an occupancy model (see Chapter 10), if we initialize at zero (corresponding to an "absence") the latent presence/absence state z for a site

where a species was detected, JAGS will crash (though not WinBUGS). This is a contradiction between model and data in the traditional occupancy model, which does not allow false-positives. Picking appropriate initial values is easy for simple models, but may become increasingly harder for more complex models (sometimes very much so). This is particularly so for JAGS, which does not forgive *any* contradictions. For complex models, there thus may be considerable trial and error involved in picking initial values. Here is a function that will create our initial values:

```
# Initial values
inits <- function() list(alpha0 = rnorm(1,0,10), alpha1 = rnorm(1,0,10), alpha2 =
rnorm(1,0,10), alpha3 = rnorm(1,0,10))
```

The next object is a list with the names of the parameters we want to estimate. While all unknown quantities described in the model are internally updated (i.e., estimated), the MCMC samples are only saved in the output for those in this list:

```
# Parameters monitored (i.e., for which estimates are saved)
params <- c("alpha0", "alpha1", "alpha2", "alpha3", "sd", "resi")
```

The final things to decide are the number of chains (`nc`), their length or number of iterations (`ni`), the length of the initial burn-in period (`nb`), and whether we want to discard any intermediate values ("thin") to produce a smaller but more information-dense sample from the posterior. For instance, setting `nt = 10`, we keep only every 10th value, which may be useful for saving disk space in very parameter-rich models. (However, there is no need to thin, since we always toss out information; MacEachern and Berliner, 1994, Link and Eaton, 2012.) The MCMC settings are also chosen by trial and error: when developing an analysis, you will often start with extremely short chains to ensure that everything is programmed fine. Then, you may run the chains longer to assess the necessary burn-in length, and finally run a production run that includes the required burn-in length and a sufficiently large post-burn-in sample:

```
# MCMC settings
ni <- 6000  ;  nt <- 1  ;  nb <- 1000  ; nc <- 3
```

To observe convergence at the start of the chains and for testing the code, you could choose the following MCMC settings.

```
# ni <- 10  ;  nt <- 1  ;  nb <- 0  ; nc <- 8 # not run
```

Now we have created all the objects that we need in order to use the `bugs` or `jags` functions to run an analysis. After loading the `R2WinBUGS` package, we run the analysis in WinBUGS first (check `?bugs` if you need information about the last three function arguments in the function call below):

```
# Call WinBUGS from R (approximate run time (ART) <1 min)
library(R2WinBUGS)
bugs.dir <- "C:/WinBUGS14/"          # Place where your WinBUGS is installed
out1B <- bugs(win.data, inits, params, "multiple_linear_regression_model.txt",
n.chains = nc, n.thin = nt, n.iter = ni, n.burnin = nb, debug = TRUE,
bugs.directory = bugs.dir, working.directory = getwd())
```

The first thing you do should always be to visually inspect the trace plots to gauge convergence in WinBUGS (see Figure 5.1 for a sample). Then, we exit WinBUGS manually and look at the results

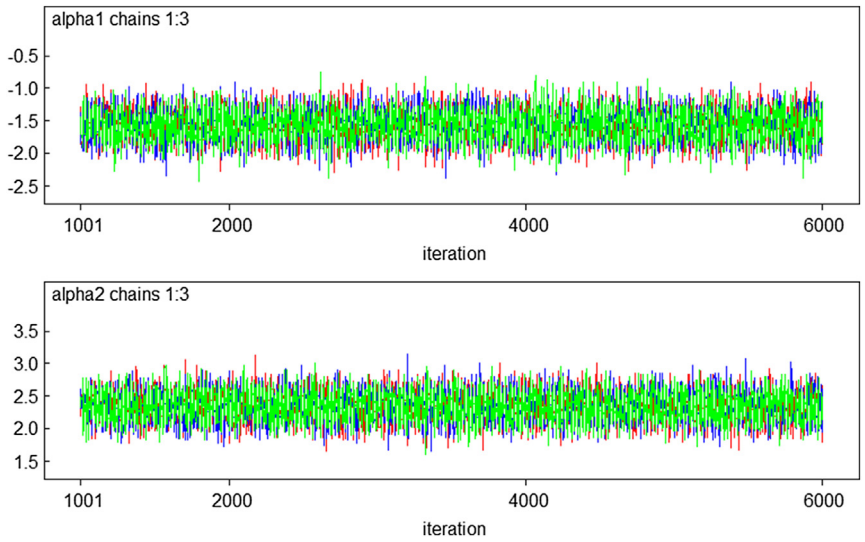

FIGURE 5.1

Trace plots produced by WinBUGS (when `debug = TRUE`) serve to visually check for convergence, here shown for two slope parameters in the multiple linear regression model. Oscillations around a horizontal level (or a "grassy" look) indicate convergence.

produced by `R2WinBUGS` in an overview. NOTE (this is important): with `debug = TRUE`, you have to manually exit WinBUGS before the BUGS results are imported into R. Before you do this, the R window is frozen.

If convergence has not been reached then we either repeat with longer chains and greater burn-in, or manually do an additional burn-in in R by tossing out additional draws at the start of the chains. You will only do the latter when a model takes very long to run; otherwise, it is much easier to simply rerun the model with changed MCMC settings.

The R object created is huge and contains many elements, of which the most frequently used are perhaps "sims.list," "summary," "mean," and "sd." Here are two overviews of the object.

```
# Overview of the object created by bugs
names(out1B)
 [1] "n.chains"          "n.iter"          "n.burnin"
 [4] "n.thin"            "n.keep"          "n.sims"
 [7] "sims.array"        "sims.list"       "sims.matrix"
[10] "summary"           "mean"            "sd"
[13] "median"            "root.short"      "long.short"
[16] "dimension.short"   "indexes.short"   "last.values"
[19] "isDIC"             "DICbyR"          "pD"
[22] "DIC"               "model.file"      "program"
```

```
str(out1B, 1)
List of 24
 $ n.chains        : num 3
 $ n.iter          : num 6000
 $ n.burnin        : num 1000
 $ n.thin          : num 1
 $ n.keep          : num 5000
 $ n.sims          : num 15000
 $ sims.array      : num [1:5000, 1:3, 1:273] 2.5 2.26 2.19 2.4 2.26 ...
  ..- attr(*, "dimnames")=List of 3
 $ sims.list       :List of 7
 $ sims.matrix     : num [1:15000, 1:273] 2.24 2.36 2.59 2.35 2.27 ...
  ..- attr(*, "dimnames")=List of 2
 $ summary         : num [1:273, 1:9] 2.36 -2.38 3.4 -1.98 2.11 ...
  ..- attr(*, "dimnames")=List of 2
 $ mean            :List of 7
 $ sd              :List of 7
 $ median          :List of 7
 $ root.short      : chr [1:7] "alpha0" "alpha1" "alpha2" "alpha3" ...
 $ long.short      :List of 7
 $ dimension.short: num [1:7] 0 0 0 0 0 1 0
 $ indexes.short   :List of 7
 $ last.values     :List of 3
 $ isDIC           : logi TRUE
 $ DICbyR          : logi FALSE
 $ pD              : num 4.98
 $ DIC             : num 1159
 $ model.file      : chr "multiple_linear_regression_model.txt"
 $ program         : chr "WinBUGS"
 - attr(*, "class")= chr "bugs"
```

Before we look any further into the results, we fit the same model in OpenBUGS using the `bugs` function in the `R2OpenBUGS` package, and finally in JAGS using the function `jags` in the package `jagsUI` (Kellner, 2015). For both, we can use exactly the same ingredients that we just prepared for the analysis in WinBUGS.

```
# Call OpenBUGS from R (ART <1 min)
library(R2OpenBUGS)
out1OB <- bugs(data=win.data, inits=inits, parameters.to.save = params, model.file =
"multiple_linear_regression_model.txt", n.chains = nc, n.thin = nt, n.iter = ni, n.burnin =
nb, debug = TRUE, working.directory = getwd())
detach("package:R2OpenBUGS", unload=T) # Otherwise R2WinBUGS is 'masked'
```

```
# Call JAGS from R (ART <1 min)
library(jagsUI)
?jags                  # Look at main function
out1J <- jags(win.data, inits, params, "multiple_linear_regression_model.txt",
n.chains = nc, n.thin = nt, n.iter = ni, n.burnin = nb)
```

We can easily run JAGS on multiple cores by setting the argument `parallel = TRUE`:

```
out1J <- jags(win.data, inits, params, parallel = TRUE,
"multiple_linear_regression_model.txt", n.chains = nc, n.thin = nt, n.iter = ni,
n.burnin = nb)
```

For models with long run times (several hours), this will speed up calculations by up to a factor of 3 when three cores are used (though usually the gain in speed is less). Shorter calculations may actually take longer, as the time to set up the parallel calculations outweighs the time saved by using multiple cores. The disadvantage of running JAGS in parallel is that the famous JAGS progress bar is no longer visible, thus diminishing our sense of accomplishment as we stare at our computer monitor.

For JAGS, we use the function `traceplot` to gauge convergence, either for a subset or for all parameters monitored.

```
par(mfrow = c(3,2))
traceplot(out1J, param = c('alpha1', 'alpha2', 'resi[c(1,3, 5:6)]')) # Subset
# traceplot(out1J)                  # All params
```

The object created by `jagsUI` is very similar to the one created when we run `R2WinBUGS`.

```
# Overview of object created by jags()
names(out1J)
 [1] "sims.list"  "means"         "sd"          "q2.5"        "q25"
 [6] "q50"        "q75"           "q97.5"       "overlap0"    "f"
[11] "Rhat"       "n.eff"         "pD"          "DIC"         "summary"
[16] "samples"    "modfile"       "model"       "parameters"  "mcmc.info"
[21] "run.date"   "random.seed"   "parallel"    "bugs.format"
```

Hence, using the R packages `R2WinBUGS`, `R2OpenBUGS`, and `jagsUI`, we can readily switch between WinBUGS, OpenBUGS, and JAGS for model fitting, and most of the time no change at all is required in the "ingredients" of the analysis. Next, we summarize the posterior distributions. Up to Monte Carlo (MC) error, estimates from BUGS and JAGS at convergence should be identical.

```
# Summarize posteriors from WinBUGS run
print(out1B, 2)
Inference for Bugs model at "multiple_linear_regression_model.txt", fit using WinBUGS,
  3 chains, each with 6000 iterations (first 1000 discarded)
  n.sims = 15000 iterations saved
             mean    sd    2.5%     25%     50%     75%   97.5% Rhat  n.eff
alpha0       1.66  0.12    1.44    1.58    1.66    1.74    1.89    1  15000
alpha1      -1.58  0.21   -1.98   -1.71   -1.58   -1.44   -1.17    1  15000
alpha2       2.35  0.20    1.95    2.21    2.35    2.48    2.73    1  15000
alpha3      -0.85  0.35   -1.55   -1.09   -0.86   -0.62   -0.15    1   5500
sd           1.86  0.08    1.71    1.80    1.86    1.91    2.03    1  15000
resi[1]     -0.48  0.17   -0.81   -0.59   -0.48   -0.37   -0.15    1  13000
resi[2]     -0.45  0.19   -0.82   -0.58   -0.45   -0.32   -0.08    1  11000
resi[3]      1.05  0.27    0.51    0.86    1.05    1.23    1.59    1  11000
```

```
[ output truncated ]
resi[265]    -0.61  0.13    -0.88    -0.70    -0.61     -0.52     -0.34     1  15000
resi[266]    -1.82  0.12    -2.06    -1.90    -1.82     -1.74     -1.58     1  15000
resi[267]     1.28  0.45     0.39     0.97     1.27      1.58      2.16     1  13000
deviance   1087.15  3.23  1083.00  1085.00  1086.00   1089.00   1095.00     1  15000

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 = 5.0 and DIC = 1092.2
DIC is an estimate of expected predictive error (lower deviance is better).

# Summarize posteriors from JAGS run
print(out1J, 2)
JAGS output for model 'multiple_linear_regression_model.txt', generated by jagsUI.
Estimates based on 3 chains of 6000 iterations,
burn-in = 1000 iterations and thin rate = 1,
yielding 15000 total samples from the joint posterior.
MCMC ran in parallel for 0.054 minutes at time 2014-11-11 14:58:27.

             mean    sd     2.5%      50%    97.5% overlap0    f  Rhat n.eff
alpha0       1.66 0.11     1.44     1.66     1.88     FALSE 1.00     1 15000
alpha1      -1.58 0.20    -1.97    -1.58    -1.18     FALSE 1.00     1 15000
alpha2       2.34 0.20     1.96     2.34     2.73     FALSE 1.00     1 15000
alpha3      -0.85 0.36    -1.55    -0.85    -0.14     FALSE 0.99     1 10597
sd           1.86 0.08     1.71     1.86     2.03     FALSE 1.00     1  9993
resi[1]     -0.48 0.17    -0.80    -0.48    -0.15     FALSE 1.00     1 15000
resi[2]     -0.45 0.19    -0.82    -0.44    -0.08     FALSE 0.99     1 13961
resi[3]      1.05 0.28     0.50     1.05     1.60     FALSE 1.00     1 15000
[ output truncated ]
resi[265]   -0.61 0.13    -0.87    -0.61    -0.35     FALSE 1.00     1 15000
resi[266]   -1.82 0.12    -2.06    -1.82    -1.58     FALSE 1.00     1 15000
resi[267]    1.28 0.45     0.40     1.27     2.15     FALSE 1.00     1 15000
deviance  1087.09 3.18  1082.89  1086.45  1094.82     FALSE 1.00     1 15000

Successful convergence based on Rhat values (all < 1.1).
Rhat is the potential scale reduction factor (at convergence, Rhat=1).
For each parameter, n.eff is a crude measure of effective sample size.

overlap0 indicates if 0 falls within the 95% credible interval for the parameter.
f represents the proportion of a parameter's posterior distribution with the same
sign as the mean; i.e., our confidence that the parameter is positive or negative.

DIC info: (pD = var(deviance)/2)
pD = 5.1 and DIC = 1092.15
DIC is an estimate of expected predictive error (lower is better).
```

The first thing to check in the work flow of our analysis is the penultimate column. It shows the value of the Gelman-Rubin convergence diagnostic `Rhat`, which is 1 at convergence (Gelman and Rubin, 1992; Brooks and Gelman, 1998); values below 1.1 or 1.2 are usually taken as indicating convergence, but not

necessarily for producing posterior summaries with sufficiently low Monte Carlo error (for which, say, 1.01 or 1.001 might be better). The chains for different parameters converge at different rates, and it is entirely possible to obtain an `Rhat` value that indicates convergence for one parameter, but not for another. Similarly, `n.eff` typically varies strongly between different parameters and often even between replicate runs of an analysis. It is the estimated size of an equivalent, independent MCMC sample that contains the same amount of information as does your dependent sample. Chains with stronger autocorrelation are associated with higher values of `Rhat` and lower values of `n.eff`. You can visually check the autocorrelation for the first five parameters and up to lag `n` by typing this (though in this simple model, there is almost no autocorrelation):

```
n <- 10                    # maximum lag
par(mfrow = c(2, 3), mar = c(5,4,2,2), cex.main = 1)
for(k in 1:5){
   matplot(0:n, autocorr.diag(as.mcmc(out1B$sims.array[,,k]), lags = 0:n),
      type = "l", lty = 1, xlab = "lag (n)", ylab = "autocorrelation",
      main = colnames(out1B$sims.matrix)[k], lwd = 2)
   abline(h = 0)
}
```

Note also that the different BUGS engines are liable to perform very differently in terms of the MCMC efficiency (as measured by `n.eff`, or more accurately, `n.eff` per unit time). Thus, you should expect that sometimes one of the BUGS engines might not mix well, or maybe not even work for a certain class of model. For this reason, we think it is always a good idea to try more than one BUGS engine on a model, at least in the exploratory stage of an analysis.

To continue our discussion of the model output; the first two columns in the posterior summary contain the posterior means and standard deviations, which can be used for a Bayesian point estimate and an analogue to the standard error of the estimate in a frequentist analysis. Columns 3–7 show the percentiles of the posterior samples, of which the 2.5% and 97.5% form a customary 95% Bayesian confidence interval (often called a credible interval and abbreviated CRI). At the bottom of the tables we obtain the effective number of parameters (pD) and the value of the deviance information criterion (DIC; Spiegelhalter et al., 2002), which for non-HMs can be used for model selection in the same way as the Akaike's information criterion (Burnham and Anderson, 2002). Unfortunately, for HMs with strongly nonnormal posterior distributions of some parameters, (which includes those with discrete random effects such as occupancy and *N*-mixture models), and hence for essentially all models in this book, the DIC is not appropriate for model selection and so you won't see us using it.

We can easily get various graphical overviews for both analyses (Figure 5.2).

```
plot(out1B)                # For WinBUGS analysis from R2WinBUGS
plot(out1J)                # For JAGS analysis from jagsUI

par(mfrow = c(1, 2), mar = c(5,4,2,2), cex.main = 1)
whiskerplot(out1J, param = c('alpha0', 'alpha1', 'alpha2', 'alpha3', 'sd',
'resi[c(1,3, 5:7)]'))    # For JAGS analysis from jagsUI
library(denstrip)          # Similar, but more beautiful, with package denstrip
plot(out1J$alpha0, xlim=c(-4, 4), ylim=c(1, 5), xlab="", ylab="", type="n", axes =
F, main = "Density strip plots")
axis(1)
```

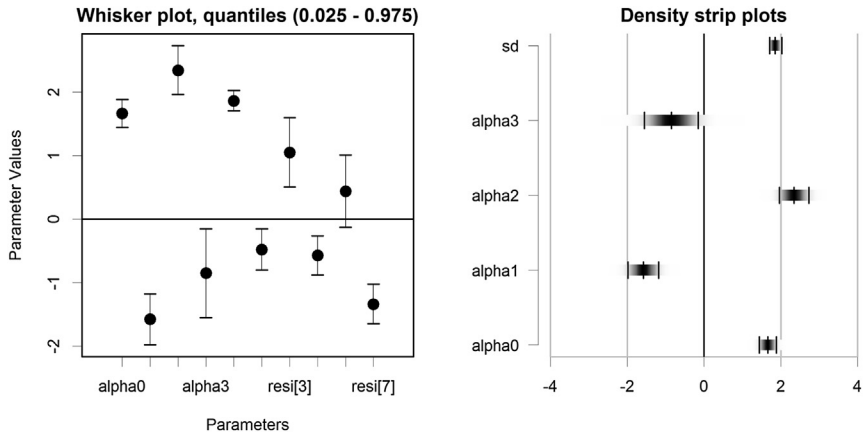

FIGURE 5.2

Two useful ways of graphically summarizing the posterior samples using the R functions `whiskerplot` (left) and `denstrip` (right).

```
axis(2, at = 1:5, labels = c('alpha0','alpha1','alpha2','alpha3','sd'), las = 1)
abline(v = c(-4,-2,2,4), col = "grey") ; abline(v = 0)
for(k in 1:5){
   denstrip(unlist(out1J$sims.list[k]), at = k, ticks = out1J$summary[k, c(3,5,7)])
}
```

For comparison, let us fit the same model using method of least-squares, which for normal models yields estimates that are identical to those obtained using the more general ML method.

```
(fm <- summary(lm(Cmean ~ elev*forest)))
Call:
lm(formula = Cmean ~ elev * forest)
[ ... ]
Coefficients:
            Estimate  Std. Error  t value  Pr(>|t|)
(Intercept)   1.6603      0.1137   14.607    < 2e-16 ***
elev         -1.5765      0.2029   -7.771   1.76e-13 ***
forest        2.3440      0.1977   11.857    < 2e-16 ***
elev:forest  -0.8507      0.3540   -2.403      0.017 *
[ ... ]
Residual standard error: 1.849 on 263 degrees of freedom
Multiple R-squared:  0.447,     Adjusted R-squared:  0.4407
F-statistic: 70.86 on 3 and 263 DF,  p-value: < 2.2e-16
```

We observe numerically very similar estimates (posterior means and sds from WinBUGS and from JAGS and MLEs with SEs from `lm` in R; note that no SE is given for the dispersion parameter using `lm` in R).

```
print(cbind(out1B$summary[1:5, 1:2], out1J$summary[1:5, 1:2], rbind(fm$coef[,1:2],
c(fm$sigma, NA))), 4) # WB cols 1-2, JAGS cols 3-4, lm cols 5-6
           mean      sd    mean      sd  Estimate Std. Error
alpha0   1.6605 0.11523   1.660 0.11334   1.6603     0.1137
alpha1  -1.5765 0.20665  -1.575 0.20164  -1.5765     0.2029
alpha2   2.3450 0.19790   2.343 0.19852   2.3440     0.1977
alpha3  -0.8532 0.35456  -0.850 0.35716  -0.8507     0.3540
sd       1.8586 0.08223   1.858 0.08153   1.8495         NA
```

We see that when we use vague priors in a Bayesian analysis, we will typically obtain parameter estimates with great numerical resemblance to the MLEs. Moreover, the posterior standard deviations will be very similar numerically to the standard errors in the ML analysis. Thus, the fundamental thing is the model, which is the same whether analyzed in a non-Bayesian or in a Bayesian way. Whatever you may want to do with this model in a non-Bayesian analysis you can also do in a Bayesian analysis (such as, for instance, residual checks). In a frequentist analysis in R, you can do the following to do residual checks (and you will find out that there are too many large residuals for the assumed normal distribution):

```
plot(lm(Cmean ~ elev*forest))
```

In a Bayesian analysis, every unknown in a model is estimated and therefore has a posterior distribution. Residuals depend on the unknown parameters, and hence are unknown quantities themselves and have an entire posterior distribution. We can plot the residuals to check for normality visually, by seeing whether there is any evidence for lack of symmetry. We can also plot the residuals against their order in the data set and versus the predicted values for a visual check of variance homogeneity. These are three frequent residual diagnostic plots (Figure 5.3). For the predictions, we could simply have saved `mu` and then used its posterior mean, but since we saved the MCMC draws from its "ingredients" (the regression coefficients), we can compute the posterior mean of `mu` outside of BUGS in R.

```
mu <- out1B$mean$alpha0 + out1B$mean$alpha1 * elev + out1B$mean$alpha2 * forest +
out1B$mean$alpha3 * elev * forest        # Compute the posterior mean of mu

par(mfrow = c(2, 2), mar = c(5,4,2,2), cex.main = 1)
plot(1:M, out1B$summary[6:272, 1], xlab = "Order of values", ylab = "Residual",
frame.plot = F, ylim = c(-10, 15))
abline(h = 0, col = "red", lwd = 2)
segments(1:267, out1B$summary[6:272, 3], 1:267, out1B$summary[6:272, 7], col = "grey")
text(10, 14, "A", cex = 1.5)
hist(out1B$summary[6:272, 1], xlab = "Residual", main = "", breaks = 50, col = "grey",
xlim = c(-10, 15))
abline(v = 0, col = "red", lwd = 2)
text(-9, 48, "B", cex = 1.5)
qq <- qnorm(seq(0,0.9999,,data$M), mean = 0, sd = out1B$summary[5, 1])
plot(sort(qq), sort(out1B$summary[6:272, 1]), xlab = "Theoretical quantile", ylab =
"Residual", frame.plot = F, ylim = c(-10, 15)) # could also use qqnorm()
abline(0, 1, col = "red", lwd = 2)
text(-4.5, 14, "C", cex = 1.5)
```

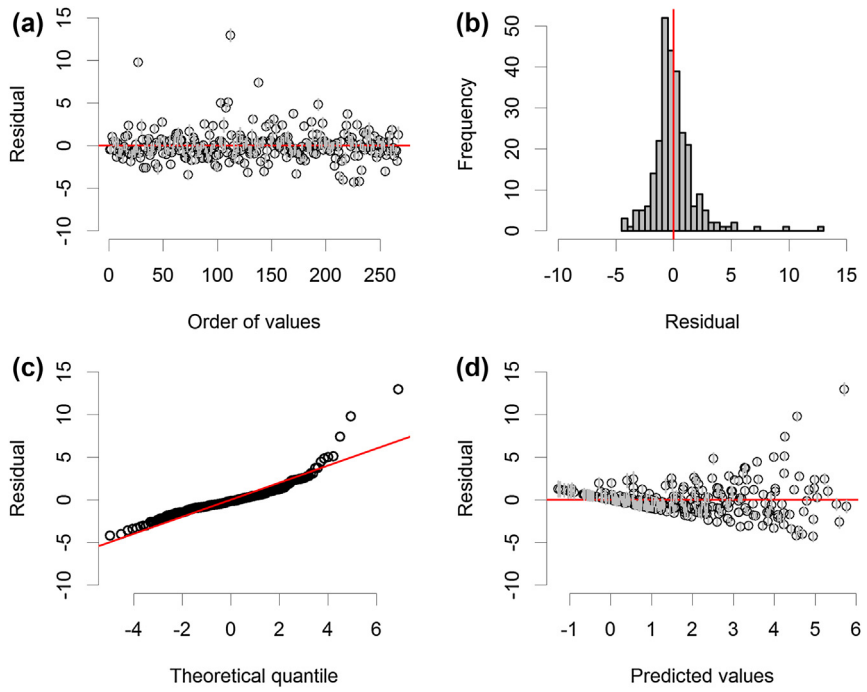

FIGURE 5.3

Residual diagnostic plots from the Bayesian analysis of the model: (a) residuals against their order, (b) histogram of residuals, (c) Q-Q plot, and (d) residuals against the fitted values. Red line shows zero, except in (c), where it is the 1:1 line.

```
plot(mu, out1B$summary[6:272, 1], xlab = "Predicted values", ylab = "Residual", frame.plot
= F, ylim = c(-10, 15))
abline(h = 0, col = "red", lwd = 2)
segments(mu, out1B$summary[6:272, 3], mu, out1B$summary[6:272, 7], col = "grey")
text(-1, 14, "D", cex = 1.5)
```

Clearly, the homoscedasticity assumption underlying the normal model is violated somewhat, since the residuals do not form a patternless cloud around a value of zero and the Q-Q plot shows deviations from a straight line. A more appropriate analysis might use log(counts), or better a Poisson GLM, for the counts directly. However, since we show this normal model for the mean great tit count for illustration only, we are not overly concerned with this potential structural problem. The important message we want to relay is simply that any model diagnostic that you can do for a frequentist analysis, you can also do with a Bayesian analysis of the same model.

