CHAPTER 16

Multilevel modeling in Bugs and R: the basics

In this chapter we introduce the fitting of multilevel models in Bugs as run from R. Following a brief introduction to Bayesian inference in Section 16.2, we fit a varying-intercept multilevel regression, walking through each step of the model. The computations in this chapter parallel Chapter 12 on basic multilevel models. Chapter 17 presents computations for the more advanced linear and generalized linear models of Chapters 12–15.

16.1 Why you should learn Bugs

As illustrated in the preceding chapters, we can quickly and easily fit many multilevel linear and generalized linear models using the `lmer()` function in R. Functions such as `lmer()`, which use point estimates of variance parameters, are useful but can run into problems. When the number of groups is small or the multilevel model is complicated (with many varying intercepts, slopes, and non-nested components), there just might not be enough information to estimate variance parameters precisely. At that point, we can get more reasonable inferences using a Bayesian approach that averages over the uncertainty in all the parameters of the model.

We recommend the following strategy for multilevel modeling:

1. Start by fitting classical regressions using the `lm()` and `glm()` functions in R. Display and understand these fits as discussed in Part 1 of this book.
2. Set up multilevel models—that is, allow intercepts and slopes to vary, using non-nested groupings if appropriate—and fit using `lmer()`, displaying as discussed in most of the examples of Part 2A.
3. As described in this part of the book, fit fully Bayesian multilevel models, using Bugs to obtain simulations representing inferential uncertainty about all the parameters in a model. Use these simulations to summarize uncertainty about coefficients, predictions, and other quantities of interest.
4. Finally, for some large or complicated models, it is necessary to go one step further and do some programming in R, as illustrated in Section 18.7.

For some analyses, it will not be necessary to go through all four steps. In fact, as illustrated in the first part of this book, often step 1 is enough.

In multilevel settings, the speed and convenience of `lmer()` allow us to try many specifications in building a model, and then the flexibility of Bugs gives us a chance to understand any particular model more fully. Also, as we discuss in Section 16.10, Bugs has an open-ended format that allows models to be expanded more generally than can be done using standard multilevel modeling packages.

16.2 Bayesian inference and prior distributions

The challenge in fitting a multilevel model is estimating the data-level regression (including the coefficients for all the group indicators) along with the group-level model. The most direct way of doing this is through *Bayesian inference*, a statistical method that treats the group-level model as "prior information" in estimating the individual-level coefficients. Chapter 18 discusses Bayesian inference as a generalization of least squares and maximum likelihood estimation. Here we briefly characterize the prior distributions that are commonly used in multilevel models.

In a Bayesian framework, all parameters must have prior distributions, and in the models of this book the prior distributions almost all fall into one of two categories: group-level models and noninformative uniform distributions. The group-level models themselves are either normal distributions (whose mean and standard deviation are themselves typically given noninformative prior distributions) or linear regressions (whose coefficients and error standard deviations are again typically modeled noninformatively).

Classical regression

Classical regression and generalized linear models represent a special case of multilevel modeling in which there is no group-level model—in Bayesian terms, no prior information. For the classical regression model, $y_i = X_i\beta + \epsilon_i$, with independent errors $\epsilon_i \sim \mathrm{N}(0, \sigma_y^2)$, the corresponding prior distribution is a uniform distribution on the entire range $(-\infty, \infty)$ for each of the components of β, and uniform on $(0, \infty)$ for σ_y as well.[1] That is, the classical model ascribes no structure to the parameters. Similarly, a classical logistic regression, $\Pr(y_i = 1) = \mathrm{logit}^{-1}(X_i\beta)$, has a uniform prior distribution on the components of β.

Simplest varying-intercept model

The simplest multilevel regression is a varying-intercept model with normally distributed individual and group-level errors: $y_i \sim \mathrm{N}(\alpha_{j[i]} + \beta x_i, \sigma_y^2)$ and $\alpha_j \sim \mathrm{N}(\mu_\alpha, \sigma_\alpha^2)$. The normal distribution for the α_j's can be thought of as a prior distribution for these intercepts. The parameters of this prior distribution, μ_α and σ_α, are called *hyperparameters* and are themselves estimated from the data. In Bayesian inference, all the hyperparameters, along with the other unmodeled parameters (in this case, β and σ_y) also need a prior distribution which, as in classical regression, is typically set to a uniform distribution.[2] The complete prior distribution can be written in probability notation as $p(\alpha, \beta, \mu_\alpha, \sigma_y, \sigma_\alpha) \propto \prod_{j=1}^{J} \mathrm{N}(\alpha_j | \mu_\alpha, \sigma_\alpha^2)$—that is, independent normal distributions for the α_j's so that their probability densities are multiplied to create the joint prior density.