Hence, we ignore the lack of model fit and continue our illustration of a Bayesian analysis. Two things that we often want to do are (1) make a statement about how certain we are that a parameter has a particular value and (2) make predictions—i.e., compute (with uncertainty assessment) what

response we would expect for one value or for an entire range of values of the covariates. We illustrate this here and again compare with an ML analysis, where we can do a significance test for each of the regression coefficients.

```
fm
[ ... ]
Coefficients:
            Estimate Std. Error t value  Pr(>|t|)
(Intercept)   1.6603     0.1137  14.607   < 2e-16 ***
elev         -1.5765     0.2029  -7.771  1.76e-13 ***
forest        2.3440     0.1977  11.857   < 2e-16 ***
elev:forest  -0.8507     0.3540  -2.403     0.017 *
```

We see that all three (and the intercept trivially so) are highly significant. We can also compute a confidence interval (CI), which has a somewhat contorted meaning in a non-Bayesian analysis: if we randomly sampled the same population of great tits many times, fitted our model, and computed a 95% CI each time, then 95% of these intervals would contain the true parameter value. Thus, the frequentist CI is an assessment of the reliability of a method, not a direct statement of uncertainty about an unknown quantity. In frequentist statistics, no probabilistic statements can be made about parameters, because parameters are not random variables. So, here are the CIs:

```
confint(lm(Cmean ~ elev*forest))
                 2.5 %     97.5 %
(Intercept)   1.436495   1.884119
elev         -1.975904  -1.177001
forest        1.954756   2.733260
elev:forest  -1.547748  -0.153619
```

In a Bayesian analysis, we use probability to express our degree of knowledge about uncertain quantities, such as the parameters in our regression model. This is the posterior distribution. Let us plot some posterior distributions and include the central range of the distributions containing 95% of their mass (Figure 5.4): this is the simplest way of computing a 95% CRI. Unlike a frequentist CI, a CRI is a direct probability statement about an unknown quantity; under the model, we can be 95% certain that the true parameter lies within this interval for the data set at hand.

```
par(mfrow = c(2, 2), mar = c(5,4,2,2), cex.main = 1)
hist(out1B$sims.list$alpha1, main = "", breaks = 100, col = "grey", freq=F)
abline(v = quantile(out1B$sims.list$alpha1, prob = c(0.025, 0.975)), col = "red", lwd = 2)
text( -2.4, 1.8, "A", cex = 1.5)
hist(out1B$sims.list$alpha2, main = "", breaks = 100, col = "grey", freq=F)
abline(v = quantile(out1B$sims.list$alpha2, prob = c(0.025, 0.975)), col = "red", lwd = 2)
text(1.7, 2, "B", cex = 1.5)
hist(out1B$sims.list$alpha3, main = "", breaks = 100, col = "grey", freq=F)
abline(v = quantile(out1B$sims.list$alpha3, prob = c(0.025, 0.975)), col = "red", lwd = 2)
text( -2.2, 1.2, "C", cex = 1.5)
hist(out1B$sims.list$sd, main = "", breaks = 100, col = "grey", freq=F)
abline(v = quantile(out1B$sims.list$sd, prob = c(0.025, 0.975)), col = "red", lwd = 2)
text(1.6, 4.9, "D", cex = 1.5)
```

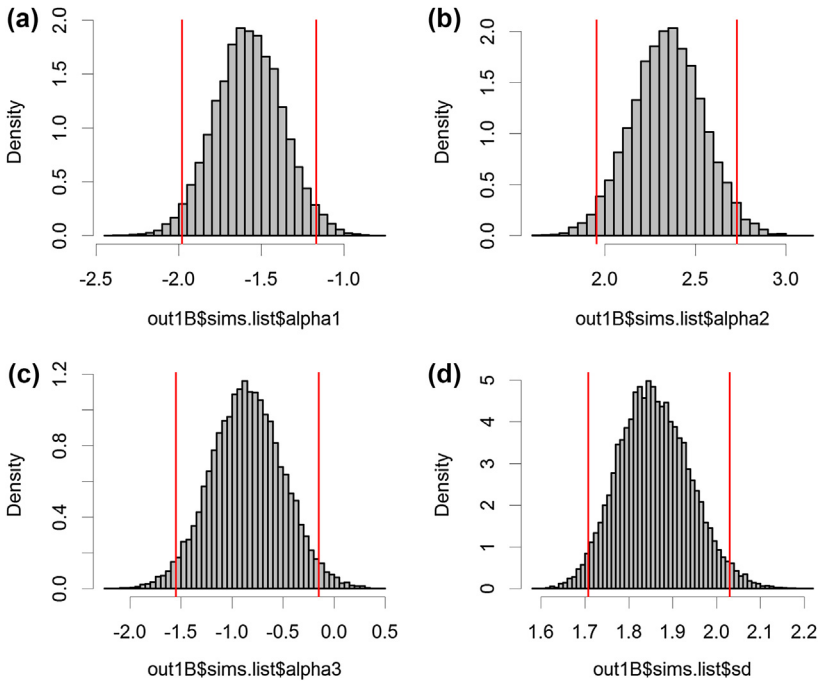

FIGURE 5.4

Marginal posterior distributions with percentile-based 95% CRIs (red) for four parameters in the normal model for mean tit counts. (a) Effect of elevation, (b) effect of forest cover, (c) interaction effect between elevation and forest cover, and (d) residual standard deviation.

Note that there are many different ways of constructing a 95% Bayesian CRI (Link and Barker, 2010), but the percentile method here is the easiest. One particular CRI is the highest-posterior density interval (HPDI), which is the shortest of such intervals, and can be computed using the `HPDinterval` function in the `coda` package.

```
HPDinterval(as.mcmc(out1B$sims.list$sd), prob = 0.95)  # HPDI
quantile(out1B$sims.list$sd, prob = c(0.025, 0.975))   # Percentile-based CRI
```

Although the fundamental meaning is quite different, CRIs in a Bayesian analysis can be used to do something analogous to a significance test in frequentist statistics: we see that zero is a fairly unlikely value for all four parameters shown and that zero is not included in the 95% CRI of any of them. Hence, we can say that we are quite certain that these parameters are different from zero. We would perhaps not want to say that the parameters are *significant*, because we feel that this term is associated too strongly with the frequentist technique of a significance test. Nevertheless, we often see Bayesians call this a significance test without even using quotes. This information is given in the posterior summary from `jags` in columns 6 and 7.

Although the posterior distribution is based on a degree-of-belief probability and does not have the same meaning as a probability ("long run frequency") in a frequentist analysis, Bayesian estimates are

often very well calibrated in a frequentist sense (Le Cam, 1990); i.e., their frequentist characteristics are often very good. That means, when replicated a great many times, a 95% Bayesian CRI will often contain the true value 95% of the time. Accordingly, frequentist CIs and Bayesian CRIs (when we use vague priors) are typically very similar numerically. Here they are side by side; the first two are the CI and the latter two form the CRI:

```
cbind(confint(lm(Cmean ~ elev*forest))[2:4,], out1B$summary[2:4, c(3,7)])
                2.5 %      97.5 %       2.5%      97.5%
elev        -1.975904   -1.177001  -1.981000  -1.167000
forest       1.954756    2.733260   1.954000   2.728025
elev:forest -1.547748   -0.153619 -1.551025  -0.150095
```

A particularly attractive feature when communicating the results of a Bayesian analysis is the ability to make direct probability statements about the magnitude of the parameters. For instance, from the posterior samples of `alpha1`, we can easily compute the probability that the slope of `elev` is more extreme than, say, –1.6, or that it lies between –1.8 and –1.6. Since the posterior is a probability distribution function and integrates to 1, the geometrical interpretation of these probabilities is simply the area under the posterior defined by these limits. e.g.,

```
mean(out1B$sims.list$alpha1 < -1.6)
[1]  0.4566
mean(out1B$sims.list$alpha1 < -1.6 & out1B$sims.list$alpha1 > -1.8)
[1]  0.3178667
```

Technically, what we obtain from the MCMC analysis is an estimate of a joint posterior distribution—i.e., the joint distribution of all estimated quantities. For n estimated quantities, you can imagine the joint posterior as a cloud in n-dimensional space. Let us look at this in an example in two dimensions only, for `alpha1` and `alpha2`. We can do "probability games" as before in two dimensions as well; e.g., we could test the hypothesis that the effect of `elev` lies between -1.9 and -1.6 and that of `forest` between 2.5 and 2.8. This is the proportion of samples that lie in the inner red square in Figure 5.5.

```
plot(out1B$sims.list$alpha1, out1B$sims.list$alpha2)
abline(h = c(2.5, 2.8), col = "red", lwd = 2)
abline(v = c(-1.9, -1.6), col = "red", lwd = 2)

mean(out1B$sims.list$alpha1 < -1.6 & out1B$sims.list$alpha1 > -1.9 &
out1B$sims.list$alpha2 > 2.5 & out1B$sims.list$alpha2 < 2.8)
0.08286667
```

Hence, that probability is about 8%. Thus, there is a lot of cool stuff that we can do with our posterior samples after running an MCMC algorithm that is not (easily) possible using a frequentist analysis. At many places in this book will we see examples of where we use posterior samples to compute derived quantities—e.g., functions of one or more parameters, by simply applying the function for each set of MCMC draws of the parameters. This can be done inside of the BUGS program, or outside, in R, if all the "ingredients" of the derived quantities have been sampled and saved.

Indeed, one of the neatest things in a Bayesian MCMC-based analysis is the ease with which such derived quantities can be obtained, along with a full assessment of their uncertainty. In non-Bayesian analyses, we have to use approximations like the delta rule (see Section 2.4.2) or else use bootstrapping

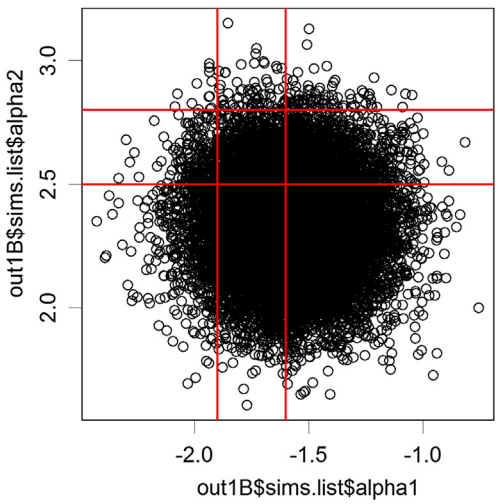

FIGURE 5.5

Joint posterior distribution of `alpha1` and `alpha2` with geometrical representation (red square) of the probability that the effect of `elev` lies between –1.9 and –1.6, and the effect of `forest` lies between 2.5 and 2.8.

(see Section 2.4.4), but in an MCMC-based analysis we can simply compute the function of interest for every step of the MCMC algorithm and then base the inference on the resulting posterior. To illustrate further, assume that we had some crazy theory that postulated the main effect of forest was more extreme than the main effect of elevation on the abundance of great tits. From the least-squares fit of the model, we can estimate the ratio between the two at `2.34/1.58 = 1.48`; this is not difficult. But what about the uncertainty associated with this estimate? And, is the observed ratio of the absolute effects "significantly" different from 1 (i.e., is one effect clearly greater than the other)?

Both estimates have an associated estimation uncertainty represented by their standard error (SE). In a simple case, it would not be too hard to work out the SE of the ratio or to bootstrap it, but in a Bayesian analysis the uncertainty of the ratio is readily obtained for models or derived quantities of any complexity. Figure 5.6 gives a picture of the posterior distribution of the absolute ratio between the effects of forest cover and elevation along with a 95% CRI (red). Since the value 1 is outside of the 95% CRI, we can thus say that the data are in agreement with the crazy theory.

```
crazy.ratio <- out1B$sims.list$alpha2 / abs(out1B$sims.list$alpha1)
hist(crazy.ratio, main = "", breaks = 100, col = "grey", freq = F)
abline(v = quantile(crazy.ratio, prob = c(0.025, 0.975)), col = "red", lwd = 3)

mean(abs(out1B$sims.list$alpha2 / out1B$sims.list$alpha1) > 1)
[1] 0.9964667
```

Thus, we can be nearly 100% certain that the effect of forest cover is more extreme than that of elevation. Hence, very rich inferences are possible and indeed readily obtainable based on the MCMC sample of posterior distributions.

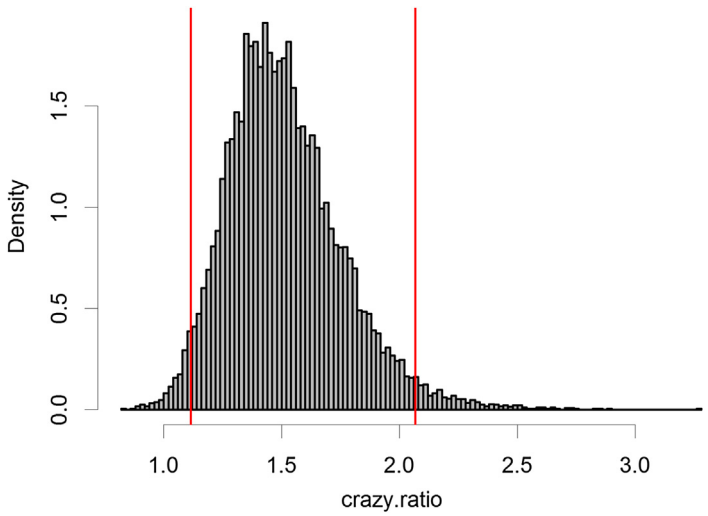

FIGURE 5.6

Posterior distribution of a crazy derived parameter, which is the absolute ratio of the effects of forest cover and elevation on the abundance of great tits, with 95% CRI shown with the red lines.

The final thing we illustrate here is the forming of predictions. Let us use our Bayesian parameter estimates to predict a response surface of the mean observed tit abundance as a function of elevation and forest cover and as a series of regression lines on elevation for selected values of forest cover (Figure 5.7(a) and (b)):

```
# Compute expected abundance for a grid of elevation and forest cover
elev.pred <- seq(-1, 1,,100)                    # Values of elevation
forest.pred <- seq(-1,1,,100)                   # Values of forest cover
pred.matrix <- array(NA, dim = c(100, 100))     # Prediction matrix
for(i in 1:100){
   for(j in 1:100){
      pred.matrix[i, j] <- out1J$mean$alpha0 + out1J$mean$alpha1 * elev.pred[i] +
out1J$mean$alpha2 * forest.pred[j] + out1J$mean$alpha3 * elev.pred[i] * forest.pred[j]
  }
}

par(mfrow = c(1, 3), mar = c(5,5,3,2), cex.main = 1.6, cex.axis = 1.5, cex.lab = 1.5)
mapPalette <- colorRampPalette(c("grey", "yellow", "orange", "red"))
image(x=elev.pred, y= forest.pred, z=pred.matrix, col = mapPalette(100), xlab =
"Elevation", ylab = "Forest cover")
contour(x=elev.pred, y=forest.pred, z=pred.matrix, add = TRUE, lwd = 1, cex = 1.5)
matpoints(elev, forest, pch="+", cex=1.5)
abline(h = c(-1, -0.5, 0, 0.5, 1))
```

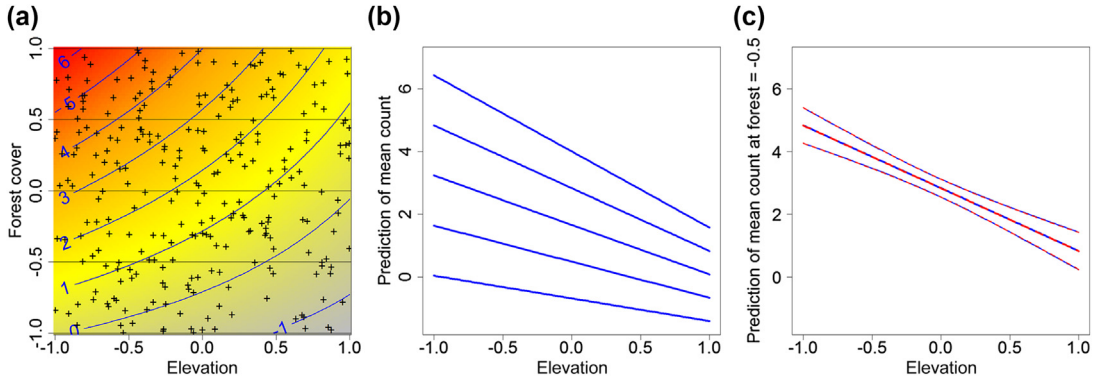

FIGURE 5.7

(a and b) Two ways of plotting predictions of the expected tit counts as a function of two continuous covariates: as a continuous response surface (a), and as a bundle of regression lines for a series of discrete values (here, for −1, −0.5, 0, 0.5, and 1 for the forest covariate) (b). Prediction with uncertainty (blue: Bayesian analysis; red: ML with 95% CRIs or CIs, respectively) is shown in (c). The curvature in (a) and nonparallelism in (b) indicate the presence of an interaction between elevation and forest cover. Note that the normal model for counts may predict impossible values such as negative tit counts in (a) and (b).

```
# Predictions for elev. at specific values of forest cover (-1,-0.5,0,0.5,1)
pred1 <- out1J$mean$alpha0 + out1J$mean$alpha1 * elev.pred + out1J$mean$alpha2 * (-1) +
out1J$mean$alpha3 * elev.pred * (-1)
pred2 <- out1J$mean$alpha0 + out1J$mean$alpha1 * elev.pred + out1J$mean$alpha2 * (-0.5) +
out1J$mean$alpha3 * elev.pred * (-0.5)
pred3 <- out1J$mean$alpha0 + out1J$mean$alpha1 * elev.pred + out1J$mean$alpha2 * 0 +
out1J$mean$alpha3 * elev.pred * 0
# pred3b <- out1J$mean$alpha0 + out1J$mean$alpha1 * elev.pred    # same
pred4 <- out1J$mean$alpha0 + out1J$mean$alpha1 * elev.pred + out1J$mean$alpha2 * 0.5 +
out1J$mean$alpha3 * elev.pred * 0.5
pred5 <- out1J$mean$alpha0 + out1J$mean$alpha1 * elev.pred + out1J$mean$alpha2 * 1 +
out1J$mean$alpha3 * elev.pred * 1
matplot(seq(-1,1,,100),cbind(pred1,pred2,pred3,pred4,pred5),type="l",lty=1,col=
"blue",ylab="Prediction of mean count",xlab="Elevation",ylim=c(-1.5,7),lwd=2)
```

These plots do not show the estimation uncertainty. This would be unwieldy in this type of plot, but it can easily be shown for simpler examples. We do this next, both for the Bayesian and the frequentist analysis, and plot the estimated relationship between the expected mean count of great tits and elevation at forest cover of 0.5 (Figure 5.7(c)). For this, we take all or a subset of the MCMC samples of the constituents of the predictions—i.e., the parameters `alpha0` through `alpha3`—and compute predictions using the values of the prediction covariate. We do this in a loop, but first prepare a matrix to contain the posterior samples for the predictions.

```
pred.mat <- array(dim = c(length(elev.pred), length(out1J$sims.list$alpha0)))
for(j in 1:length(out1J$sims.list$alpha0)){
   pred.mat[,j] <- out1J$sims.list$alpha0[j] + out1J$sims.list$alpha1[j] * elev.pred +
out1J$sims.list$alpha2[j] * 0.5 + out1J$sims.list$alpha3[j] * elev.pred * 0.5
}
```

To plot the 95% CRI of the Bayesian prediction we can use the point-wise 95% CRI:

```
CL <- apply(pred.mat, 1, function(x){quantile(x, prob = c(0.025, 0.975))})
plot(seq(-1, 1,,100), pred4, type = "l", lty= 1, col = "blue", ylab = "Prediction of mean
count at forest = -0.5", xlab = "Elevation", las =1, ylim = c(-1.5, 7), lwd = 3)
matlines(seq(-1, 1,,100), t(CL), lty = 1, col = "blue", lwd = 2)
```

We add the frequentist prediction along with the 95% confidence limits.

```
pred <- predict(lm(Cmean ~ elev*forest), newdata = data.frame(elev = seq(-1, 1,,100),
forest = 0.5), se.fit = TRUE, interval = "confidence")
lines(seq(-1, 1,,100), pred$fit[,1], lty= 2, col = "red", lwd = 3)
matlines(seq(-1, 1,,100), pred$fit[,2:3], lty = 2, col = "red", lwd = 2)
```

In real-life data analysis, what many find confusing is how predictions are presented for a covariate that has been transformed (e.g., scaled) before analysis. Then, we first have to form the predictions for the transformed covariates and then back-transform again for graphing. We show this in later parts of the book—e.g., in Section 5.9.

5.4 THE R PACKAGE rjags

We briefly illustrate model fitting in JAGS using the rjags package, including the production of trace plots and density plots, which is a fairly widely used way of running JAGS from R.

```
library(rjags)
load.module("glm")     # Careful with that package, see JAGS discussion list
load.module("dic")

# Have to explicitly list the deviance if want samples
params <- c("alpha0", "alpha1", "alpha2", "alpha3", "sd", "deviance")

# Adaptative phase to maximize MCMC efficiency
model <- jags.model(file = "multiple_linear_regression_model.txt", data = win.data,
inits = inits, n.chains = nc, n.adapt = 1000)

# Burnin
update(model, nb)

# Generate posterior samples
samples <- coda.samples(model = model, variable.names = params, n.iter = ni - nb, thin = nt)
```

```
# Get the summary statistics for the posterior samples
summfit <- summary(samples)
print(summfit, 2)
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
alpha0       1.66 0.114 0.00093         0.00093
alpha1      -1.58 0.204 0.00167         0.00170
alpha2       2.34 0.200 0.00163         0.00165
alpha3      -0.85 0.353 0.00288         0.00288
deviance  1087.14 3.237 0.02643         0.02923
sd           1.86 0.083 0.00068         0.00077

 2. Quantiles for each variable:

            2.5%     25%      50%      75%   97.5%
alpha0       1.4     1.6     1.66     1.74    1.88
alpha1      -2.0    -1.7    -1.58    -1.44   -1.18
alpha2       2.0     2.2     2.34     2.47    2.73
alpha3      -1.5    -1.1    -0.85    -0.62   -0.14
deviance  1082.9  1084.8 1086.48 1088.79 1095.23
sd           1.7     1.8     1.86     1.91    2.03

# Traceplots and posterior densities
plot(samples[,1:4])

# Compute the Brooks-Gelman-Rubin statistic (R-hat)
gelman.diag(samples)
Potential scale reduction factors:

          Point est. Upper C.I.
alpha0             1          1
alpha1             1          1
alpha2             1          1
alpha3             1          1
deviance           1          1
sd                 1          1

Multivariate psrf
1

# Compute the effective sample size
effectiveSize(samples)
  alpha0   alpha1   alpha2   alpha3 deviance       sd
15000.00 14541.42 14761.32 15000.00 12267.66 11601.70
```

```
# Secondary burnin can be applied (e.g. another 500 samples tossed out)
#samples <- window(samples, start = nb + 500 + 1, end = ni)

# More samples can be drawn (starting where the old chains stopped, not starting from 0)
newsamples <- coda.samples(model = model, variable.names = params, n.iter = 1500,
thin = nt)

# Combine the new samples with the old ones (ugly but works)
mc1 <- as.mcmc(rbind(samples[[1]], newsamples[[1]]))
mc2 <- as.mcmc(rbind(samples[[2]], newsamples[[2]]))
mc3 <- as.mcmc(rbind(samples[[3]], newsamples[[3]]))
allsamples <- as.mcmc.list(list(mc1, mc2, mc3))

# Mean deviance
Dbar <- summfit$statistics["deviance","Mean"]

# Variance of the deviance
varD <- summfit$statistics["deviance","SD"]^2

# Compute pD and DIC (according to A. Gelman, implemented in R2jags)
pD <- varD/2
DIC <- Dbar + pD

# Another DIC computation (according to M. Plummer). DIC = Penalized deviance
(dic.pD <- dic.samples(model, 2000, "pD"))

Mean deviance: 1088
penalty 3.993
Penalized deviance: 1092
```

Hence, we can readily use `rjags` directly for model fitting with JAGS or use a wrapper function like `jags` in `jagsUI`.

5.5 MISSING VALUES (NAs) IN A BAYESIAN ANALYSIS

How to deal with missing values (NAs) is the subject of a big field in statistics (Little and Rubin, 2002). An important distinction is between data that are missing at random (MAR) and those that are missing not at random (MNAR). With MAR, the probability for a datum to be missing does *not* depend on the value that it would have taken had it not been missing, and these data are therefore called ignorable missing values. The converse holds for MNAR, which are also called nonignorable missing data; to avoid biased inference, the missing-value-generating mechanism needs to be modeled for the data that are MNAR. An example for MAR data would be a study of the body mass in a population of mice where detection probability does not depend on body mass of an individual. In contrast, if heavier individuals are more likely to get caught and weighed, the missing value-generating mechanism—i.e., the mass-dependence of detection probability—needs to be modeled to avoid overestimating the population mean weight (Royle, 2008). In the following illustration, we assume data are MAR. Then the main difference in a Bayesian analysis is whether we have NAs in the response or in the covariates—i.e., whether the NAs are on the left- or right-hand side of the twiddle ($\sim$) in a BUGS model. The brief formula is then: a missing response is fine, but missing covariates must be dealt with.

5.5.1 SOME RESPONSES MISSING

When NAs are on the left of a twiddle, we can fit the same model and BUGS will simply update (=estimate, or predict) the missing responses. For illustration, we directly use code from Section 5.4. and analyze a modified data set where the first 10 responses (mean counts of great tits) are turned into NAs.

```
# Copy mean counts and turn first 10 into NAs
Cm <- Cmean        # Copy Cmean into Cm
Cm[1:10] <- NA     # turn first 10 into missing

# Bundle data (inside BUGS use Cm for Cmean)
win.data <- list(Cmean = Cm, M = length(Cm), elev = elev, forest = forest)
```

When we specify the response variable as a quantity to estimate, BUGS is (surprisingly) clever enough to know that only the missing elements of the response must be estimated, while the non-missing response data are fixed. Strangely though, in this case we have to set the argument `DIC = FALSE` to avoid the dreaded `"NIL dereference (read)"` crash in WinBUGS.

```
# Parameters monitored (i.e., for which estimates are saved)
params <- c("alpha0", "alpha1", "alpha2", "alpha3", "sd", "Cmean", "mu")

# ... or this to get a subset of the parameters
params <- c("alpha0", "alpha1", "alpha2", "alpha3", "sd", "Cmean[1:10]", "mu[1:10]")

# Call WinBUGS or JAGS from R (ART <1 min) and summarize posteriors
out1.1 <- bugs(win.data, inits, params, "multiple_linear_regression_model.txt",
n.chains = nc, n.thin = nt, n.iter = ni, n.burnin = nb, debug = TRUE, bugs.directory =
bugs.dir, working.directory = getwd(), DIC = FALSE)

out1.1 <- jags(win.data, inits, params, "multiple_linear_regression_model.txt",
n.chains = nc, n.thin = nt, n.iter = ni, n.burnin = nb)

print(out1.1, 2)
JAGS output for model 'multiple_linear_regression_model.txt', generated by jagsUI.
Estimates based on 3 chains of 6000 iterations,
burn-in = 1000 iterations and thin rate = 1,
yielding 15000 total samples from the joint posterior.
MCMC ran for 0.072 minutes at time 2014-09-20 14:53:55.

           mean   sd   2.5%    50%  97.5% overlap0    f  Rhat  n.eff
alpha0     1.67 0.12   1.44   1.67   1.90    FALSE 1.00     1  15000
alpha1    -1.59 0.21  -2.00  -1.59  -1.17    FALSE 1.00     1  15000
alpha2     2.34 0.21   1.93   2.34   2.75    FALSE 1.00     1   4321
alpha3    -0.87 0.37  -1.60  -0.87  -0.15    FALSE 0.99     1  10679
sd         1.89 0.08   1.73   1.89   2.06    FALSE 1.00     1   4599
Cmean[1]   3.48 1.89  -0.26   3.47   7.20     TRUE 0.97     1  15000
Cmean[2]   3.79 1.89   0.12   3.77   7.57    FALSE 0.98     1  15000
Cmean[3]   2.95 1.92  -0.81   2.96   6.70     TRUE 0.94     1  15000
Cmean[4]  -0.44 1.91  -4.15  -0.42   3.30     TRUE 0.59     1   7555
```

```
Cmean[5]   0.56 1.90   -3.20    0.56    4.25    TRUE 0.61     1 15000
Cmean[6]  -0.41 1.90   -4.12   -0.40    3.36    TRUE 0.58     1 15000
Cmean[7]   3.35 1.91   -0.41    3.35    7.09    TRUE 0.96     1 15000
Cmean[8]   2.37 1.92   -1.37    2.38    6.08    TRUE 0.89     1 15000
Cmean[9]   0.87 1.90   -2.83    0.88    4.61    TRUE 0.68     1 15000
Cmean[10]  0.93 1.90   -2.84    0.93    4.61    TRUE 0.69     1 15000
mu[1]      3.49 0.17    3.15    3.49    3.83   FALSE 1.00     1 15000
mu[2]      3.80 0.20    3.41    3.80    4.18   FALSE 1.00     1 15000
mu[3]      2.94 0.29    2.37    2.95    3.50   FALSE 1.00     1  5161
mu[4]     -0.45 0.23   -0.90   -0.45   -0.01   FALSE 0.98     1  6941
mu[5]      0.58 0.16    0.26    0.58    0.90   FALSE 1.00     1 15000
mu[6]     -0.43 0.30   -1.02   -0.43    0.17    TRUE 0.92     1 15000
mu[7]      3.36 0.17    3.03    3.36    3.68   FALSE 1.00     1 15000
mu[8]      2.38 0.28    1.82    2.38    2.93   FALSE 1.00     1  5856
mu[9]      0.88 0.18    0.53    0.88    1.23   FALSE 1.00     1 15000
mu[10]     0.92 0.21    0.51    0.92    1.34   FALSE 1.00     1 15000
```

Where does the information come from to estimate the missing responses `Cmean[1:10]`? As an aside, what is the difference between `Cmean[1:10]` and `mu[1:10]`? Let us compare the true values of the first 10 responses with their estimates for `Cmean[1:10]` and expectation `mu[1:10]`.

```
print(cbind(Truth=Cmean[1:10], out1.1$summary[6:15,c(1:3,7)], out1.1$summary[16:25,
c(1:3,7)]),3)
          Truth    mean    sd    2.5% 97.5%    mean     sd    2.5%    97.5%
Cmean[1]  3.000   3.476  1.89 -0.257  7.20   3.494  0.172   3.154   3.8321
Cmean[2]  3.333   3.792  1.89  0.117  7.57   3.800  0.196   3.412   4.1835
Cmean[3]  4.000   2.952  1.92 -0.813  6.70   2.942  0.288   2.373   3.4976
Cmean[4]  0.000  -0.436  1.91 -4.154  3.30  -0.453  0.227  -0.903  -0.0107
Cmean[5]  0.000   0.559  1.90 -3.201  4.25   0.581  0.162   0.263   0.8968
Cmean[6]  0.000  -0.414  1.90 -4.120  3.36  -0.429  0.302  -1.016   0.1666
Cmean[7]  2.000   3.347  1.91 -0.405  7.09   3.355  0.167   3.026   3.6806
Cmean[8]  2.667   2.367  1.92 -1.374  6.08   2.381  0.284   1.822   2.9327
Cmean[9]  0.000   0.873  1.90 -2.834  4.61   0.880  0.179   0.531   1.2335
Cmean[10] 0.333   0.926  1.90 -2.842  4.61   0.923  0.215   0.506   1.3435
```

To understand the difference between `Cmean` and `mu`, let us look at the relationship between the realized response $Cmean_i$ and the expected response μ_i (see Section 5.3).

$$Cmean_i \sim Normal(\mu_i, \sigma^2)$$

The information to estimate a missing response `Cmean` comes from the estimated regression relationship between the counts and the covariates elevation and forest: $\mu_i = \alpha_0 + \alpha_1 * elev_i + \alpha_2 * forest_i + \alpha_3 * elev_i * forest_i$. Based on that and the known covariate values, we can obtain an estimate for the missing responses. The difference between `mu` (μ_i) and `Cmean` ($Cmean_i$) is that between the expected response and the realized response, respectively. Up to Monte Carlo error, the point estimates of these quantities should be identical. However, the uncertainty about μ_i stems only

from having to estimate the regression parameters α_0, α_1, α_2, and α_3, whereas the estimate for a realized response $Cmean_i$ also contains a contribution of the sampling variability of the data represented by the unknown residual (and the variance parameter σ^2) and the estimation uncertainty in σ^2. Hence, estimates of $Cmean_i$ are more variable than are those of the mean response μ_i. Clearly, even if we knew the expected response (μ_i), we would still be uncertain about the actual realized response. The 95% uncertainty interval around `Cmean` is also called a prediction interval, while that around `mu` is called a confidence interval (or, in the Bayesian analysis here, a credible interval).

The ability in a Bayesian analysis to produce estimates of the response variable provides us with a way to produce hypothetical replicate data sets under our model, which is at the root of a very general Bayesian GoF assessment technique; see Section 5.10. Note that the parameter estimates would have been the same if we had tossed out the data for the 10 NA responses altogether, since these data units do not contain any information about the regression relationship. This is the same as when we make predictions.

5.5.2 ALL RESPONSES MISSING

Next, let us look at an extreme case and fit the model when *all* responses are turned into NAs. This extreme case reiterates that in a Bayesian analysis the "result", i.e., the posterior distribution, is affected by the information in the data *and* that in the prior distribution. If there is no data set, parameter estimates are simply informed by the priors. In complex HMs, this can be useful to gauge the induced priors for those parameters for which we do not specify priors directly. We illustrate by monitoring the expected response `mu` (μ) for the first two data points.

```
# Bundle data: simply drop the response from list
win.data <- list(M = length(Cm), elev = elev, forest = forest)

# Alternatively, add all-NA data vector
win.data$Cmean <- as.numeric(rep(NA, length(Cmean)))
str(win.data)  # Cmean is numeric

# Parameters monitored
params <- c("alpha0", "alpha1", "alpha2", "alpha3", "sd", "mu[1:2]")

# Call WinBUGS from R (ART <1 min) and summarize posteriors
out1.2 <- bugs(win.data, inits, params, "multiple_linear_regression_model.txt",
n.chains = nc, n.thin = nt, n.iter = ni, n.burnin = nb, debug = TRUE, bugs.directory =
bugs.dir, working.directory = getwd(), DIC = FALSE)

print(out1.2, 2)
         mean      sd      2.5%      25%      50%     75%   97.5% Rhat n.eff
alpha0 -15.12 1003.24  -1976.05  -703.97  -15.59  663.62 1936.00    1 15000
alpha1   2.46  990.95  -1946.00  -656.18    5.81  674.78 1957.05    1 15000
alpha2  -8.87 1004.61  -1957.00  -690.30   -5.06  662.00 1953.05    1 15000
alpha3 -14.24  984.28  -1948.03  -684.70   -4.48  649.35 1886.00    1  3900
sd     500.49  288.57     25.50   251.30  504.45  749.20  976.10    1 15000
mu[1]  -17.42 1183.05  -2350.03  -814.40  -24.67  784.02 2312.02    1 10000
mu[2]  -16.93 1252.07  -2475.00  -863.08  -19.27  826.00 2449.07    1  9500
```

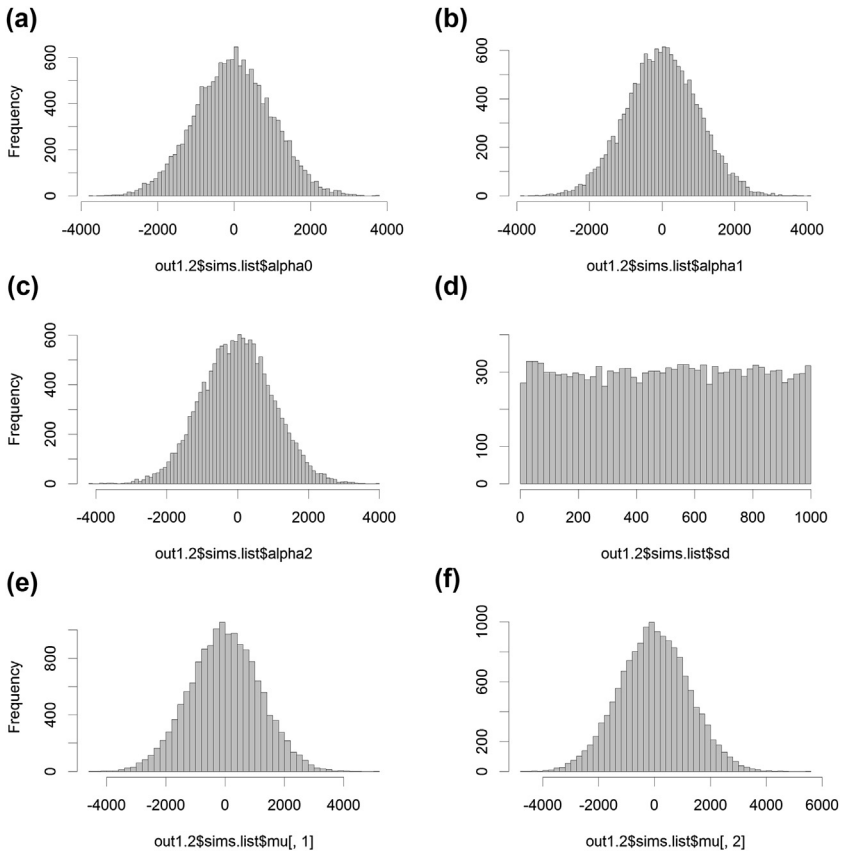

FIGURE 5.8

Induced priors can be studied by fitting a model with the response data unlinked from the analysis. For the first four parameters (a–d), priors are specified directly in the analysis, while for the last two (e–f), priors are induced.

The posterior distributions for the model parameters are directly the priors specified in the model, while those for `mu` are induced (Figure 5.8).

```
par(mfrow = c(3, 2), mar = c(5,5,3,2), cex.lab = 1.5, cex.axis = 1.5)
hist(out1.2$sims.list$alpha0, breaks = 100, col = "grey", main = "")
hist(out1.2$sims.list$alpha1, breaks = 100, col = "grey", main = "", ylab = "")
hist(out1.2$sims.list$alpha2, breaks = 100, col = "grey", main = "")
hist(out1.2$sims.list$sd, breaks = 100, col = "grey", main = "", ylim = c(0, 230),
ylab = "")
hist(out1.2$sims.list$mu[,1], breaks = 100, col = "grey", main = "")
hist(out1.2$sims.list$mu[,2], breaks = 100, col = "grey", main = "", ylab = "")
```

5.5.3 MISSING VALUES IN A COVARIATE

When we have missing covariate values in our model, we cannot simply use the model to estimate them; there is no information available to estimate them and BUGS would crash with an "undefined node" error. Heuristically, in order to estimate missing covariates, we must add code so that somewhere in the model the missing covariate appears on the left-hand side of a twiddle—i.e., as a modeled quantity. Here, we only mention some of the simplest manners of dealing with missing covariates. The simplest of all (method 1) is often called mean imputation, where the NAs are replaced with the covariate mean. This method ignores the uncertainty stemming from having to estimate the missing covariates, and it does not use information about the association of the missing covariates with observed responses or observed values of the covariate. A slightly more refined method (2) is to estimate the missing covariate values as unrelated draws from a vague prior. This does incorporate the estimation uncertainty and the information stemming from the (possibly) observed response for a missing covariate. Finally, a third method is to specify a prior for all (observed and unobserved) covariate values and estimate hyperparameters of that prior. This method is the most refined and exploits information coming from both the observed values of the response and the observed covariates. Next, we show methods 2 and 3.

```
# Shoot 'holes' in the covariate data
ele <- elev        # copy of elevation covariate
ele[1:10] <- NA    # create some missing values in covariate elevation

# Bundle data: feed new 'ele' into 'elev' covariate inside of BUGS model
win.data <- list(Cmean = Cmean, M = length(Cmean), elev = ele, forest = forest)

# Specify model in BUGS language
cat(file = "missing_cov_imputation_model_1.txt","
model {

# Priors
alpha0 ~ dnorm(0, 1.0E-06)        # Prior for intercept
alpha1 ~ dnorm(0, 1.0E-06)        # Prior for slope of elev
alpha2 ~ dnorm(0, 1.0E-06)        # Prior for slope of forest
alpha3 ~ dnorm(0, 1.0E-06)        # Prior for slope of interaction
tau <- pow(sd, -2)
sd ~ dunif(0, 1000)               # Prior for dispersion on sd scale

# Likelihood
for (i in 1:M){
   Cmean[i] ~ dnorm(mu[i], tau)   # precision tau = 1 / variance
   mu[i] <- alpha0 + alpha1 * elev[i] + alpha2 * forest[i] + alpha3 * elev[i] * forest[i]
}

# Model for missing covariates
for (i in 1:M){
  elev[i] ~ dnorm(0, 0.001)
}
}")
```

```
# Initial values
inits <- function() list(alpha0 = rnorm(1,,10), alpha1 = rnorm(1,,10), alpha2 =
rnorm(1,,10), alpha3 = rnorm(1,,10))

# Parameters monitored
params <- c("alpha0", "alpha1", "alpha2", "alpha3", "sd", "elev")

# MCMC settings
ni <- 6000  ;  nt <- 1  ;  nb <- 1000  ; nc <- 3

# Call WinBUGS from R (ART <1 min)
out1.3 <- bugs(win.data, inits, params, "missing_cov_imputation_model_1.txt",
n.chains = nc, n.thin = nt, n.iter = ni, n.burnin = nb, debug = TRUE,
bugs.directory = bugs.dir, working.directory = getwd())
```