Varying-intercept, varying-slope model

The more complicated model, $y_i = \alpha_{j[i]} + \beta_{j[i]} x_i + \epsilon_i$, has $2J$ modeled parameters, which are modeled as J pairs, (α_j, β_j). Their "prior distribution" is the bivariate normal distribution (13.1) on page 279, once again with independent uniform prior distributions on the hyperparameters.

[1] Technically, the classical least squares results are reproduced with a prior distribution that is uniform on $\log \sigma_y$, that is, the prior distribution $p(\log \sigma_y) \propto 1$, which is equivalent to $p(\sigma_y) \propto 1/\sigma_y$. The distinction between $p(\sigma_y) \propto 1$ and $p(\sigma_y) \propto 1/\sigma_y$ matters little in practice, and for convenience we work with the simpler uniform distribution on σ_y itself. See Gelman et al. (2003) for further discussion of such points.

[2] The inverse-gamma distribution is often used as a prior distribution for variance parameters; however, this model creates problems for variance components near zero, and so we prefer the uniform or, if more information is necessary, the half-t model; see Section 19.6.

Multilevel model with group-level predictors: exchangeability and prior distributions

We typically do not assign a model to coefficients of group-level predictors, or of individual-level predictors that do not vary by group. That is, in Bayesian terminology, we assign noninformative uniform prior distributions to these coefficients. More interestingly, a group-level regression induces different prior distributions on the group coefficients.

Consider a simple varying-intercept model with one predictor at the individual level and one at the group level:

$$\begin{aligned} y_i &= \alpha_{j[i]} + \beta x_i + \epsilon_i, \quad \epsilon_i \sim \mathrm{N}(0, \sigma_y^2), \text{ for } i = 1, \dots, n \\ \alpha_j &= \gamma_0 + \gamma_1 u_j + \eta_j, \quad \eta_j \sim \mathrm{N}(0, \sigma_\alpha^2), \text{ for } j = 1, \dots, J. \end{aligned} \tag{16.1}$$

The first equation, for the y_i's, is called the *data model* or the *likelihood* (see Sections 18.1–18.2). The second equation, for the α_j's, is called the *group-level model* or the *prior model.* (The choice of name does not matter except that it draws attention either to the grouped structure of the data or to the fact that the parameters α_j are given a probability model.)

The α_j's in (16.1) have different prior distributions. For any particular group j, its α_j has a prior distribution with mean $\hat{\alpha}_j = \gamma_0 + \gamma_1 u_j$ and standard deviation σ_α. (As noted, this prior distribution depends on unknown parameters $\gamma_0, \gamma_1, \sigma_\alpha$, which themselves are estimated from the data and have noninformative uniform prior distributions.) These prior estimates of the α_j's differ because they differ in the values of their predictor u_j; the α_j's thus are not "exchangeable" and have different prior distributions.

An equivalent way to think of this model is as an exchangeable prior distribution on the group-level errors, η_j. From this perspective, the α_j's are determined by the group-level predictors u_j and the η_j's, which are assigned a common prior distribution with mean 0 and standard deviation σ_α. This distribution represents the possible values of η_j in a hypothetical population from which the given J groups have been sampled.

Thus, a prior distribution in a multilevel model can be thought of in two ways: as models that represent a "prior," or group-level, estimate for each of the α_j's; or as a single model that represents the distribution of the group-level errors, η_j. When formulating models with group-level predictors in Bugs, the former approach is usually more effective (it avoids the step of explicitly defining the η_j's, thus reducing the number of variables in the Bugs model and speeding computation).

Noninformative prior distributions

Noninformative prior distributions are intended to allow Bayesian inference for parameters about which not much is known beyond the data included in the analysis at hand. Various justifications and interpretations of noninformative priors have been proposed over the years, including invariance, maximum entropy, and agreement with classical estimators. In our work, we consider noninformative prior distributions to be "reference models" to be used as a standard of comparison or starting point in place of the proper, informative prior distributions that would be appropriate for a full Bayesian analysis.