Method 2 estimates each NA of elevation as an unrelated quantity. In method 3, next, we additionally specify a model for the covariates, thereby the observed values of the covariate will contribute to estimation of the NAs of the covariate.

```
# Specify model in BUGS language
cat(file = "missing_cov_imputation_model_2.txt","
model {

# Priors
alpha0 ~ dnorm(0, 1.0E-06)            # Prior for intercept
alpha1 ~ dnorm(0, 1.0E-06)            # Prior for slope of elev
alpha2 ~ dnorm(0, 1.0E-06)            # Prior for slope of forest
alpha3 ~ dnorm(0, 1.0E-06)            # Prior for slope of interaction
tau <- pow(sd, -2)
sd ~ dunif(0, 1000)                   # Prior for dispersion on sd scale

# Likelihood
for (i in 1:M){
  Cmean[i] ~ dnorm(mu[i], tau)        # precision tau = 1 / variance
  mu[i] <- alpha0 + alpha1 * elev[i] + alpha2 * forest[i] + alpha3 * elev[i] * forest[i]
}

# Covariate mean as a model for missing covariates
for (i in 1:M){
 elev[i] ~ dnorm(mu.elev, tau.elev)  # Assume elevation normally distributed
}
mu.elev ~ dnorm(0, 0.0001)
tau.elev <- pow(sd.elev, -2)
sd.elev ~ dunif(0, 100)
}")

# Initial values
inits <- function() list(alpha0 = rnorm(1,,10), alpha1 = rnorm(1,,10), alpha2 =
rnorm(1,,10), alpha3 = rnorm(1,,10))
```

```
# Parameters monitored
params <- c("alpha0", "alpha1", "alpha2", "alpha3", "sd", "elev", "mu.elev", "sd.elev")

# MCMC settings
ni <- 6000  ;  nt <- 1  ;  nb <- 1000  ;  nc <- 3

# Call WinBUGS from R (ART <1 min)
out1.4 <- bugs(win.data, inits, params, "missing_cov_imputation_model_2.txt",
n.chains = nc, n.thin = nt, n.iter = ni, n.burnin = nb, debug = TRUE,
bugs.directory = bugs.dir, working.directory = getwd())
```

We compare the two approaches graphically (Figure 5.9). Both sets of estimates are about equally far away from the truth, but we see that when we can come up with a reasonable model for the covariates, we can obtain estimates with much reduced uncertainty.

```
par(cex = 1.5, lwd = 2)
plot(elev[1:10]-0.01, out1.3$summary[6:15,1], ylim = c(-10, 10), col = "red",
xlab = "True value of covariate", ylab = "Estimate (with 95% CRI)", frame.plot =F)
segments(elev[1:10]-0.01, out1.3$summary[6:15,3], elev[1:10]-0.01,
out1.3$summary[6:15,7], col = "red")
points(elev[1:10]+0.01, out1.4$summary[6:15,1], ylim = c(-3, 3), col = "blue")
segments(elev[1:10]+0.01, out1.4$summary[6:15,3], elev[1:10]+0.01,
out1.4$summary[6:15,7], col = "blue")
abline(0,1)
```

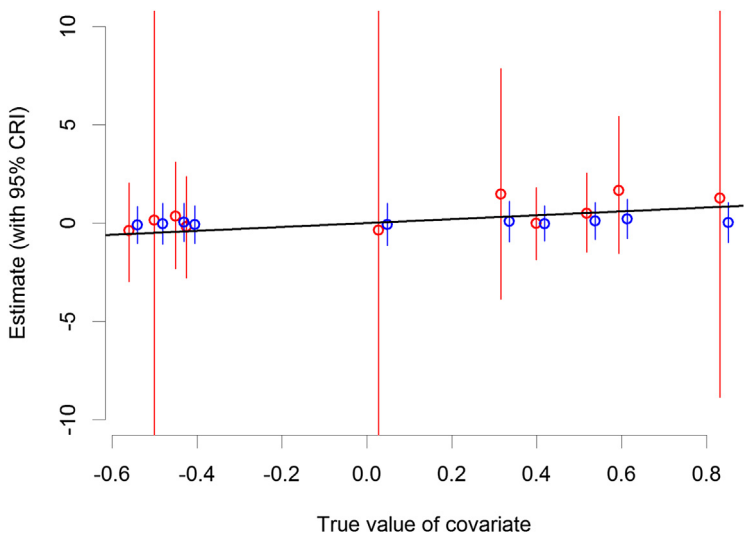

FIGURE 5.9

Estimates of missing covariates (with 95% CRI) when no model is specified for the covariate apart from a vague prior (red), and when missing covariate values are assumed exchangeable with the observed covariate values by specifying a normal prior with mean and variance estimated from the available covariate values (blue).

5.6 LINEAR MODEL WITH NORMAL RESPONSE (NORMAL GLM): ANALYSIS OF COVARIANCE (ANCOVA)

Returning to the illustration of common linear models, we next use BUGS to fit a linear model that underlies a technique called analysis of covariance (ANCOVA). Specifically, within a GLM with normal response, we fit to the mean tit counts the linear model underlying a fixed-effects ANCOVA with interaction effects. For this, we somewhat artificially first construct a factor that classifies the continuous covariate `forest` cover into four levels or groups, with level 1 for values between −1 and −0.5, level 2 corresponding to −0.49 and 0, etc.; see Figure 5.10(a) for the raw relationship between mean tit count and levels of the forest factor (`facFor`). Factors in BUGS must be labeled with integer numbers and not, for instance, with letters or words, and the numbering must start at 1 and end at the number of levels—i.e., have no jumps (e.g., 1, 2, 4, and 5 would cause a crash). We fit the following model in the effects and the means parameterization (see Chapter 3), where j indexes the four levels of the forest factor:

$$Cmean_i \sim Normal(\mu_i, \sigma^2)$$

$$\mu_i = \alpha_{0,j} + \alpha_{1,j} * elev_i$$

```
# Generate factor and plot raw data in boxplot as function of factor A
facFor <- as.numeric(forest < -0.5)      # Factor level 1
facFor[forest < 0 & forest > -0.5] <- 2  # Factor level 2
facFor[forest < 0.5 & forest > 0] <- 3   # Factor level 3
facFor[forest > 0.5] <- 4                # Factor level 4
table(facFor)                            # every site assigned a level OK
```

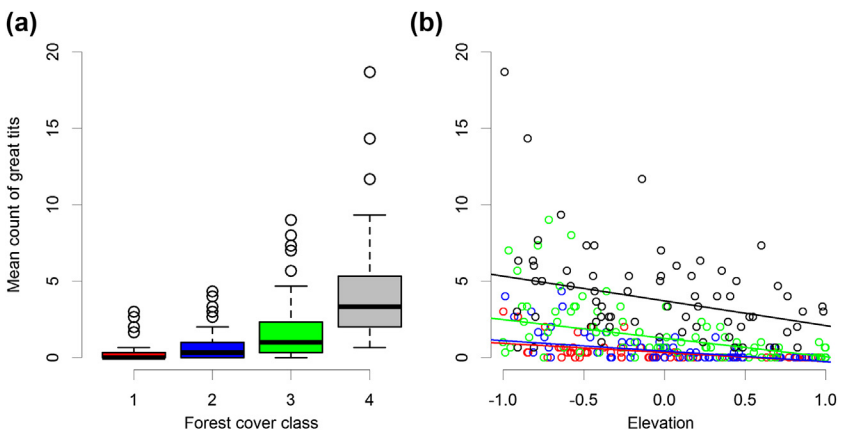

FIGURE 5.10

(a) Relationship between the mean count of great tits and the levels of the forest cover factor (`facFor`). Raw data are shown for each level of `facFor`. (b) Raw data and predicted relationship with elevation under the ANCOVA model with a least-squares fit. Colors denote the four levels of `facFor`.

```
par(mfrow = c(1, 2), mar = c(5,5,3,2), cex.lab = 1.5, cex.axis = 1.5)
plot(Cmean ~ factor(facFor), col = c("red", "blue", "green", "grey"), xlab = "Forest cover
class", ylab = "Mean count of great tits", frame.plot = F, ylim = c(0,20))

# Bundle data
win.data <- list(Cmean = Cmean, M = length(Cmean), elev = elev, facFor = facFor)
```

We can define the model in the effects or the means parameterization, and we show both. In either case, we define vector-valued parameters using the handy *nested indexing* in the BUGS language. We will fit the model in WinBUGS, JAGS, and compare with the MLEs obtained by using the least-squares method by way of the function `lm` in R.

```
# Specify model in BUGS language in effects parameterization
cat(file = "ANCOVA1.txt","
model {

# Priors
alpha ~ dnorm(0, 1.0E-06)       # Prior for intercept = effect of level 1 of forest factor
beta2 ~ dnorm(0, 1.0E-06)       # Prior for slope = effect of elevation for level 1 of forest
factor
beta1[1] <- 0                   # Set to zero effect of first level of facFor
beta3[1] <- 0                   # Set to zero effect of first level of facFor of elevation
for(k in 2:4){
 beta1[k] ~ dnorm(0, 1.0E-06)   # Prior for effects of factor facFor
 beta3[k] ~ dnorm(0, 1.0E-06)   # Prior for effects of factor facFor
}
tau <- pow(sd, -2)
sd ~ dunif(0, 1000)             # Prior for dispersion on sd scale

# Likelihood
for (i in 1:M){
 Cmean[i] ~ dnorm(mu[i], tau)   # precision tau = 1 / variance
 mu[i] <- alpha + beta1[facFor[i]] + beta2 * elev[i] + beta3[facFor[i]] * elev[i]
}
}
")
```

We must not give any initial values for fixed quantities (here, `beta1[1]` and `beta3[1]`); note that in place of the initial for the first element of the parameter vectors `beta1` and `beta3`, we have an "NA."

```
# Initial values
inits <- function() list(alpha = rnorm(1,,10), beta1 = c(NA, rnorm(3,,10)), beta2 =
rnorm(1,,10), beta3 = c(NA, rnorm(3,,10)))

# Parameters monitored
params <- c("alpha", "beta1", "beta2", "beta3", "sd")

# MCMC settings
ni <- 6000  ;  nt <- 1  ;  nb <- 1000  ; nc <- 3
```

```
# Call WinBUGS or JAGS from R (ART <1 min)
out3 <- bugs(win.data, inits, params, "ANCOVA1.txt", n.chains = nc, n.thin = nt, n.iter =
ni, n.burnin = nb, debug = TRUE, bugs.directory = bugs.dir, working.directory = getwd())

out3J <- jags(win.data, inits, params, "ANCOVA1.txt", n.chains = nc, n.thin = nt,
n.iter = ni, n.burnin = nb)
# traceplot(out3J)

# Fit model using least-squares (yields equivalent estimates as MLE)
(fm <- summary(lm(Cmean ~ as.factor(facFor)*elev)))

Coefficients:
                         Estimate Std. Error  t value Pr(>|t|)
(Intercept)               0.3353     0.2301    1.457  0.14633
as.factor(facFor)2        0.4244     0.3231    1.313  0.19028
as.factor(facFor)3        1.2690     0.3083    4.115  5.2e-05 ***
as.factor(facFor)4        3.7205     0.3162   11.766  < 2e-16 ***
elev                     -0.6013     0.4203   -1.431  0.15377
as.factor(facFor)2:elev  -0.6866     0.5999   -1.145  0.25345
as.factor(facFor)3:elev  -1.2116     0.5427   -2.232  0.02644 *
as.factor(facFor)4:elev  -1.6164     0.5708   -2.832  0.00499 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.783 on 259 degrees of freedom
Multiple R-squared:  0.4941,   Adjusted R-squared:  0.4804
F-statistic: 36.13 on 7 and 259 DF,  p-value: < 2.2e-16

# Summarize posteriors
print(out3, 3)
              mean    sd      2.5%       25%       50%       75%     97.5%   Rhat n.eff
alpha        0.337 0.231    -0.114     0.183     0.337     0.491     0.799  1.001 15000
beta1[2]     0.422 0.324    -0.211     0.205     0.422     0.641     1.051  1.001 15000
beta1[3]     1.268 0.310     0.664     1.062     1.267     1.473     1.887  1.001 15000
beta1[4]     3.721 0.318     3.093     3.509     3.721     3.931     4.350  1.001 15000
beta2       -0.602 0.421    -1.442    -0.885    -0.600    -0.319     0.222  1.001 15000
beta3[2]    -0.687 0.605    -1.859    -1.101    -0.692    -0.277     0.503  1.001 15000
beta3[3]    -1.215 0.544    -2.290    -1.581    -1.218    -0.847    -0.158  1.001 15000
beta3[4]    -1.611 0.578    -2.744    -1.999    -1.610    -1.223    -0.456  1.001  6100
sd           1.791 0.078     1.648     1.737     1.788     1.842     1.953  1.001 15000
deviance  1067.483 4.339  1061.000  1064.000  1067.000  1070.000  1078.000  1.001 15000

DIC info (using the rule, pD = Dbar-Dhat)
pD = 9.0 and DIC = 1076.5
DIC is an estimate of expected predictive error (lower deviance is better).
```

We see the usual close numerical agreement between the Bayesian estimates and the MLEs obtained with function `lm` in R. Next, we fit the model using the means parameterization, where we fit directly the effect of each level of factor `facFor` (note the changed parameter naming in the output). We do not need any change in the data bundle. In addition, we also illustrate how we can estimate

custom contrasts as derived quantities—i.e., differences or other functions of parameters. We estimate all pair-wise differences between the group means `beta[1:4]`. Of course, we could also easily compute these derived quantities in R using posterior samples of the vector `beta` produced by BUGS.

```
# Specify model in BUGS language
cat(file = "ANCOVA2.txt","
model {

# Priors
for(k in 1:4){
  alpha[k] ~ dnorm(0, 1.0E-06)   # Priors for intercepts
  beta[k] ~ dnorm(0, 1.0E-06)    # Priors for slopes
}
tau <- pow(sd, -2)
sd ~ dunif(0, 1000)              # Prior for dispersion on sd scale

# Likelihood
for (i in 1:M){
 Cmean[i] ~ dnorm(mu[i], tau)    # precision tau = 1 / variance
 mu[i] <- alpha[facFor[i]] + beta[facFor[i]] * elev[i]
}

# Derived quantities: comparison of slopes (now you can forget the delta rule !)
for(k in 1:4){
  diff.vs1[k] <- beta[k] - beta[1]  # Differences relative to beta[1]
  diff.vs2[k] <- beta[k] - beta[2]  # ... relative to beta[2]
  diff.vs3[k] <- beta[k] - beta[3]  # ... relative to beta[3]
  diff.vs4[k] <- beta[k] - beta[4]  # ... relative to beta[4]
}
}
")

# Initial values
inits <- function() list(alpha = rnorm(4,,10), beta = rnorm(4,,10))

# Parameters monitored
params <- c("alpha", "beta", "sd", "diff.vs1", "diff.vs2", "diff.vs3", "diff.vs4")

# MCMC settings
ni <- 6000  ;  nt <- 1  ;  nb <- 1000  ; nc <- 3

# Call WinBUGS or JAGS from R (ART <1 min) and summarize posteriors
out4 <- bugs(win.data, inits, params, "ANCOVA2.txt", n.chains = nc, n.thin = nt,
n.iter = ni, n.burnin = nb, debug = TRUE, bugs.directory = bugs.dir,
working.directory = getwd())

system.time(out4J <- jags(win.data, inits, params, "ANCOVA2.txt", n.chains = nc,
n.thin = nt, n.iter = ni, n.burnin = nb))
traceplot(out4J)

print(out4, 2)
Inference for Bugs model at "ANCOVA2.txt", fit using WinBUGS,
```

```
Current: 3 chains, each with 6000 iterations (first 1000 discarded)
Cumulative: n.sims = 15000 iterations saved
              mean    sd   2.5%    25%    50%    75%  97.5%  Rhat  n.eff
alpha[1]      0.33  0.23  -0.13   0.18   0.33   0.49   0.79     1  15000
alpha[2]      0.76  0.23   0.31   0.61   0.76   0.91   1.21     1  15000
alpha[3]      1.60  0.21   1.19   1.46   1.60   1.74   2.00     1  15000
alpha[4]      4.06  0.22   3.64   3.91   4.05   4.20   4.49     1  15000
beta[1]      -0.60  0.42  -1.41  -0.88  -0.60  -0.31   0.23     1  15000
beta[2]      -1.29  0.43  -2.14  -1.58  -1.29  -0.99  -0.43     1  15000
beta[3]      -1.82  0.34  -2.49  -2.05  -1.82  -1.59  -1.13     1   6700
beta[4]      -2.22  0.39  -2.97  -2.48  -2.22  -1.96  -1.45     1  15000
sd            1.79  0.08   1.65   1.74   1.79   1.84   1.96     1   9900
diff.vs1[1]   0.00  0.00   0.00   0.00   0.00   0.00   0.00     1      1
diff.vs1[2]  -0.69  0.60  -1.88  -1.10  -0.69  -0.28   0.48     1  15000
diff.vs1[3]  -1.22  0.55  -2.30  -1.58  -1.22  -0.86  -0.15     1  14000
diff.vs1[4]  -1.62  0.57  -2.73  -2.01  -1.63  -1.23  -0.49     1  15000
diff.vs2[1]   0.69  0.60  -0.48   0.28   0.69   1.10   1.88     1  15000
diff.vs2[2]   0.00  0.00   0.00   0.00   0.00   0.00   0.00     1      1
diff.vs2[3]  -0.53  0.56  -1.63  -0.90  -0.53  -0.16   0.57     1  15000
diff.vs2[4]  -0.93  0.58  -2.07  -1.32  -0.93  -0.54   0.21     1  15000
diff.vs3[1]   1.22  0.55   0.15   0.86   1.22   1.58   2.30     1  14000
diff.vs3[2]   0.53  0.56  -0.57   0.16   0.53   0.90   1.63     1  15000
diff.vs3[3]   0.00  0.00   0.00   0.00   0.00   0.00   0.00     1      1
diff.vs3[4]  -0.40  0.52  -1.42  -0.75  -0.40  -0.05   0.62     1   9700
diff.vs4[1]   1.62  0.57   0.49   1.23   1.63   2.01   2.73     1  15000
diff.vs4[2]   0.93  0.58  -0.21   0.54   0.93   1.32   2.07     1  15000
diff.vs4[3]   0.40  0.52  -0.62   0.05   0.40   0.75   1.42     1   9700
diff.vs4[4]   0.00  0.00   0.00   0.00   0.00   0.00   0.00     1      1
```

```
# Fit model using maximum likelihood
(fm <- summary(lm(Cmean ~ as.factor(facFor)*elev-1-elev)))

Coefficients:
                         Estimate  Std. Error t value   Pr(>|t|)
as.factor(facFor)1         0.3353     0.2301   1.457  0.146328
as.factor(facFor)2         0.7596     0.2269   3.348  0.000935 ***
as.factor(facFor)3         1.6042     0.2052   7.816 1.37e-13 ***
as.factor(facFor)4         4.0558     0.2169  18.700   < 2e-16 ***
as.factor(facFor)1:elev  -0.6013     0.4203  -1.431  0.153772
as.factor(facFor)2:elev  -1.2880     0.4280  -3.009  0.002880 **
as.factor(facFor)3:elev  -1.8129     0.3433  -5.281 2.73e-07 ***
as.factor(facFor)4:elev  -2.2177     0.3862  -5.743 2.60e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.783 on 259 degrees of freedom
Multiple R-squared:  0.6689,   Adjusted R-squared:  0.6587
F-statistic: 65.42 on 8 and 259 DF,  p-value: < 2.2e-16
```

We will often see the linear model that underlies an ANOVA (analysis of variance) or an ANCOVA using nested indexing in the BUGS language. Let us plot the predicted response as a function of the explanatory variables `facFor` and `elev` (Figure 5.10(b)). We use the parameter estimates from the least-squares fit (= MLEs), but clearly could also use the Bayesian posterior means.

```
plot(elev[facFor==1], Cmean[facFor==1], col = "red", ylim = c(0, 20), xlab = "Elevation",
ylab = "",    frame.plot = F)
points(elev[facFor==2],  Cmean[facFor==2], col = "blue")
points(elev[facFor==3],  Cmean[facFor==3], col = "green")
points(elev[facFor==4],  Cmean[facFor==4], col = "black")
abline(fm$coef[1,1],  fm$coef[5,1], col = "red")
abline(fm$coef[2,1],  fm$coef[6,1], col = "blue")
abline(fm$coef[3,1],  fm$coef[7,1], col = "green")
abline(fm$coef[4,1],  fm$coef[8,1], col = "black")
```

To further illustrate how simple it is to test custom hypotheses in an MCMC-based analysis, let us compute the probability that the difference in the slopes between level 3 of `facFor` and the other levels of that factor is greater than 1. We plot the histograms of these contrasts (Figure 5.11) and then compute the proportion of the area under the curve that lies to the right of 1.

```
attach.bugs(out4)   # Allows to directly address the sims.list
str(diff.vs3)
par(mfrow = c(1, 3), mar = c(5,5,3,2), cex.lab = 1.5, cex.axis = 1.5)
hist(diff.vs3[,1], col = "grey", breaks = 100, main = "", freq=F, ylim = c(0, 0.8))
abline(v = 1, lwd = 3, col = "red")
hist(diff.vs3[,2], col = "grey", breaks = 100, main = "", freq=F, ylim = c(0, 0.8))
abline(v = 1, lwd = 3, col = "red")
hist(diff.vs3[,4], col = "grey", breaks = 100, main = "", freq=F, ylim = c(0, 0.8))
abline(v = 1, lwd = 3, col = "red")
```

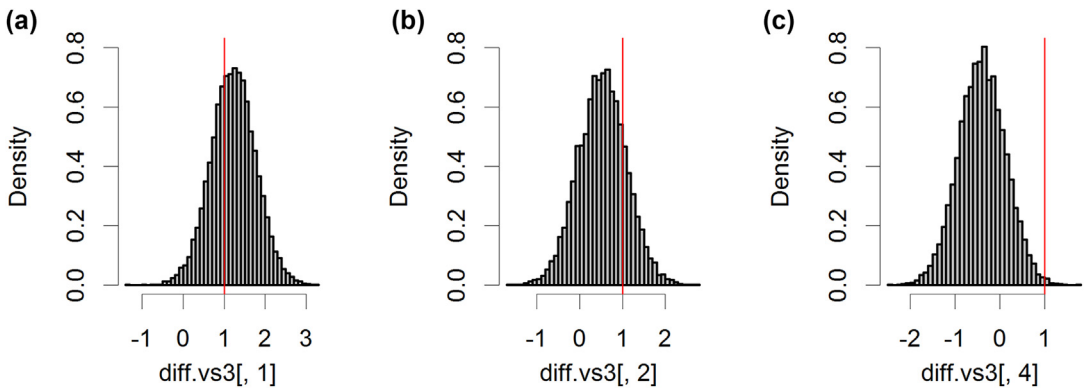

FIGURE 5.11

Posterior distributions of the difference between the slope of the regression of the mean tit counts on elevation in `facFor` level 3 versus levels 1 (a), 2 (b), and 4 (c). The probability that this difference is greater than 1 is represented by the area under the curve to the right of the red line for each posterior distribution.

```
# Prob. difference greater than 1
mean(diff.vs3[,1] > 1)
mean(diff.vs3[,2] > 1)
mean(diff.vs3[,4] > 1)
 [1] 0.6554667
 [1] 0.1981333
 [1] 0.003733333
```

Hence, there is a 66% probability that the difference between the slopes in groups 1 and 3 of `facFor` is greater than 1, and we find corresponding probabilities of 20% and of essentially 0% for the analogous slope differences between group 3 and groups 2 and 4, respectively.

5.7 PROPORTION OF VARIANCE EXPLAINED (R^2)

Continuing in the ANCOVA example, we show how R^2 can be computed: the proportion of variance explained by a given model relative to a null model with an intercept only. For the least-squares analysis from the `lm` fit of the model using the effects parameterization, we get $R^2 = 0.49$ (the value of 0.67 from the means parameterization is not appropriate, since there is no intercept in that model). We can obtain a similar value in a Bayesian model by fitting a null model with only an intercept and then expressing as an R^2 value the proportional reduction in the residual variance (not the standard deviation) achieved by fitting a more complex model. We first fit the null model with an intercept only.

```
cat(file = "Model0.txt","
model {

# Priors
mu ~ dnorm(0, 1.0E-06)
tau <- pow(sd, -2)
sd ~ dunif(0, 1000)
# Likelihood
for (i in 1:M){
  Cmean[i] ~ dnorm(mu, tau)
}
}
")
inits <- function() list(mu = rnorm(1))
params <- c("mu", "sd")
ni <- 6000  ;  nt <- 1  ;  nb <- 1000  ; nc <- 3
out0 <- jags(win.data, inits, params, "Model0.txt", n.chains = nc, n.thin = nt, n.iter = ni,
n.burnin = nb)
print(out0)
              mean    sd      2.5%       50%    97.5% overlap0 f  Rhat n.eff
mu           1.792 0.152    1.493     1.792    2.087     FALSE 1 1.000 15000
sd           2.485 0.108    2.282     2.481    2.705     FALSE 1 1.000 15000
deviance  1242.216 1.996 1240.258  1241.596 1247.474     FALSE 1 1.001  6256
```

We express the total unexplained variance around the mean in the null model as the residual variance. Then, we compute as R^2 the difference between the unexplained variance in the null model and that in the model with elevation and forest cover as a proportion of the total unexplained variance in the null model.

```
# Compute R2 from BUGS analysis
(total.var <- mean(out0$sims.list$sd^2))        # Total variance around the mean
[1] 6.185405
(unexplained.var <- mean(out3$sims.list$sd^2)) # Not explained by the ANCOVA
[1] 3.214631
(prop.explained <- (total.var - unexplained.var)/total.var)
[1] 0.4802878
```

Thus, a model with interaction effects of forest cover and elevation explains about half of the total variance in the mean counts of great tits. The idea of comparing the magnitude of a variance under a model with and without some covariate(s) is very general and can be used to express the explanatory power of covariates also in more complex models, such as HMs. For instance, in a survival analysis, we may express the effect of some climate variable on annual survival (phi) from a Cormack–Jolly–Seber model by the proportional reduction in the temporal variance of survival between a model that does and another that does not contain that covariate; i.e., compare model phi(random time) with model phi(covariate + random time)—see Grosbois et al. (2008) and Kéry and Schaub (2012, p. 189).

5.8 FITTING A MODEL WITH NONSTANDARD LIKELIHOOD USING THE ZEROS OR THE ONES TRICKS

Using the standard distribution functions in BUGS, you can fit a vast number of models. Moreover, by combining two or more of them, e.g., in an HM, you can extend the range of models considerably still. However, sometimes you may encounter a distribution that you cannot specify, or you may want to fit the integrated likelihood (see Chapter 2) of an HM directly. When you know how to write the likelihood of your model, you can fit it in BUGS using what is known as the "zeros trick" or the "ones trick" (Lunn et al., 2013, pp. 204–206).

For the zeros trick, imagine that you want to fit a model to a data set where observation i contributes a likelihood term L[i]. If we invent a dummy data set of all zeros and assume a Poisson(ϕ) distribution for it, then every observation has a contribution to the likelihood equal to $\exp(-\phi)$. If we then specify ϕ to be equal to the negative log-likelihood of observation i under our original model, we obtain the correct likelihood. In practice, we will have to add an arbitrary constant C to ensure nonnegativity of the Poisson mean. For the ones trick, we start with dummy data consisting solely of ones and specify a Bernoulli distribution with success parameter p_i, which is defined to be proportional to the desired likelihood term L[i] under the original model. Again, an arbitrary scaling constant C is usually required to ensure that $p_i \leq 1$. Here we illustrate both approaches for the trivial example of the normal response multiple linear regression from Section 5.3. We compare both solutions with the solution obtained using the standard distribution function for the normal.

Remember the likelihood of an individual normal observation (see Section 2.2),

$$L(\mu, \sigma^2|\mathbf{y}) = \sqrt{\frac{1}{2\pi\sigma^2}}\exp\left(-\frac{(y_i-\mu)^2}{2\sigma^2}\right),$$

and hence the negative log-likelihood is

$$NLL(\mu, \sigma^2|\mathbf{y}) = -\log\left(\sqrt{\frac{1}{2\pi\sigma^2}}\right) + \frac{(y_i-\mu)^2}{2\sigma^2}.$$

Below we will use both equations in the likelihood specification, the former in the ones trick and the latter in the zeros trick.

```
# Package the data needed in a bundle
win.data <- list(Cmean1 = Cmean, Cmean2 = Cmean, zeros = rep(0, M), ones = rep(1, M),
M = length(Cmean), elev = elev, forest = forest) # note 2 copies of response

# Write text file with model description in BUGS language
cat(file = "multiple_linear_regression_model.txt",
"model {

# Priors
for(k in 1:3){ # Loop over three ways to specify likelihood
 alpha0[k] ~ dnorm(0, 1.0E-06)    # Prior for intercept
 alpha1[k] ~ dnorm(0, 1.0E-06)    # Prior for slope of elev
 alpha2[k] ~ dnorm(0, 1.0E-06)    # Prior for slope of forest
 alpha3[k] ~ dnorm(0, 1.0E-06)    # Prior for slope of interaction
 sd[k] ~ dunif(0, 1000)           # Prior for dispersion on sd scale
}
var1 <- pow(sd[1], 2)             # Variance in zeros trick
var2 <- pow(sd[2], 2)             # Variance in ones trick
tau <- pow(sd[3], -2)             # Precision tau = 1/(sd^2)

C1 <- 10000 # zeros trick: make large enough to ensure lam >= 0
C2 <- 10000 # ones trick: make large enough to ensure p <= 1
pi <- 3.1415926

# Three variants of specification of the likelihood
for (i in 1:M){
# 'Zeros trick' for normal likelihood
  zeros[i] ~ dpois(phi[i]) # likelihood contribution is exp(-phi)
#  negLL[i] <- log(sd[1]) + 0.5 * pow((Cmean1[i] - mu1[i]) / sd[1],2 )
  negLL[i] <- -log(sqrt(1/(2*pi*var1))) + pow(Cmean1[i]-mu1[i],2)/(2*var1)
  phi[i] <- negLL[i] + C1
  mu1[i] <- alpha0[1] + alpha1[1]*elev[i] + alpha2[1]*forest[i] + alpha3[1]*elev[i]*
  forest[i]

# 'Ones trick' for normal likelihood
 ones[i] ~ dbern(p[i]) # likelihood contribution is p directly
 L[i] <- sqrt(1/(2*pi*var2)) * exp(-pow(Cmean1[i]-mu2[i],2)/(2*var2))
```

```
 p[i] <- L[i] / C2
 mu2[i] <- alpha0[2] + alpha1[2]*elev[i] + alpha2[2]*forest[i] + alpha3[2]*elev[i]*forest
[i]

# Standard distribution function for the normal
 Cmean2[i] ~ dnorm(mu3[i], tau)
 mu3[i] <- alpha0[3] + alpha1[3]*elev[i] + alpha2[3]*forest[i] + alpha3[3]*elev[i]*forest
[i]
}
}"
)

# Initial values
inits <- function() list(alpha0 = rnorm(3, 0, 10), alpha1 = rnorm(3,0,10), alpha2 =
rnorm(3,0,10), alpha3 = rnorm(3,0,10))

# Parameters monitored (i.e., for which estimates are saved)
params <- c("alpha0", "alpha1", "alpha2", "alpha3", "sd")

# MCMC settings
ni <- 1200  ;  nt <- 1  ;  nb <- 200  ; nc <- 3  # For JAGS
```

Much longer chains are required for BUGS to converge, so we fit the model in JAGS.

```
# Call JAGS
library(jagsUI)
outX <- jags(win.data, inits, params, "multiple_linear_regression_model.txt",
n.chains = nc, n.thin = nt, n.iter = ni, n.burnin = nb)
print(outX)
```

```
                mean      sd        2.5%         50%        97.5%  overlap0     f  Rhat  n.eff
alpha0[1]      1.662   0.115       1.441       1.663        1.884     FALSE 1.000 1.002   1417
alpha0[2]      1.660   0.114       1.433       1.660        1.874     FALSE 1.000 1.000   3000
alpha0[3]      1.657   0.115       1.436       1.660        1.878     FALSE 1.000 1.000   3000
alpha1[1]     -1.567   0.202      -1.957      -1.570       -1.176     FALSE 1.000 1.001   3000
alpha1[2]     -1.577   0.206      -1.980      -1.570       -1.187     FALSE 1.000 1.001   2180
alpha1[3]     -1.585   0.209      -1.989      -1.587       -1.170     FALSE 1.000 1.000   3000
alpha2[1]      2.346   0.191       1.966       2.347        2.719     FALSE 1.000 1.000   3000
alpha2[2]      2.349   0.202       1.963       2.345        2.753     FALSE 1.000 1.000   3000
alpha2[3]      2.346   0.199       1.957       2.342        2.720     FALSE 1.000 1.001   1435
alpha3[1]     -0.848   0.362      -1.574      -0.849       -0.153     FALSE 0.990 1.001   1226
alpha3[2]     -0.842   0.355      -1.524      -0.846       -0.128     FALSE 0.991 1.000   3000
alpha3[3]     -0.837   0.359      -1.548      -0.845       -0.119     FALSE 0.991 1.000   3000
sd[1]          1.861   0.083       1.706       1.858        2.033     FALSE 1.000 1.000   3000
sd[2]          1.861   0.083       1.702       1.858        2.032     FALSE 1.000 1.002   1333
sd[3]          1.860   0.084       1.704       1.856        2.030     FALSE 1.000 1.000   3000
deviance 5348179.889   5.641 5348171.008 5348179.076 5348192.794     FALSE 1.000 1.000   3000
```

Up to Monte Carlo error, we get identical answers for all three specifications of the normal likelihood. Finally, let us amuse ourselves by doing the analogous thing with ML—i.e., maximize the likelihood for the explicit description of the normal negative log-likelihood. We again find estimates that are numerically virtually identical to the posterior means.

```
# Define negative log-likelihood function
neglogLike <- function(param) {
 alpha0 = param[1]
 alpha1 = param[2]
 alpha2 = param[3]
 alpha3 = param[4]
 sigma = exp(param[5])  # Estimate sigma on log-scale
 mu = alpha0 + alpha1*elev + alpha2*forest + alpha3*elev*forest
#  -sum(dnorm(Cmean, mean=mu, sd=sigma, log=TRUE)) # cheap quick way
sum(-log(sqrt(1/(2*3.1415926*sigma^2))) + (Cmean-mu)^2/(2*sigma^2))
}

# Find parameter values that minimize function value
(fit <- optim(par = rep(0, 5), fn = neglogLike, method = "BFGS"))
 [1]   1.6603052 -1.5764599  2.3440036 -0.8506793  0.6073662

exp(fit$par[5])           # Backtransform to get sigma
[1] 1.83559
```

The zeros and ones tricks allow you to fit very general models, and you will sometimes see people fit HMs using these methods (e.g., Morales et al., 2004, for a hierarchical model for animal movement; Garrard et al. 2008; 2013 for time-to-detection occupancy models, and; Chelgren et al. 2011b for a distance sampling model).

5.9 POISSON GLM

We continue with a non-normal GLM and adopt a Poisson distribution with ANCOVA linear model for the counts of great tits. Since we have not one count per site, but three, we will follow a common approach and simply analyze the maximum count at each site, knowing that this must be the best non-model-based approximation, in the sense of being closest, to the true abundance of great tits at a site. We do *not* encourage this practice in general, but show this analysis here simply to illustrate a Poisson GLM. Let j index the four levels of `facFor`, and then we fit the following model:

$$Cmax_i \sim Poisson(\lambda_i)$$
$$\log(\lambda_i) = \alpha_{0,j} + \alpha_{1,j} * elev_i$$

where $Cmax_i$ is the maximum count for unit i and λ_i is the expected maximum count. We also compute Pearson residuals, $(Cmax_i - \lambda_i)/\sqrt{\lambda_i}$, which have the form of a raw residual divided by the standard deviation of unit i. To avoid numerical problems when the expected value becomes equal to zero, resulting in a division by zero in the Pearson residual, we add a small number e to the denominator. To emphasize the relatedness among different GLMs, we fit the same ANCOVA linear model as in the previous section, namely main and interaction effects of the forest factor (`facFor`) and elevation.

```
# Summarize data by taking max at each site
Cmax <- apply(C, 1, max)
table(Cmax)
Cmax
```

```
 0  1  2  3  4  5  6  7 8 9 10 11 12 13 14 15 16 17 22 30
77 55 30 12 17 23 11 10 6 6  2  2  2  3  4  1  2  1  2  1

# Bundle data
win.data <- list(Cmax = Cmax, M = length(Cmax), elev = elev, facFor = facFor, e = 0.0001)

# Specify model in BUGS language
cat(file = "Poisson_GLM.txt","
model {

# Priors
for(k in 1:4){
 alpha[k] ~ dnorm(0, 1.0E-06)   # Prior for intercepts
 beta[k] ~ dnorm(0, 1.0E-06)    # Prior for slopes
}

# Likelihood
for (i in 1:M){
 Cmax[i] ~ dpois(lambda[i])     # note no variance parameter
 log(lambda[i]) <- alpha[facFor[i]] + beta[facFor[i]] * elev[i]
 resi[i] <- (Cmax[i]-lambda[i]) / (sqrt(lambda[i])+e)  # Pearson residual
}
}
")
```

This nonnormal GLM is specified in the first two code lines inside the loop under the heading of the likelihood. The first defines the data distribution and the second line specifies a log link and the linear predictor, with the latter corresponding simply to an ANCOVA linear model as before.

```
# Initial values
inits <- function() list(alpha = rnorm(4,,3), beta = rnorm(4,,3))

# Parameters monitored
params <- c("alpha", "beta", "lambda", "resi")

# MCMC settings
ni <- 6000  ;  nt <- 1  ;  nb <- 1000  ; nc <- 3

# Call WinBUGS or JAGS from R and summarize posteriors
out5 <- bugs(win.data, inits, params, "Poisson_GLM.txt", n.chains = nc, n.thin = nt, n.iter
= ni, n.burnin = nb, debug = TRUE, bugs.directory = bugs.dir, working.directory = getwd())

out5J <- jags(win.data, inits, params, "Poisson_GLM.txt", n.chains = nc, n.thin = nt,
n.iter = ni, n.burnin = nb)
par(mfrow = c(4,2))  ;  traceplot(out5J, c("alpha[1:4]", "beta[1:4]"))
print(out5J, 3)
            mean    sd    2.5%     50%   97.5% overlap0     f  Rhat n.eff
alpha[1]  -1.028 0.248 -1.526  -1.019  -0.569    FALSE 1.000 1.030    57
alpha[2]  -0.170 0.151 -0.484  -0.166   0.115     TRUE 0.874 1.012 10409
alpha[3]   0.754 0.091  0.569   0.755   0.927    FALSE 1.000 1.001 12407
```

```
alpha[4]   1.909  0.049   1.810   1.910   2.002    FALSE  1.000  1.001  2094
beta[1]   -2.294  0.398  -3.068  -2.287  -1.529    FALSE  1.000  1.023    97
beta[2]   -1.926  0.245  -2.418  -1.922  -1.461    FALSE  1.000  1.011  2797
beta[3]   -1.315  0.139  -1.595  -1.313  -1.051    FALSE  1.000  1.002  1818
beta[4]   -0.715  0.082  -0.875  -0.716  -0.555    FALSE  1.000  1.001  5423
```

We produce three residual plots as we did for the normal GLM (Figure 5.12). They do not look quite perfect; there are more large than small residuals. Since we fit this model for illustration only, we ignore this moderate lack of fit here.

```
par(mfrow = c(1, 3), mar = c(5,5,3,2), cex = 1.3, cex.lab = 1.5, cex.axis = 1.5)
hist(out5$summary[276:542, 1], xlab = "Pearson residuals", col = "grey", breaks = 50,
main = "", freq = F, xlim = c(-5, 5), ylim = c(0, 0.57))
abline(v = 0, col = "red", lwd = 2)

plot(1:267, out5$summary[276:542, 1], main = "", xlab = "Order of data", ylab = "Pearson
residual", frame.plot = F)
abline(h = 0, col = "red", lwd = 2)

plot(out5$summary[9:275, 1],out5$summary[276:542, 1], main = "", xlab = "Predicted
values", ylab = "Pearson residual", frame.plot = F, xlim = c(-1, 14))
abline(h = 0, col = "red", lwd = 2)
```

Next, we fit the model using iterative reweighted least-squares (the resulting solutions are equivalent to MLEs for a GLM) and find the usual comforting numerical similarity with the Bayesian estimates. Note how we specify the same means parameterization of the linear model that we chose in the BUGS fit.

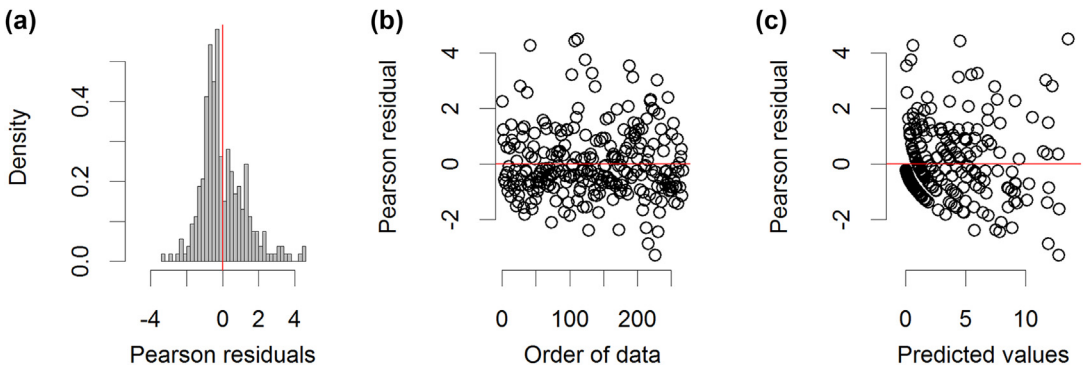

FIGURE 5.12

Three diagnostic plots for the Pearson residuals in a Poisson GLM: (a) histogram of the frequency distribution of the residuals, (b) residuals versus the order of the data, and (c) residuals versus predicted values. We could also plot the residuals against the covariates included in the analysis or against those left out of the model.

```
summary(glm(Cmax ~ factor(facFor)*elev-1-elev, family = poisson))

Coefficients:
                      Estimate  Std. Error  z value   Pr(>|z|)
factor(facFor)1        -0.99784    0.26366  -3.785  0.000154 ***
factor(facFor)2        -0.13993    0.15705  -0.891  0.372948
factor(facFor)3         0.76246    0.08999   8.472   < 2e-16 ***
factor(facFor)4         1.90985    0.04914  38.867   < 2e-16 ***
factor(facFor)1:elev   -2.27293    0.41834  -5.433  5.54e-08 ***
factor(facFor)2:elev   -1.90978    0.25114  -7.605  2.86e-14 ***
factor(facFor)3:elev   -1.31210    0.13730  -9.556   < 2e-16 ***
factor(facFor)4:elev   -0.69995    0.08343  -8.389   < 2e-16 ***
```

Derived quantities (= functions of parameters) may be computed inside of BUGS or outside in R, using results from a BUGS model fit. Sometimes it can be advantageous to do such calculations in R after fitting the model in BUGS. We illustrate this using the expected maximum count of great tits—i.e., `lambda`. The Poisson expectation is a deterministic function of the regression parameters `alpha[1:4]` and `beta[1:4]`, for which we have 15,000 posterior samples each. We can use these to get samples of the posterior distribution of `lambda` for each unit in the data set. We create a data structure wherein we store the posterior samples for the newly calculated `lambda`, let us call it `lambda2`. We do this by applying the linear regression with each pair of MCMC samples from the two parameters (`facFor` and `elev`) and backtransforming using the inverse of the log link function. As an aside, note how we can use nested indexing in R exactly as in BUGS.

```
lambda2 <- array(dim = c(15000, 267))
for(j in 1:267){                  # Loop over sites
  lambda2[,j] <- exp(out5$sims.list$alpha[,facFor[j]] +
out5$sims.list$beta[,facFor[j]] * elev[j]) # linear regression/backtransform
}
plot(out5$sims.list$lambda ~ lambda2, pch = ".")          # Check the two are identical
lm(c(out5$sims.list$lambda) ~ c(lambda2))
```

Finally, let us assume that we want to produce predictions of the expected maximum count versus elevation separately for each level of `facFor` (but we illustrate this for the first level only). We produce three plots (Figure 5.13) that illustrate different ways of graphing the point estimate of a prediction along with its uncertainty. Note that we have to sort pairs of elevation and predicted response according to the order of the values of elevation for easy plotting. In the third plot, we show the uncertainty around the prediction by plotting a random sample of 50 from their posterior predictive distribution.

```
sorted.ele1 <- sort(elev[facFor == 1])
sorted.y1 <- out5$summary[9:275,][facFor == 1,][order(elev[facFor == 1]),]

# Plot A
par(mfrow = c(1, 3), mar = c(5,5,3,2), cex.lab = 1.5, cex.axis = 1.5)
plot(elev[facFor == 1], jitter(Cmax[facFor ==1]), ylab = "Maximum count",
xlab = "Elevation (scaled)", frame.plot=F, ylim = c(0, 6))
```

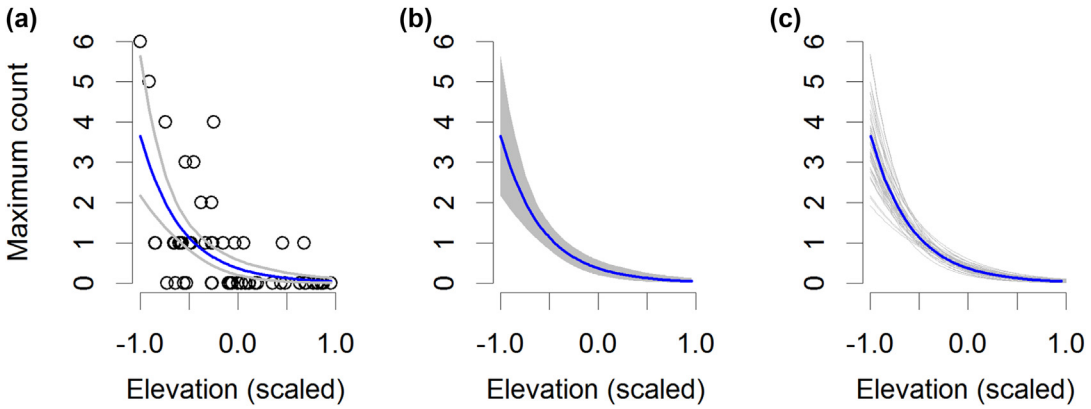

FIGURE 5.13

Three graphical summaries of the Poisson ANCOVA model fitted as the regression of the maximum count of great tits at each site on the scaled elevation and forest cover factor (shown for level 1 of `facFor` only; analogous plots could be produced for levels 2–4 of `facFor`). Posterior means of the predicted regression line of the mean tit count on scaled elevation are shown in blue, and the uncertainty around the prediction is depicted in grey ((a and b) 95% CRI; (c) a random sample of 50 posterior samples is shown). Note that the credible interval is for the expected count, not for observed counts; hence unsurprisingly, in the left plot most of the observed data lie outside of the interval (moreover, they are slightly jittered).

```
lines(sorted.ele1, sorted.y1[,1], col = "blue", lwd = 2) # Post. mean
lines(sorted.ele1, sorted.y1[,3], col = "grey", lwd = 2) # Lower 95% CL
lines(sorted.ele1, sorted.y1[,7], col = "grey", lwd = 2) # Upper 95% CL

# Plot B
plot(sorted.ele1, sorted.y1[,1], type='n', xlab = "Elevation (scaled)", ylab = "",
frame.plot = F, ylim = c(0, 6))
polygon(c(sorted.ele1, rev(sorted.ele1)), c(sorted.y1[,3], rev(sorted.y1[,7])),
col='grey', border=NA)
lines(sorted.ele1, sorted.y1[,1], col = "blue", lwd = 2)

# Plot C
elev.pred <- seq(-1,1, length.out = 200) # Cov. for which to predict lambda
n.pred <- 50                             # Number of prediction profiles
pred.matrix <- array(NA, dim = c(length(elev.pred), n.pred))
for(j in 1:n.pred){
  sel <- sample(1:length(out5$sims.list$alpha[,1]),1) # Choose one post. draw
  pred.matrix[,j] <- exp(out5$sims.list$alpha[sel,1] + out5$sims.list$beta[sel,1] *
elev.pred)
}
```

```
plot(sorted.ele1, sorted.y1[,1], type='n', xlab = "Elevation (scaled)", ylab = "",
frame.plot = F, ylim = c(0, 6))
matlines(elev.pred, pred.matrix, col = "grey", lty = 1, lwd = 1)
lines(sorted.ele1, sorted.y1[,1], col = "blue", lwd = 2)
```

5.10 GoF ASSESSMENT: POSTERIOR PREDICTIVE CHECKS AND THE PARAMETRIC BOOTSTRAP

We use the Poisson GLM to illustrate the assessment of the GoF of a model Bayesianly, using posterior predictive distributions (Rubin, 1984; Gelman et al., 1996). We also compare this approach with a related (purely) frequentist technique, the parametric bootstrap (Efron and Tibshirani, 1993; Dixon, 2006). We say "purely frequentist" because the Bayesian method has a decidedly frequentist flavor—it is based on hypothetical replicate data sets and hence could be called "semifrequentist" (or a "Bayesianly justifiable frequentist calculation," Rubin, 1984).

What is a posterior *predictive* distribution? This is simply the posterior distribution of anything that can be observed, here, of the data. We have seen earlier that using Bayesian analysis we can readily generate replicate data under our model when a response is missing. That is, using our model fit to the observed data and using the parameter values estimated at every iteration in the MCMC algorithm, we can create a "replicate" data set under the very same model with the same parameter values. As usual, we have an entire (posterior predictive) distribution for each replicate datum.

What does it mean to say "a model fits the data"? One reasonable answer is that the model is likely to produce data sets that are in some way "similar" to the data set we have at hand. Using MCMC, we can readily obtain the posterior predictive distribution of the data or of any function of the data. A posterior predictive check compares the observed data (or some function thereof) with their posterior predictive distribution (or some function thereof). And how do we measure "similarity," or conversely, dissimilarity? There is great flexibility in defining discrepancy measures (see, for instance, Gelman et al., 1996). For instance, we might choose as our discrepancy measure some statistic of the data that does not involve the estimated parameters, such as the mean, median, standard deviation, mean, maximum or range of the data. We might even use a visual impression of the data—i.e., graphs that are informative in some way about patterns in the data (e.g., pp. 154–155 in Gelman et al., 2014). However, the most common approach is to choose a test statistic that depends on the data and on the parameter estimates, and to compute an omnibus test that quantifies the "distance" between the observed data and the expected data under the model. Typically, such discrepancy measures are Chi-square, sums of squares, or the Freeman–Tukey statistic (see also Chapters 6 and 7).

We will see many applications of bootstrapping and posterior predictive distributions in later chapters and when using custom functions (such as `parboot` in `unmarked`), so it is good that we know how they work under the hood. Here, for the Poisson GLM from Section 5.9, we will compute a Chi-square discrepancy [i.e., the sum of (observed-expected)^2 / expected] as a posterior predictive check, and similarly in a parametric bootstrap. We will add to the expected values a very small number to avoid division by zero due to rounding errors. In addition, we will use the range of the observed values to check whether the spread of the data is sufficiently well represented by our model. We cannot

directly compute the range in WinBUGS, since the `min` and the `max` functions do not work for vectors like they do in R, but we can do so in OpenBUGS, and also in JAGS, when loading the module 'bugs.'

```
# Bundle data
win.data <- list(Cmax = Cmax, M = length(Cmax), elev = elev, facFor = facFor, e = 0.0001)

# Specify model in BUGS language
cat(file = "Poisson_GLM.txt","
model {

# Priors
for(k in 1:4){
 alpha[k] ~ dnorm(0, 1.0E-06)
 beta[k] ~ dnorm(0, 1.0E-06)
}

# Likelihood and computations for posterior predictive check
for (i in 1:M){
 Cmax[i] ~ dpois(lambda[i])
 log(lambda[i]) <- alpha[facFor[i]] + beta[facFor[i]] * elev[i]

# Fit assessments: Chi-square test statistic and posterior predictive check
 chi2[i] <- pow((Cmax[i]-lambda[i]),2) / (sqrt(lambda[i])+e)          # obs.
 Cmax.new[i] ~ dpois(lambda[i])    # Replicate (new) data set
 chi2.new[i] <- pow((Cmax.new[i]-lambda[i]),2) / (sqrt(lambda[i])+e) # exp.
}

# Add up discrepancy measures for entire data set
fit <- sum(chi2[])                 # Omnibus test statistic actual data
fit.new <- sum(chi2.new[])         # Omnibus test statistic replicate data

# range of data as a second discrepancy measure
obs.range <- max(Cmax[]) - min(Cmax[])
exp.range <- max(Cmax.new[]) - min(Cmax.new[])
}
")

# Initial values
inits <- function() list(alpha = rnorm(4,,3), beta = rnorm(4,,3))

# Parameters monitored
params <- c("chi2", "fit", "fit.new", "obs.range", "exp.range")
params <- c("Cmax.new", "chi2.new", "chi2", "fit", "fit.new", "obs.range", "exp.range")

# MCMC settings
ni <- 6000  ;  nt <- 1  ;  nb <- 1000  ; nc <- 3
```

We fit the model in JAGS and load the module 'bugs.'

```
# Call JAGS from R and summarize posteriors
out5.1 <- jags(win.data, inits, params, "Poisson_GLM.txt", n.chains = nc,
n.thin = nt, n.iter = ni, n.burnin = nb, modules = 'bugs')
```

```
print(out5.1, 2)
JAGS output for model 'Poisson_GLM.txt', generated by jagsUI.
Estimates based on 3 chains of 6000 iterations,
burn-in = 1000 iterations and thin rate = 1,
yielding 15000 total samples from the joint posterior.
MCMC ran for 0.235 minutes at time 2014-11-14 20:33:41.

              mean    sd    2.5%     50%   97.5%  overlap0 f Rhat n.eff
chi2[1]       9.80  1.43    7.12    9.74   12.75     FALSE 1    1  7655
chi2[2]       3.29  0.85    1.77    3.23    5.08     FALSE 1    1  5262
chi2[3]       1.74  0.69    0.58    1.68    3.28     FALSE 1    1  4379
[ ... ]
chi2[265]     0.24  0.15    0.03    0.22    0.59     FALSE 1    1  1871
chi2[266]     1.47  0.27    0.99    1.45    2.05     FALSE 1    1  4567
chi2[267]     0.02  0.01    0.00    0.01    0.05     FALSE 1    1  2478
fit         842.40 17.32  815.61  840.14  882.95     FALSE 1    1 15000
fit.new     416.91 45.66  332.45  414.98  512.72     FALSE 1    1 15000
obs.range    30.00  0.00   30.00   30.00   30.00     FALSE 1  NaN     1
exp.range    18.60  2.45   15.00   18.00   24.00     FALSE 1    1 15000
deviance   1004.05  4.05  998.12 1003.37 1013.55     FALSE 1    1 15000
```

First of all, we see that the Chi-square test statistic computed for data sets simulated under the assumptions of our model, and using the posterior distributions of the parameters estimated from our model (`fit.new`), is substantially smaller than that for our actual data set (`fit`). We can express the degree of lack of fit as the ratio of `fit/fit.new`, which is about 2. We can summarize this analysis by plotting the posterior distributions of the two fit statistics against each other (Figure 5.14(a)).

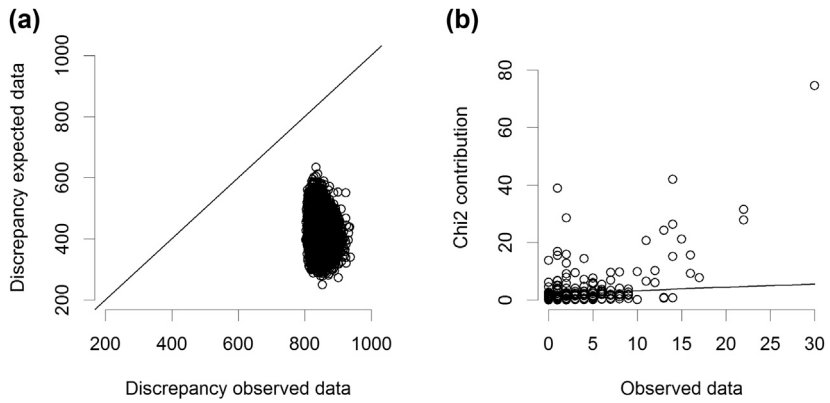

FIGURE 5.14

(a) Posterior predictive check for an omnibus fit statistic (Chi-square) for the Poisson model for maximum tit counts, showing the expected values of the statistic (computed from the replicate data sets) versus observed values (for the observed data set). (b) Contributions to the overall fit statistic of each individual datum, and expected values of the Chi-square contributions (this is the line).

```
par(mfrow = c(1, 2), mar = c(5,5,3,2), cex.lab = 1.5, cex.axis = 1.5)
plot(out5.1$sims.list$fit, out5.1$sims.list$fit.new, xlim = c(200, 1000), ylim =
c(200, 1000), main = "", xlab = "Discrepancy observed data", ylab = "Discrepancy
expected data", frame.plot = F, cex = 1.5)
abline(0,1, lwd = 2)
```

We can compute a Bayesian *p*-value, which is the probability to obtain, under the null hypothesis of a correctly specified model, a test statistic that is at least as extreme as the observed test statistic computed from the actual data. This is represented by the proportion of points above the diagonal equality line and here it is equal to zero.

```
(bpv <- mean(out5.1$sims.list$fit.new > out5.1$sims.list$fit))
[1] 0
```

A model that fits the data will have a Bayesian *p*-value around 0.5, and hence our *p*-value of zero suggests the model does not fit. The null distribution of Bayesian *p*-values is not uniform, so, we cannot say that a Bayesian *p*-value outside of the interval of, say, (0.025, 0.975) represents a significant lack of fit. Rather, we have to judge fit more qualitatively.

If this were a "real" analysis, then such a clear indication of lack of fit might be devastating to some. However, it would not be sensible to throw away the model and claim that the data set cannot be analyzed. First of all, we do not know whether the chosen fit statistic is really relevant for the objective of our modeling. And second, much more important than a black-and-white answer to the question "Does the model fit?" is an answer to "*Where* does the model fit and *where* does it not?" Hence, we are big fans of residual checks, where we can inspect patterns in the lack of fit. For instance, with count data it might be that the lack of fit is simple overdispersion—i.e., residuals that are too extreme and have no systematic pattern. We could then accommodate such overdispersion by adding a normal random effect in the linear predictor, thereby converting the model into a Poisson-lognormal model (see chapter 4 in Kéry and Schaub, 2012). We will not do so in this illustration of the technique, but want to point out again that model building is best seen as an iterative process in which we build a model, check whether it describes the salient features of our data well, fit an improved model and so forth.

Next, we illustrate such model diagnostics by looking at patterns in the point-wise contributions to the overall Chi-square GoF statistic `fit`. We plot these contributions versus the observed values (Figure 5.14(b)). For a fitting model, we would expect these contributions to be proportional to the square root of the data.

```
plot(Cmax, out5.1$mean$chi2, xlab = "Observed data", ylab = "Chi2 contribution",
frame.plot = F, ylim = c(0, 70), cex = 1.5)
lines(0:30, sqrt(0:30), lwd = 2)
```

There does not seem to be a clear pattern, but before we accept simple overdispersion in a "real" analysis, rather than a structural problem of the model, we should probably look a little harder for any patterns in the lack of fit.

We show this analysis to illustrate the mechanics of a posterior predictive check and how it can be summarized in a Bayesian *p*-value, but will not bother to track down a fitting model here. Rather, we

next illustrate a parametric bootstrap to conclude this brief practical demonstration of two important methods for gauging GoF. In a parametric bootstrap, we repeat the following a large number of times (e.g., 1000–5000 times):

1. use the MLEs of the parameters from our data set to generate a replicate data set
2. refit the model to that new data set
3. compute a GoF discrepancy measure, such as Chi-square, using the simulated, new data set and associated MLEs

This yields the empirical distribution of the GoF statistic under the hypothesis of a fitting model. Exactly as for the posterior predictive check, we know that it does so because we generated this distribution for "perfect" data sets. They are perfect in the sense that they were generated under the exact assumptions of the model and using the point estimates from our data set. This is exactly like the replicated data sets in the posterior predictive checks earlier, except that the parametric bootstrap does not incorporate the parameter estimation uncertainty, while the Bayesian posterior predictive distributions do. If the test statistic obtained from our actual data set is not atypical in this reference distribution, then we conclude that the model fits, while if the test statistic is extreme (typically larger), then we conclude that the model does not fit. Let us do this now. Again, as for a posterior predictive check, we can be creative in our choice of test statistic. Here, we use Chi-square and the range of data as before.

```
# Fit model to actual data and compute two fit statistics
fm <- glm(Cmax ~ as.factor(facFor)*elev, family = poisson) # Fit model
observed <- Cmax
expected <- predict(fm, type = "response")
plot(observed, expected)
abline(0,1)
chi2.obs <- sum((observed - expected)^2 / (expected + 0.0001)) # fit stat 1
range.obs <- diff(range(observed))                             # fit stat 2

# Generate reference distribution of fit stat under a fitting model
simrep <- 100000                                # Might want to try 1000 first
chi2vec <- rangevec <- numeric(simrep)          # vectors for Chi2 and maximum
for(i in 1:simrep){
 cat(paste(i, "\n"))
 Cmaxrep <- rpois(n = 267, lambda = expected)   # Generate replicate data set
 fmrep <- glm(Cmaxrep ~ as.factor(facFor)*elev, family = poisson) # Refit model
 expectednew <- predict(fmrep, type = "response")
 chi2vec[i] <- sum((Cmaxrep - expectednew)^2 / (expectednew + 0.0001))
 rangevec[i] <- diff(range(Cmaxrep))
}
```

Hence, the parametric bootstrap does something quite analogous to a posterior predictive check. It is instructive to look at the computed quantities side by side (Figure 5.15).

```
# Summarize bootstrap results and compare with posterior predictive dist.
par(mfrow = c(2, 2), mar = c(5,5,3,2), cex.lab = 1.5, cex.axis = 1.5)
hist(out5.1$sims.list$fit.new, col = "grey", main = "", breaks = 100, xlim = c(180, 900),
freq = F, ylim = c(0, 0.01))
```

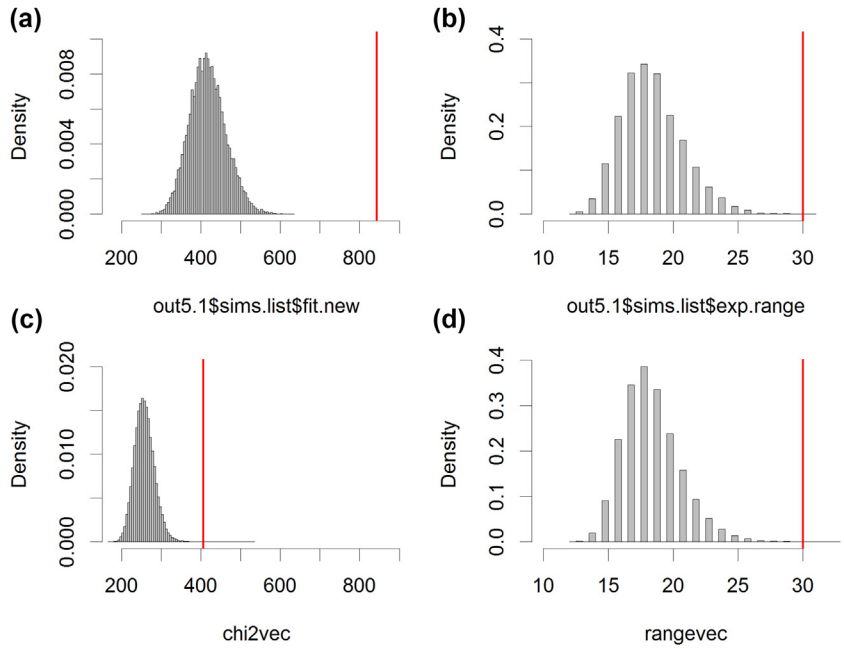

FIGURE 5.15

Posterior predictive checks (top) and parametric bootstrap (bottom) for a Chi-square test statistic (left) and for the range of the observed data (right). Histograms show the frequency distribution for replicate data sets simulated under the model. Red lines show the posterior means of the Chi-square test statistic for the actual data (a and c) or for the range of observed data (b and d).

```
abline(v = mean(out5.1$sims.list$fit), col = "red", lwd = 2)
hist(out5.1$sims.list$exp.range, col = "grey", main = "", breaks = 50, xlim = c(10, 32),
freq = F, ylim = c(0, 0.40))
abline(v = mean(out5.1$sims.list$obs.range), col = "red", lwd = 2)
hist(chi2vec, col = "grey", main = "", breaks = 100, xlim = c(180, 900), freq = F,
ylim = c(0, 0.02))
abline(v = chi2.obs, col = "red", lwd = 2)
hist(rangevec, col = "grey", main = "", breaks = 50, xlim = c(10, 32), freq = F)
abline(v = range.obs, col = "red", lwd = 2)
```

We see that the posterior predictive checks and the parametric bootstrap yield qualitatively very similar answers. However, the statistics are larger in magnitude for the former than for the latter, perhaps because the parametric bootstrap ignores some of the parameter estimation uncertainty, while in the posterior predictive check, the full parameter estimation uncertainty is propagated into the discrepancy measure. We can quantify the magnitude of the lack of fit by the ratio of the fit statistic for the observed data over that for the expected data.

```
# Lack of fit ratio in PPD and parboot
mean(out5.1$sims.list$fit/out5.1$sims.list$fit.new) # ppc
mean(chi2.obs/chi2vec)                              # parboot
[1] 2.045696
[1] 1.58963
```

For the parametric bootstrap, we can compute the *p*-value as 1 minus the percentile of the observed value in the reference distribution characterized by `chi2vec` and `rangevec`. This is the probability to obtain a more extreme value under the null hypothesis of a fitting model than the value of the test statistic for the observed data. This corresponds to the area under the curve to the right of the red line in Figure 5.15(d) and (e).

```
(pval1 <- 1-rank(c(chi2vec, chi2.obs))[simrep+1]/(simrep+1))
(pval2 <- 1-rank(c(rangevec, range.obs))[simrep+1]/(simrep+1))
[1] 0.0002199978
[1] 0.0001549985
```

Both posterior predictive checks and the parametric bootstrap are extremely flexible and powerful to test the fit of a model to your data set and should be more widely used, even though we do not know in general the power of the tests and, in the case of Bayesian *p*-value, we do not know their calibration either. These tests are perhaps best used in a qualitative way and, especially if an omnibus test suggests lack of fit, are best followed by more detailed residual diagnostics.

5.11 BINOMIAL GLM (LOGISTIC REGRESSION)

We next use BUGS to fit a logistic regression, also known as a binomial GLM. For this, we first quantize the great tit counts from one survey to obtain zeros and ones, indicating whether a count is zero or greater than zero. At several places in this book we emphasize that this is exactly how "presence/absence" or "detection/nondetection" data often arise in practice, namely as a simple summary of an abundance distribution (or here, of a distribution of counts). Once more, we fit the model using Bayesian and ML methods to emphasize their numerical similarity. We actually fit a Bernoulli GLM (i.e., a binomial GLM with trial size 1), but binomial GLMs with trial sizes >1 occur throughout the book, for instance in Chapter 6. We fit the following model to `y[,1]`, the detection/nondetection data for the first survey:

$$y_{i,1} \sim Bernoulli(\theta_i)$$

$$\text{logit}(\theta_i) = \alpha_{0,j} + \alpha_{1,j} * elev_i$$

As before, j indexes the four levels of `facFor`.

```
# Quantize counts from first survey and describe
y1 <- as.numeric(C[,1] > 0) # Gets 1 if first count greater than zero
table(y1)
```

```
y1
  0   1
137 130

mean(N > 0)      # True occupancy
mean(y1)         # Observed occupancy after first survey
```

Hence, true occupancy (proportion of occupied sites) is 77%, while the observed occupancy after the first survey is only 49%. We fit the ANCOVA linear model to the binary detection/nondetection response by specifying a Bernoulli GLM.

```
# Bundle data
win.data <- list(y1 = y1, M = length(y1), elev = elev, facFor = facFor)

# Specify model in BUGS language
cat(file = "Bernoulli_GLM.txt","
model {

# Priors
for(k in 1:4){
 alpha[k] <- logit(mean.psi[k])   # intercepts
 mean.psi[k] ~ dunif(0,1)
 beta[k] ~ dnorm(0, 1.0E-06)      # slopes
}

# Likelihood
for (i in 1:M){
 y1[i] ~ dbern(theta[i])
 logit(theta[i]) <- alpha[facFor[i]] + beta[facFor[i]] * elev[i]
}
}
")

# Initial values
inits <- function() list(mean.psi = runif(4), beta = rnorm(4,,3))  # Priors 2

# Parameters monitored
params <- c("mean.psi", "alpha", "beta", "theta")

# MCMC settings
ni <- 6000  ;  nt <- 1  ;  nb <- 1000  ; nc <- 3

# Call WinBUGS or JAGS from R (ART <1 min)
out6 <- bugs(win.data, inits, params, "Bernoulli_GLM.txt", n.chains = nc,
n.thin = nt, n.iter = ni, n.burnin = nb, debug = TRUE, bugs.directory = bugs.dir,
working.directory = getwd())

out6J <- jags(win.data, inits, params, "Bernoulli_GLM.txt", n.chains = nc, n.thin = nt,
n.iter = ni, n.burnin = nb)
par(mfrow = c(4,2))  ;  traceplot(out6J, c("alpha[1:4]", "beta[1:4]"))
```

We can summarize the posterior distributions for the logit-linear regression parameters, though we can no longer directly compare them with the truth, since the data were generated under a different model than the one fitted.

```
print(out6, 2)
              mean    sd   2.5%     25%    50%    75%  97.5%  Rhat  n.eff
mean.psi[1]   0.17  0.06   0.08    0.13   0.17   0.21   0.29     1   3900
mean.psi[2]   0.25  0.06   0.14    0.21   0.25   0.30   0.39     1   4000
mean.psi[3]   0.54  0.06   0.43    0.50   0.54   0.58   0.66     1  15000
mean.psi[4]   0.82  0.05   0.73    0.80   0.83   0.86   0.91     1  15000
alpha[1]     -1.62  0.41  -2.50   -1.87  -1.59  -1.33  -0.88     1   4100
alpha[2]     -1.11  0.35  -1.82   -1.33  -1.10  -0.87  -0.47     1   4000
alpha[3]      0.17  0.24  -0.29    0.01   0.17   0.33   0.65     1  15000
alpha[4]      1.58  0.33   0.99    1.36   1.57   1.79   2.26     1  15000
beta[1]      -1.98  0.78  -3.62   -2.47  -1.93  -1.43  -0.57     1   2700
beta[2]      -2.56  0.74  -4.12   -3.05  -2.53  -2.05  -1.21     1   8500
beta[3]      -0.95  0.42  -1.77   -1.22  -0.94  -0.66  -0.13     1  11000
beta[4]       0.49  0.59  -0.63    0.10   0.49   0.88   1.68     1  15000
theta[1]      0.64  0.07   0.50    0.59   0.64   0.68   0.76     1  15000
theta[2]      0.66  0.07   0.52    0.62   0.67   0.71   0.79     1  15000
theta[3]      0.85  0.06   0.73    0.81   0.85   0.89   0.94     1  15000
[ ... ]
```

```
# Compare with MLEs obtained by iteratively-reweighted least-squares method
summary(glm(y1 ~ factor(facFor)*elev-1-elev, family = binomial))

[ .... ]
Coefficients:
                     Estimate Std. Error z value   Pr(>|z|)
factor(facFor)1       -1.6194     0.4172  -3.881   0.000104 ***
factor(facFor)2       -1.1122     0.3481  -3.195   0.001398 **
factor(facFor)3        0.1774     0.2392   0.742   0.458303
factor(facFor)4        1.5873     0.3307   4.801   1.58e-06 ***
factor(facFor)1:elev  -1.9277     0.7860  -2.453   0.014180 *
factor(facFor)2:elev  -2.4480     0.7260  -3.372   0.000747 ***
factor(facFor)3:elev  -0.9154     0.4125  -2.219   0.026477 *
factor(facFor)4:elev   0.4924     0.5821   0.846   0.397600
[ .... ]
```

We overlay the estimated regression equations onto plots of the observed data (jittered for enhanced readability); see Figure 5.16. Note the slight contortions to get the ordering of the x and y variables right.

```
# Plot of observed response vs. two covariates
par(mfrow = c(1, 2), mar = c(5,5,3,2), cex.lab = 1.5, cex.axis = 1.5)
F1 <- facFor == 1 ; F2 <- facFor == 2 ; F3 <- facFor == 3 ; F4 <- facFor == 4
plot(jitter(y1,,0.05) ~ facFor, xlab = "Forest factor", ylab = "Observed occupancy
probability", frame.plot = F, ylim = c(0, 1.15))
```

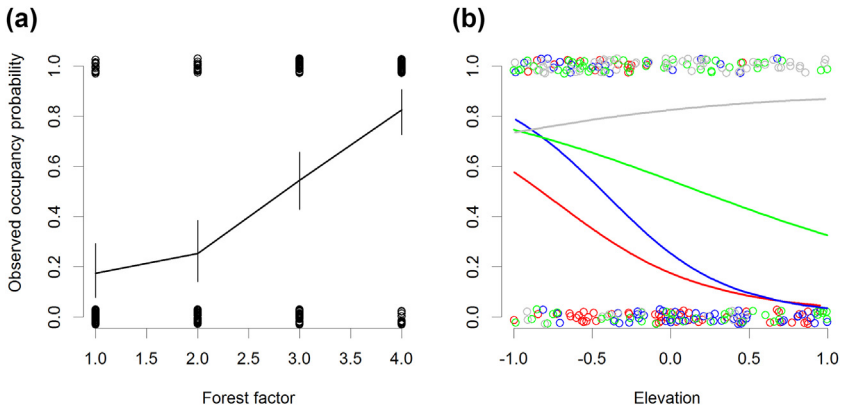

FIGURE 5.16

Predictions of the observed occupancy probability (theta) of simulated great tit data under the Bernoulli model; circles are observed data (jittered). (a) Observed detection/nondetection data versus `facFor`; line shows estimates of theta (with 95% CRI) at average scaled elevation. (b) Observed and estimated relationship of theta with elevation for all four levels of `facFor` (1—red, 2—blue, 3—green, 4—grey).

```
lines(1:4, out6$summary[1:4,1], lwd = 2)
segments(1:4, out6$summary[1:4,3], 1:4, out6$summary[1:4,7])

plot(elev[F1], jitter(y1,,0.1)[F1], xlab = "Elevation", ylab = "", col = "red",
frame.plot = F)
points(elev[F2], jitter(y1,,0.05)[F2], col = "blue")
points(elev[F3], jitter(y1,,0.05)[F3], col = "green")
points(elev[F4], jitter(y1,,0.05)[F4], col = "grey")
lines(sort(elev[F1]), out6$mean$theta[F1][order(elev[F1])], col="red", lwd=2)
lines(sort(elev[F2]), out6$mean$theta[F2][order(elev[F2])], col="blue", lwd=2)
lines(sort(elev[F3]), out6$mean$theta[F3][order(elev[F3])], col="green", lwd=2)
lines(sort(elev[F4]), out6$mean$theta[F4][order(elev[F4])], col="grey", lwd=2)
```

5.12 MOMENT-MATCHING IN A BINOMIAL GLM TO ACCOMMODATE UNDERDISPERSION

Assume for the sake of illustration that we want to model our observed counts with a distribution *less* variable than a Poisson. A good example in avian population ecology is provided by counts of chicks in the nest. One solution would be to directly use a variant of a Poisson that allows underdispersion (Ridout and Besbeas, 2004; Wu et al., 2013; Lynch et al., 2014). Another solution is to adopt a binomial instead and apply a very useful technique called *moment matching* (see Section 3.3.3 in this book and Section 3.4.4 in Hobbs and Hooten, 2015). In our example data, the maximum observed count is 30, so let us assume that for some reason we *knew* that values beyond 32 are essentially impossible. In this situation, a Poisson might not be adequate, because it might easily predict responses greater than 32. If we adopt a binomial with trial size 32 instead, no count greater than 32 is possible.

However, we then face a problem when we want to model variation in the mean response, since there is no parameter in the binomial that corresponds to the mean. Rather, the mean is a function of parameter p and trial size N: Np. Interestingly, we can simply apply a linear model to a link transformation of that mean! Since we want to avoid negative predicted counts in a binomial, we would typically adopt a log link, exactly as in the Poisson GLM. We illustrate moment matching by modeling the expected count by site elevation and want the response to be bounded by 0 and 32.

For count $y_{i,j}$ at site i during survey j we fit the following model:

$$y_{i,j} \sim Binomial(p_i, N).$$

Here, N is *not* population size but the upper limit we wish to impose on the counts, while p_i is the expected count at site *i as a proportion of that limit*. To model structure in the mean of that "squeezed" (between 0 and N) response, note first that the binomial mean is:

$$\mu_i = Np_i$$

We can then apply a linear model to a suitable transformation of the expected response as usual.

$$\log(\mu_i) = \alpha + \beta * elev_i$$

We rearrange this to express p_i a function of N and the parameters of the linear model:

$$p_i = \frac{1}{N}\exp(\alpha + \beta * elev_i)$$

Note that this formulation does not constrain p to be $<= 1$, because whenever the exponential becomes greater than N (here, 32) inadmissible values for p will result. If this is a problem with BUGS, we can likely diagnosed it by a crash. A simple workaround will then be to simply scale the covariates more, e.g., divide them by 10, 100 or 1000, as needed. In the BUGS code, to obtain predictions of the response for selected values of elevation, we also add a line that computes the expected binomial response Np_i under the model.

```
# Bundle data
win.data <- list(y = data$C, M = nrow(data$C), J = ncol(data$C), elev = elev, N = 32)

# Specify model in BUGS language
cat(file = "squeezed_count_GLM.txt","
model {

# Priors
alpha ~ dnorm(0, 1.0E-06)
beta ~ dnorm(0, 1.0E-06)

# Likelihood
for (i in 1:M){
  p[i] <- 1/N * exp(alpha + beta * elev[i])  # linear model for expected response
(relative to bound N)
  mu[i] <- N * p[i]  # Expected response = binomial mean = first moment
  for(j in 1:J){
    y[i,j] ~ dbin(p[i], N)
  }
}
}
")
```

```
# Initial values
inits <- function() list(alpha = runif(1), beta = rnorm(1))
# Parameters monitored
params <- c("alpha", "beta", "mu")

# MCMC settings
ni <- 300  ;  nt <- 1  ;  nb <- 100  ;  nc <- 3

# Call JAGS from R (ART <1 min)
library(jagsUI)
out7 <- jags(win.data, inits, params, "squeezed_count_GLM.txt", n.chains = nc,
n.thin = nt, n.iter = ni, n.burnin = nb)
par(mfrow = c(4,2))  ;  traceplot(out7)
print(out7, 3)
```

As emphasized by Hobbs and Hooten (2015), moment matching is a very useful technique that gives you greatly flexibility in your modeling. For another example, see Section 7.6, where we use moment matching to reparameterize the BUGS implementation of the negative binomial distribution such that we have a mean and a variance parameter and can apply a linear model to the former. Another useful example Hobbs and Hooten (2015) give is for a beta distribution, with parameters α and β. The beta can be used to model ratios of numbers that are *not* counts but continuous measurements, say, to model coverage data in plant surveys. The mean (μ) of a beta distribution is $\alpha/(\alpha + \beta)$; hence, $\alpha = \mu\beta/(1 + \mu)$, and we can specify linear models for μ with an appropriate link function such as the logit. See also Cribari-Neto and Zeileis (2010) for an introduction to beta regression.

5.13 RANDOM-EFFECTS POISSON GLM (POISSON GLMM)

In the concluding two examples of the BUGS language, we illustrate the specification of random effects. Random effects are a defining feature of HMs, and the BUGS language makes particularly transparent what they are: realizations from an unobserved or partially observed random variable. In other words, random effects are parameters or latent variables that are given a distribution with (hyper) parameters that are estimated from the data.

We illustrate in the context of an HM that is somewhat related to the N-mixture model (see Chapter 6): a Poisson regression, or GLM, with random site effects fitted to all three simulated counts of great tits per site (Dennis et al., 2015a, also unpublished manuscript by W. A. Link). We include a random site effect, perhaps to avoid pseudoreplication and to account for the correlation of replicated counts made at the same site. In contrast to the *N*-mixture model, which is a binomial-Poisson mixture, this model is a Poisson–normal mixture model. Because we no longer aggregate the repeated measures, we can now directly fit both site covariates (elevation and forest cover) and sampling covariates (wind speed). We fit the following GLMM to the counts C_{ij} of great tits at site i during replicate survey j:

$$C_{ij} \sim Poisson(\lambda_{ij})$$

$$\log(\lambda_{ij}) = \alpha_{0,i} + \alpha_1 * elev_i + \alpha_2 * forest_i + \alpha_3 * elev_i * forest_i + \alpha_4 * wind_{ij}$$

$$\alpha_{0,i} \sim Normal\left(\mu_\alpha, \sigma_\alpha^2\right)$$

It is the last line that defines the regression intercepts as random effects: the `alpha0` parameters are defined to be draws from a normal distribution with mean and variance as hyperparameters that are estimated.

We write the code in a slightly less verbose format now. We will also compute, as a derived quantity in the BUGS code, a zero-centered version of the random site effects `alpha0`—i.e., subtract the value of their hypermean—for direct comparison with the frequentist analyses below. Note that much longer chains are required to achieve convergence in random-effects models.

```
# Bundle data
win.data <- list(C = C, M = nrow(C), J = ncol(C), elev = elev, forest = forest, elev.forest =
elev * forest, wind = wind)

# Specify model in BUGS language
cat(file = "RE.Poisson.txt","
model {

# Priors
mu.alpha ~ dnorm(0, 0.001)      # Mean hyperparam
tau.alpha <- pow(sd.alpha, -2)
sd.alpha ~ dunif(0, 10)         # sd hyperparam
for(k in 1:4){
 alpha[k] ~ dunif(-10, 10)      # Regression params
}

# Likelihood
for (i in 1:M){
  alpha0[i] ~ dnorm(mu.alpha, tau.alpha)  # Random effects and hyperparams
  re0[i] <- alpha0[i] - mu.alpha          # zero-centered random effects
  for(j in 1:J){
    C[i,j] ~ dpois(lambda[i,j])
    log(lambda[i,j]) <- alpha0[i] + alpha[1] * elev[i] + alpha[2] * forest[i] +
alpha[3] * elev.forest[i] + alpha[4] * wind[i,j]
  }
}
}")

# Other model run preparations
inits <- function() list(alpha0 = rnorm(M), alpha = rnorm(4))    # Inits
params <- c("mu.alpha", "sd.alpha", "alpha0", "alpha", "re0")   # Params
ni <- 30000 ; nt <- 25 ; nb <- 5000 ; nc <- 3                   # MCMC settings

# Call WinBUGS or JAGS from R (ART 6-7 min) and summarize posteriors
out8 <- bugs(win.data, inits, params, "RE.Poisson.txt", n.chains = nc,
n.thin = nt, n.iter = ni, n.burnin = nb, debug = TRUE, bugs.directory = bugs.dir,
working.directory = getwd())

out8 <- jags(win.data, inits, params, "RE.Poisson.txt", n.chains = nc, n.thin = nt,
n.iter = ni, n.burnin = nb)
par(mfrow = c(3,2)) ; traceplot(out8, c("mu.alpha", "sd.alpha", "alpha[1:3]"))
```

```
print(out8, 3)
              mean    sd    2.5%     50%   97.5% overlap0     f  Rhat  n.eff
mu.alpha    -0.839 0.072  -0.980  -0.840  -0.699    FALSE 1.000 1.001   2756
sd.alpha     0.267 0.049   0.169   0.268   0.362    FALSE 1.000 1.003   3000
alpha0[1]   -0.856 0.218  -1.306  -0.847  -0.444    FALSE 1.000 1.001    796
alpha0[2]   -0.910 0.208  -1.330  -0.903  -0.522    FALSE 1.000 1.004    650
alpha0[3]   -1.216 0.215  -1.657  -1.210  -0.809    FALSE 1.000 1.002   3000
[ ... ]
alpha0[265] -0.845 0.269  -1.409  -0.839  -0.340    FALSE 0.999 1.003    950
alpha0[266] -0.909 0.273  -1.466  -0.894  -0.387    FALSE 0.999 1.002   1298
alpha0[267] -0.836 0.276  -1.386  -0.831  -0.301    FALSE 0.999 1.001   3000
alpha[1]    -1.694 0.108  -1.900  -1.695  -1.486    FALSE 1.000 1.002   2944
alpha[2]     2.128 0.096   1.943   2.130   2.315    FALSE 1.000 1.003   2938
alpha[3]     1.390 0.161   1.076   1.391   1.702    FALSE 1.000 1.000   2721
alpha[4]    -1.711 0.063  -1.837  -1.711  -1.590    FALSE 1.000 1.001   3000
re0[1]      -0.017 0.206  -0.437  -0.014   0.386     TRUE 0.529 1.001    997
re0[2]      -0.071 0.195  -0.475  -0.064   0.298     TRUE 0.637 1.002    821
re0[3]      -0.377 0.195  -0.783  -0.368  -0.013    FALSE 0.981 1.001   3000
[ ... ]
re0[265]    -0.006 0.262  -0.534   0.003   0.491     TRUE 0.495 1.003   1243
re0[266]    -0.069 0.262  -0.598  -0.061   0.445     TRUE 0.596 1.002   1776
re0[267]     0.003 0.266  -0.544   0.003   0.524     TRUE 0.506 1.001   3000
```

If your computer runs out of memory, try running the model without the zero-centered random effects in the params list, and compute the posterior samples for `re0` in R. You might also want to thin more to save fewer samples per parameter.

At last, we fit this model non-Bayesianly using ML with package `lme4` (Bates et al. 2014). Function `glmer` requires the data to be input in a vector format, rather than as an array. Therefore, we first reformat our data set. Note how the frequentist model fit is much faster than using MCMC (though, admittedly, our MCMC settings are a little overkill).

```
Cvec <- as.vector(C)              # Vector of M*J counts
elev.vec <- rep(elev, J)          # Vectorized elevation covariate
forest.vec <- rep(forest, J)      # Vectorized forest covariate
wind.vec <- as.vector(wind)       # Vectorized wind covariate
fac.site <- factor(rep(1:M, J))  # Site indicator (factor)
cbind(Cvec, fac.site, elev.vec, forest.vec, wind.vec) # Look at data

# Fit same model using maximum likelihood (NOTE: glmer uses ML instead of REML)
library(lme4)
summary(fm <- glmer(Cvec ~ elev.vec*forest.vec + wind.vec + (1|fac.site), family =
poisson))                         # Fit model
ranef(fm)                         # Print zero-centered random effects

Generalized linear mixed model fit by maximum likelihood (Laplace Approximation)
['glmerMod']
 Family: poisson ( log )
Formula: Cvec ~ elev.vec * forest.vec + wind.vec + (1 | fac.site)
[ ... ]
```

```
Random effects:
 Groups    Name        Variance Std.Dev.
 fac.site (Intercept) 0.06461  0.2542
Number of obs: 801, groups:  fac.site, 267

Fixed effects:
                     Estimate  Std. Error z value  Pr(>|z|)
(Intercept)          -0.83243     0.07252  -11.48  <2e-16 ***
elev.vec             -1.69303     0.10480  -16.16  <2e-16 ***
forest.vec            2.12447     0.09546   22.25  <2e-16 ***
wind.vec             -1.70929     0.06326  -27.02  <2e-16 ***
elev.vec:forest.vec   1.39146     0.15582    8.93  <2e-16 ***
[ ... ]
```

We compare the fixed-effects estimates in a table below, and also the random-effects estimates from the Bayesian and the non-Bayesian analyses in a table and a graph (Figure 5.17(a)).

```
# Compare fixed-effects estimates (in spite of the confusing naming in glmer output),
Bayesian post. means and sd left, frequentist MLEs and SEs right
print(cbind(out8$summary[c(1:2, 270:273), 1:2], rbind(summary(fm)$coef[1,1:2],
c(sqrt(summary(fm)$varcor$fac.site), NA), summary(fm)$coef[c(2,3,5,4),1:2])), 3)
            mean     sd Estimate Std. Error
mu.alpha   -0.837 0.0758   -0.832     0.0725
sd.alpha    0.264 0.0507    0.254         NA
alpha[1]   -1.693 0.1079   -1.693     0.1048
alpha[2]    2.126 0.0976    2.124     0.0955
alpha[3]    1.389 0.1602    1.391     0.1558
alpha[4]   -1.712 0.0650   -1.709     0.0633
```

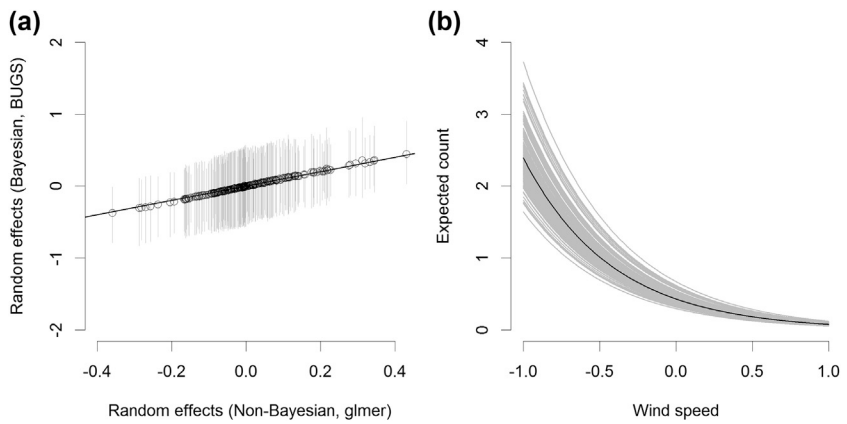

FIGURE 5.17

(a) Estimates of random site intercepts from a Bayesian (with 95% CRI in grey) and non-Bayesian analysis of the Poisson GLMM. (b) Bayesian estimates of site-specific relationships between the expected count of great tits and wind speed (black line is the population average).

```
# Compare graphically non-Bayesian and Bayesian random effects estimates
Freq.re <- ranef(fm)$fac.site[,1]      # Non-Bayesian estimates (MLEs)
Bayes.re <- out8$summary[274:540,]     # Bayesian estimates

par(mfrow = c(1, 2), mar = c(5,5,3,2), cex.lab = 1.5, cex.axis = 1.5) # Fig. 5.17 (a)
plot(Freq.re, Bayes.re[,1], xlab = "Non-Bayesian (glmer)", ylab = "Bayesian (BUGS)",
xlim = c(-0.4, 0.42), ylim = c(-2, 2), frame.plot = F, type = "n")
segments(Freq.re, Bayes.re[,3], Freq.re, Bayes.re[,7], col = "grey", lwd = 0.5)
abline(0, 1, lwd = 2)
points(Freq.re, Bayes.re[,1])
```

Once again, we note what many consider to be a comforting similarity between the non-Bayesian and the Bayesian parameter estimates, even with respect to their uncertainty (SEs and posterior standard deviations, respectively). In the Bayesian analysis, exact (i.e., not asymptotic) uncertainty estimates of the random effects are obtained easily (the grey error bars in Figure 5.17(a)), while for the non-Bayesian estimate, these would be harder to get (presumably by bootstrapping).

Finally, we plot predictions of the relationship between the expected counts and wind speed for each site separately and at a determined value of the other covariates (elevation and forest cover; Figure 5.17(b)). We could use the posterior samples of all parameters to produce uncertainty intervals as well, but this would be graphically unwieldy; hence, we plot only the posterior means. For clarity, we write the entire linear predictor, including the effects of the three other covariates set at zero (i.e., at their mean value).

```
wind.pred <- seq(-1, 1, , 1000)   # Covariate values for prediction
pred <- array(NA, dim = c(1000, 267))
for(i in 1:267){
  pred[,i]<- exp(out8$mean$alpha0[i] + out8$mean$alpha[1] * 0 + out8$mean$alpha[2] * 0 +
out8$mean$alpha[3] * 0 + out8$mean$alpha[4] * wind.pred)   # Predictions for each site
}

matplot(wind.pred, pred, type = "l", lty = 1, col = "grey", xlab = "Wind speed",
ylab = "Expected count", frame.plot = F, ylim = c(0, 4)) # Fig. 5.17 (b)
lines(wind.pred, exp(out8$mean$mu.alpha + out8$mean$alpha[4] * wind.pred),
col = "black", lwd = 3)
```

Note that for HMs, it is not straightforward to compute a quantity like the proportion of explained variation (R^2). The reason is that one may define such quantities for every level in the hierarchy of the model. For an attempt at defining R^2-like quantities in HMs, see Gelman and Pardoe (2006).

Often, fitting a model is only the first step of an analysis and much can be learnt by summarizing the results of that fit. This is true especially of a Bayesian analysis, where rich insights are possible based on the full posterior distributions and functions thereof, as we have already seen. The ability to make direct probability statements about parameters and the ease with which derived quantities such as predictions of the response for selected values of covariates can be computed, with a full assessment of the uncertainty, are great and underused benefits of a Bayesian analysis using MCMC.

5.14 RANDOM-EFFECTS BINOMIAL GLM (BINOMIAL GLMM)

We conclude our overview of "standard" GLMs and GLMMs by fitting a random-effects model with binomial response. We convert the counts for each survey at a site into "presence/absence," or more accurately "detection/nondetection," observations—i.e., "squash" the counts into zeros and ones, depending on whether they are equal to zero or greater.

```
# Get detection/nondetection data
y <- C
y[y > 0] <- 1
```

We then fit a kind of "naive" site-occupancy model, where we describe the detection/nondetection observations by a logistic regression with a continuous site-specific random effect assumed to be drawn from a normal distribution. This means that we assume that the effect of wind is specific to each site, but that the slopes of wind among all sites cluster around some common mean with a variance that is estimated. In contrast to the site-occupancy model (see Chapter 10), which is a Bernoulli–Bernoulli mixture, this model is a Bernoulli–normal mixture. We fit the following GLMM to the binary detection-nondetection observations y_{ij} at site i during replicate survey j:

$$y_{ij} \sim Bernoulli(\theta_{ij})$$

$$\text{logit}(\theta_{ij}) = \alpha_{0,i} + \alpha_1 * elev_i + \alpha_2 * forest_i + \alpha_3 * elev_i * forest_i + \alpha_{4,i} * wind_{ij}$$

$$\alpha_{0,i} \sim Normal\left(\mu_{\alpha 0}, \sigma^2_{\alpha 0}\right)$$

$$\alpha_{4,i} \sim Normal\left(\mu_{\alpha 4}, \sigma^2_{\alpha 4}\right)$$

For illustration, we now fit a slightly more complicated random-effects model than in the previous section, since in addition to the site-specific intercepts we now also specify the coefficients of wind to be random, by defining them to be draws from another (normal) prior distribution, with hyperparameters that we estimate. Other than that, we keep the linear model of this Bernoulli GLMM identical to that of the Poisson GLMM in the previous section, to emphasize that the linear model can be chosen entirely separately from the distribution chosen to describe the randomness in the response.

```
# Bundle data
win.data <- list(y = y, M = nrow(y), J = ncol(y), elev = elev, forest = forest,
elev.forest = elev * forest, wind = wind)
str(win.data)
```

In the BUGS code, we have to take the coefficient `alpha4` out of the vector `alpha`, since it is no longer a scalar, but a vector itself.

```
# Specify model in BUGS language
cat(file = "RE.Bernoulli.txt","
model {

# Priors
mu.alpha0 <- logit(mean.theta)      # Random intercepts
mean.theta ~ dunif(0,1)
```

```
tau.alpha0 <- pow(sd.alpha0, -2)
sd.alpha0 ~ dunif(0, 10)
mu.alpha4 ~ dnorm(0, 0.001)                  # Random slope on wind
tau.alpha4 <- pow(sd.alpha4, -2)
sd.alpha4 ~ dunif(0, 10)
for(k in 1:3){
  alpha[k] ~ dnorm(0, 0.001)                 # Slopes
}

# Likelihood
for (i in 1:M){
 alpha0[i] ~ dnorm(mu.alpha0, tau.alpha0)  # Intercept random effects
 re00[i] <- alpha0[i] - mu.alpha0           # same zero-centered
 alpha4[i] ~ dnorm(mu.alpha4, tau.alpha4)  # Slope random effects
 re04[i] <- alpha4[i] - mu.alpha4           # same zero-centered
 for(j in 1:J){
  y[i,j] ~ dbern(theta[i,j])
  logit(theta[i,j]) <- alpha0[i] + alpha[1] * elev[i] + alpha[2] * forest[i] +
alpha[3] * elev.forest[i] + alpha4[i] * wind[i,j]
 }
}
}")

# Other model run preparations
inits <- function() list(alpha0 = rnorm(M), alpha4 = rnorm(M))# Inits
params <- c("mu.alpha0", "sd.alpha0", "alpha0", "alpha", "mu.alpha4", "sd.alpha4",
"alpha4", "re00", "re04")                    # Params
ni <- 30000 ; nt <- 25 ; nb <- 5000 ; nc <- 3    # MCMC settings

# Call WinBUGS from R .... and crash !
out9 <- bugs(win.data, inits, params, "RE.Bernoulli.txt", n.chains = nc,
n.thin = nt, n.iter = ni, n.burnin = nb, debug = TRUE, bugs.directory = bugs.dir,
working.directory = getwd())
```

While WinBUGS gets an "undefined real result" crash, JAGS works fine; this is an example that shows that JAGS is sometimes numerically more robust than WinBUGS. (We could get WinBUGS to run by adding into the model some numerical "stabilization"—see trick 15 in Appendix 1 of Kéry and Schaub, 2012.)

```
# Call JAGS from R (ART 2.5 min)
out9 <- jags(win.data, inits, params, "RE.Bernoulli.txt", n.chains = nc, n.thin = nt,
n.iter = ni, n.burnin = nb)
par(mfrow = c(2,2))
traceplot(out9, c("mu.alpha0", "sd.alpha0", "alpha[1:3]", "mu.alpha4", "sd.alpha4"))
print(out9, 3)
```

Again, if you run into memory problems, do not save the posterior samples for the zero-centered random effects, but compute them in R instead. Finally, we fit this model non-Bayesianly using maximum likelihood (ML) in the package `lme4` after vectorizing the data.

```
yvec <- as.vector(y)                  # Vector of M*J counts
elev.vec <- rep(elev, J)              # Vectorized elevation covariate
forest.vec <- rep(forest, J)          # Vectorized forest covariate
wind.vec <- as.vector(wind)           # Vectorized wind covariate
fac.site <- factor(rep(1:M, J))       # Site indicator (factor)
cbind(yvec, fac.site, elev.vec, forest.vec, wind.vec) # Look at data

# Fit same model using maximum likelihood
library(lme4)          # Load package
summary(frem <- glmer(yvec ~ elev.vec*forest.vec + wind.vec + (wind.vec || fac.site),
family = binomial))        # Fit model

Random effects:
 Groups      Name         Variance  Std.Dev.
 fac.site    (Intercept)  4.255     2.063
 fac.site.1  wind.vec     8.779     2.963
Number of obs: 801, groups:     fac.site, 267

Fixed effects:
                     Estimate Std. Error  z value   Pr(>|z|)
(Intercept)           -0.6381     0.2504   -2.548  0.010837 *
elev.vec              -3.8703     0.7855   -4.927  8.35e-07 ***
forest.vec             5.7574     1.0614    5.424  5.82e-08 ***
wind.vec              -5.9007     1.1714   -5.037  4.72e-07 ***
elev.vec:forest.vec    3.8181     1.0009    3.815  0.000136 ***

# Compare Bayesian and non-Bayesian estimates
print(out9$summary[c(1:2, 270:274),c(1:3,7:9)], 4)
               mean      sd     2.5%    97.5%    Rhat n.eff
mu.alpha0   -0.6091  0.2414  -1.1101  -0.1457  0.9996  3000
sd.alpha0    2.0662  0.4821   1.2639   3.2134  1.0014  3000
alpha[1]    -3.8639  0.6980  -5.5057  -2.7587  1.0032  1633
alpha[2]     5.7575  0.9339   4.3289   7.9514  1.0057  1322
alpha[3]     3.7679  0.9383   2.1356   5.8646  1.0001  3000
mu.alpha4   -5.8058  1.0006  -8.1132  -4.3193  1.0067  1443
sd.alpha4    2.4177  1.0510   0.5137   4.6309  1.0034  1086
```

We find slightly less agreement between the two analyses now than in all the previous examples. This is perhaps not surprising given that binary data contain much less information about all the parameters of this fairly complex model. Next, we look at the estimates of the random effects.

```
(re <- ranef(frem))                    # Print zero-centered random effects
$fac.site
        (Intercept)          wind.vec
1     0.01538103261  -0.5521873386140
2     0.13475066011  -0.0394227279709
3    -1.07204621369  -1.4696483198851
[...]
```

```
265   0.63961927035   2.0279735503384
266  -1.07504683084  -0.1964205985223
267  -0.00131161668   0.0019269593948
```

The random-effects estimates of the site-specific relationships between the observation of at least one great tit (i.e., our response) and wind speed can be plotted (Figure 5.18(a)).

```
par(mfrow = c(1, 3), mar = c(5,5,3,2), cex.lab = 1.5, cex.axis = 1.5)
pop.mean.int <- summary(frem)$coef[1,1]
pop.mean.slope <- summary(frem)$coef[4,1]
plot(sort(wind.vec), plogis(pop.mean.int + sort(wind.vec) * pop.mean.slope), type = "l",
xlab = "Wind speed", ylab = "Prob. to count >0 great tits", lwd = 3, frame.plot = F)
for(i in 1:267){
  lines(sort(wind.vec), plogis(pop.mean.int + re$fac.site[i,1] + sort(wind.vec) *
(pop.mean.slope + re$fac.site[i,2])), lwd = 1, col = i)
}
```

To conclude, we plot a response surface of the expected detection/nondetection probability for a grid of elevation by forest cover along with its uncertainty (Figure 5.18(b) and (c)).

```
# Compute expected detection/nondetection probability for a grid of elevation and forest
cover, at wind-speed = 0 (covariate average) and for hypermean of intercepts alpha0
n.sims <- length(out9$sims.list$mu.alpha0)
elev.pred <- seq(-1, 1,,100)          # Values of elevation
forest.pred <- seq(-1,1,,100)         # Values of forest cover
pred.array <- array(NA, dim = c(100, 100, n.sims)) # Prediction array
```

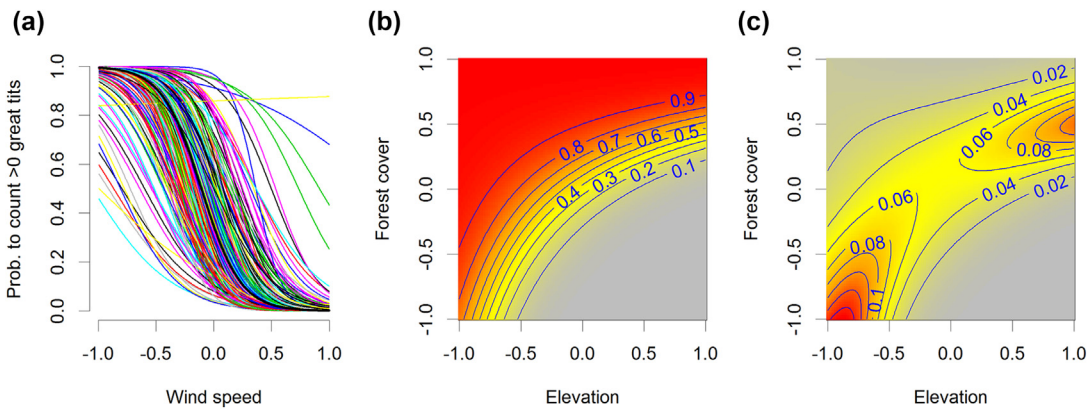

FIGURE 5.18

Site-specific estimates (ML) of the relationship between wind speed and the probability to count at least one great tit (a), and Bayesian predictions of detection/nondetection probability in a grid of elevation and forest cover ((b) posterior mean, (c) posterior standard deviation). Compare this with Figure 4.3.

```
for(i in 1:100){
  for(j in 1:100){
    pred.array[i,j,] <- plogis(out9$sims.list$mu.alpha0 + out9$sims.list$alpha[,1] *
elev.pred[i] + out9$sims.list$alpha[,2] * forest.pred[j] + out9$sims.list$alpha[,3] *
elev.pred[i] * forest.pred[j])
  }
}
pm.pred.array <- apply(pred.array, c(1,2), mean)  # Get posterior mean
psd.pred.array <- apply(pred.array, c(1,2), sd)   # Get posterior sd

mapPalette <- colorRampPalette(c("grey", "yellow", "orange", "red"))
image(x=elev.pred, y= forest.pred, z=pm.pred.array, col = mapPalette(100),
xlab = "Elevation", ylab = "Forest cover")
contour(x=elev.pred, y=forest.pred, z= pm.pred.array, add = TRUE, lwd = 1)

image(x=elev.pred, y= forest.pred, z=psd.pred.array, col = mapPalette(100),
xlab = "Elevation", ylab = "Forest cover")
contour(x=elev.pred, y=forest.pred, z= psd.pred.array, add = TRUE, lwd = 1)
```

5.15 GENERAL STRATEGY OF MODEL BUILDING WITH BUGS

We have introduced statistical modeling with BUGS for some of the simplest possible models: GLMs with just a few covariates and with zero to two random effects. When you start modeling your real data, there are a *lot* of things that can and frequently will go wrong. Of course, many problems can be avoided with a little experience. So, can we give a few tips to avoid the many possible pitfalls in BUGS modeling?

Unfortunately, this is not possible in general. We have given a list of BUGS survival tips elsewhere (see Appendix 1 in Kéry and Schaub, 2012). No doubt, there will be additional anecdotal wisdom available *somewhere*, which in some cases may be essential to success. Here we mention what we believe are some of the most important, general tips.

1. *By far* the single most important BUGS modeling tip is: ***Always start from the simplest possible version of your problem!*** (Or at least a very simple one.) Strip down your model to its bare bones and toss out any nonessential details at the start, such as covariates, time dependence, and spatial or temporal autocorrelation. Start with a very simple caricature of the model you want to get at; in a sense, *start with a model of your model*. Once you get this simple model to run and you understand it, add details one at a time until you reach the desired model structure. This stepwise approach will help you recognize and diagnose problems when something goes wrong—e.g., when some parameter estimates change dramatically when going from one model to a slight variant. Similarly, when something goes wrong in your modeling, go back to the last (simpler) model variant that did work and then try to identify what caused the problem in a new step-up approach. Arguably, this advice is essential for any kind of even moderately advanced statistical modeling and we acknowledge Gelman et al. (2003), who stress this in their book. There is also a heuristic advantage of such a step-up modeling strategy that carries over to all of science: there can hardly ever be a point in trying to understand a big version of a problem unless you first understand the simplest version of the problem. Thus, this modeling strategy is just a variant of a powerful model for the process of learning in general.

2. *Start with a template that runs*. Only very rarely will you write code for a new analysis from scratch; it is much too easy to get bogged down in myriad errors that are easy to commit. Rather, it is usually far more efficient to take code from a version of your problem, that you developed earlier for a related problem, that a colleague has given you, or that you found in a book or on the Internet, and then adapt that code toward the model you want to fit.
3. *Play with simulated data first*. If you get the right answers and understand your model, then you will be much better prepared to fit the model to your real data and understand it.
4. *Do not forget to google cryptic error messages* (B. Schmidt, pers. comm.).
5. *Write tidy code* (inspired by B. Schmidt).

5.16 SUMMARY AND OUTLOOK

In this chapter, we have covered much ground. In an applied way, we have wrapped up much of the contents of Chapters 2–4. Most importantly, we have presented a crash course in BUGS, which we use throughout the book to fit HMs. All three BUGS incarnations (WinBUGS, OpenBUGS, JAGS) as well as the new NIMBLE software (NIMBLE Development Team, 2015; de Valpine et al., in review) use virtually the same model definition language, the BUGS language, and hence from R you can readily send a model to any of them. Indeed, it often pays to try a model in more than one BUGS engine; sometimes an analysis does not work in one but will work in another, and typically one engine is faster. We have also introduced our typical work flow for a Bayesian analysis using BUGS run from R. We have illustrated many times the richness of posterior inferences in a Bayesian analysis. We fitted all models with maximum likelihood (ML) as well, thereby emphasizing that with reasonable sample size, Bayesian estimates with vague priors agree numerically very well with the corresponding MLEs. We have covered several types of linear models, GLMs, and two simple random-effects, or mixed, models (GLMMs).

To emphasize the value of generating and analyzing simulated data sets, we have worked exclusively with a simulated data set in this chapter, generated using the code from Chapter 4, and imagined these were counts of a small Eurasian passerine bird. We have encountered various ways in which such counts can be summarized or aggregated—e.g., to obtain presence/absence or species distribution data from the underlying counts. Finally, we have also covered quite an eclectic set of other topics that we think are important in applied Bayesian modeling, including prediction, GoF and residual diagnostics, the Bayesian treatment of missing values, proportion of variance explained, specification of nonstandard likelihoods using the zeros and the ones tricks, and alternative parameterizations of a model using moment matching. One potentially important topic that we have not touched upon in this chapter is model selection (see Burnham and Anderson, 2002; Kadane and Lazaar, 2004; O'Hara and Sillanpää, 2009; Tenan et al., 2014a; Hobbs and Hooten, 2015), but we introduced it in Chapter 2 and illustrate aspects of it throughout the book. We do not claim to cover any of these topics exhaustively. If Bayesian analysis, linear models, GLMs, or simple mixed models, and BUGS software is completely new to you, then you should complement your reading of this book with that of other books on these topics. At the very least, you should read the preceding Chapters 2–4 a couple of times.

We do not claim that all of the models in this chapter were the best possible analysis for our data set. For instance, the Poisson GLMM in Section 5.13 would probably not be our first choice; rather, we might adopt a binomial *N*-mixture model (see Chapter 6) for inference about abundance from such data. Similarly, the binomial GLMM in Section 5.14 is not the most natural model for such data.

Instead, we would naturally adopt a site-occupancy model (see Chapter 10) for inference about species occurrence from such data.

We wanted to introduce BUGS with the simplest models that you are likely to know well and use a lot: linear models, GLMs, and simple mixed models. These form the backbone of most of applied statistics in ecology, and yet, there are many aspects that ecologists may find confusing—e.g., the parameterization of factors, link functions, and random effects. The BUGS language can greatly enhance your understanding of these fundamental statistical models, because of the way these models are described in the BUGS language. BUGS may be an algorithmic black box, but it is the opposite of a modeling black box—in the BUGS language, the model specified is utterly transparent.

Nowadays, mixed models of a certain class in particular have become the workhorse models of many ecologists, and are widely fitted using functions such as `(g)lmer` in R or `PROC MIXED` in SAS. You can think of BUGS, with its simple but comprehensive model definition language, as a superpowerful `lmer` function for fitting virtually *any* kind of mixed (= hierarchical) model. The power of BUGS extends way beyond the "simple" mixed models with normal random effects, which are the only ones you can fit with `lmer`. It includes models with many nested levels of random effects, random effects that are nested or crossed, continuous or discrete-valued, and that can come from many distributions other than normal. Thus, BUGS allows you to easily specify a very large class of hierarchical (= mixed) models.

In almost all of the rest of the book, we will combine GLMs like those in this chapter to build HMs that exactly reflect the way in which we imagine the sequence of processes (e.g., ecological and observational) that produces the observed data. Hence, if you understand linear models, GLMs, and the concept of random effects, and know how to specify these modeling figures using the BUGS language, then you are very well prepared for building a vast number of HMs in a powerful and creative manner. Moreover, if you learn how to "walk" in BUGS, then you are likely to experience a completely exhilarating modeling freedom that you would never have dreamt of as an ecologist.

In other words, almost all of the rest of the book presents variants of HMs consisting of combinations of two to a couple of submodels, each of which can be described as a simple GLM in a sense. Their specific combination of GLM represents the basic structure of the particular model. Everything else then boils down to clever specification of a linear predictor to address the specific question and the specifics of your data collection protocol. In summary, a very good practical understanding of linear modeling and of GLMs is vital for your existence as a hierarchical modeler.

EXERCISES

1. In Section 5.3, fit the linear model that you get when typing in R `lm(Cmean ~ (elev + I(elev^2)) * (forest + I(forest^2)))`; i.e., add quadratic effects of elevation and forest cover including all pair-wise interaction effects.
2. In Section 5.6, the residuals are seen to be heterogeneous in the different groups (levels of `facFor`); see Figure 5.10. Why is this? What can be done about this? Plot the residuals for each group for the model in that section. Then, in BUGS fit a normal model with heterogeneous variances for each level of `facFor`. Alternatively, specify a linear model with an effect of the continuous covariate `forest` in the variance to see whether in this way the heterogeneous variance can be accommodated.

3. Fit a Poisson GLM to one of the count vectors (e.g., from the first survey; Section 5.6), and check whether the residual plots look better than those for the maximum count.
4. In Section 5.9, the residuals in the Poisson regression of the maximum count have too many large values. Try to add an "extra residual," assumed to be normally distributed, and convert the Poisson GLM into a Poisson-lognormal GLM to soak up that variability, and inspect whether you get new residuals that are more nearly symmetrical around zero. Check how the posterior standard deviations of the coefficient estimates change and whether the Bayesian p-value now indicates a fitting model.
5. In the Poisson GLM (Section 5.9), fit a quadratic and a cubic term for elevation as well. Are these additional polynomial terms "significant"?
6. In Section 5.11, fit a binomial GLM to the detection frequency of the great tit—i.e., to the number of surveys (out of three) that a great tit was detected at a site.
7. In 5.13, fit a normal–normal mixed model to the logarithm of the replicated counts.
8. In 5.13, turn the Poisson GLMM into a Poisson GLM with fixed site effects.
9. Fit a binomial GLMM in Section 5.14 to the detection frequency data (as in Exercise 6).
10. In the Bernoulli GLMM in Section 5.14, model the site-specific effect of wind speed by some site-specific covariates, such as elevation. That is, fit a model with the last line as this $\alpha_{4,i} \sim Normal(\mu_{\alpha 4} + \beta_{\alpha 4} * elev_i, \sigma^2_{\alpha 4})$. What proportion of the variance of the slope of wind speed is explained by elevation?
11. In Section 5.14, check whether increasing the number of replicates per site (for instance, to 10 or 20) improves the agreement between the Bayesian and the frequentist estimates.
12. To find out which estimates under the Bernoulli GLMM are more trustworthy with small samples (as those in Chapter 5.14), you can analyze simulated occupancy data where you have a known truth to compare with (see Exercise 4.11 in Chapter 4 and Section 10.5).