We view any noninformative prior distribution as inherently provisional—after the model has been fit, one should look at the posterior distribution and see if it makes sense. If the posterior distribution does not make sense, this implies that additional prior knowledge is available that has not been included in the model, and that contradicts the assumptions of the prior distribution (or some other part of the model) that has been used. It is then appropriate to go back and alter the model to be more consistent with this external knowledge.

16.3 Fitting and understanding a varying-intercept multilevel model using R and Bugs

We introduce Bugs by stepping through the varying-intercept model for log radon levels described in Section 12.3. We first fit classical complete-pooling and no-pooling models in R using `lm()`, then perform a quick multilevel fit using `lmer()` as described in Section 12.4, then fit the multilevel model in Bugs, as called from R.

Setting up the data in R

We begin by loading in the data: radon measurements and floors of measurement for 919 homes sampled from the 85 counties of Minnesota. (The dataset also contains several other measurements at the house level that we do not use in our model.) Because it makes sense to assume multiplicative effects, we want to work with the logarithms of radon levels; however, some of the radon measurements have been recorded as 0.0 picoCuries per liter. We make a simple correction by rounding these up to 0.1 before taking logs.

R code
```
srrs2 <- read.table ("srrs2.dat", header=TRUE, sep=",")
mn <- srrs2$state=="MN"
radon <- srrs2$activity[mn]
y <- log (ifelse (radon==0, .1, radon))
n <- length(radon)
x <- srrs2$floor[mn]              # 0 for basement, 1 for first floor
```

County indicators. We must do some manipulations in R to code the counties from 1 to 85:

R code
```
srrs2.fips <- srrs2$stfips*1000 + srrs2$cntyfips
county.name <- as.vector(srrs2$county[mn])
uniq.name <- unique(county.name)
J <- length(uniq.name)
county <- rep (NA, J)
for (i in 1:J){
  county[county.name==uniq.name[i]] <- i
}
```

There may very well be a better way to create this sort of index variable; this is just how we did it in one particular problem.

Classical complete-pooling regression in R

We begin with simple classical regression, ignoring the county indicators (that is, complete pooling):

R code
```
lm.pooled <- lm (y ~ x)
display (lm.pooled)
```

which yields

R output
```
            coef.est coef.se
(Intercept)  1.33     0.03
x           -0.61     0.07
  n = 919, k = 2
  residual sd = 0.82, R-Squared = 0.07
```

Classical no-pooling regression in R

Including a constant term and 84 county indicators. Another alternative is to include all 85 indicators in the model—actually, just 84 since we already have a constant term:

R code
```
lm.unpooled.0 <- lm (formula = y ~ x + factor(county))
```

which yields

R output
```
                   coef.est coef.se
(Intercept)         0.84     0.38
x                  -0.72     0.07
factor(county)2     0.03     0.39
factor(county)3     0.69     0.58
. . .
factor(county)85    0.35     0.66
  n = 919, k = 86
  residual sd = 0.76, R-Squared = 0.29
```

This is the no-pooling regression. Here, county 1 has become the reference condition. Thus, for example, the log radon levels in county 2 are 0.03 higher, and so unlogged radon levels are approximately 3% higher in county 2, on average, than those in county 1, after controlling for the floor of measurement. The model including county indicators fits quite a bit better than the previous regression (the residual standard deviation has declined from 0.82 to 0.76, and R^2 has increased from 7% to 29%)—but this is no surprise since we have added 84 predictors. The estimates for the individual counties in this new model are highly uncertain (for example, counties 2, 3, and 85 shown above are not statistically significantly different from the default county 1).

Including 85 county indicators with no constant term. For making predictions about individual counties, it it slightly more convenient to fit this model without a constant term, so that each county has its own intercept:

R code
```
lm.unpooled <- lm (formula = y ~ x + factor(county) - 1)
```

(adding "-1" to the linear model formula removes the constant term), to yield

R output
```
                   coef.est coef.se
x                  -0.72     0.07
factor(county)1     0.84     0.38
factor(county)2     0.87     0.10
. . .
factor(county)85    1.19     0.53
  n = 919, k = 86
  residual sd = 0.76, R-Squared = 0.77
```

These estimates are consistent with the previous parameterization—for example, the estimate for county 1 is 0.84 with a standard error of 0.38, which is identical to the inference for the intercept from the previous model. The estimate for county 2 is 0.87, which equals the intercept from the previous model, plus 0.03, which was the estimate for county 2, when county 1 was a baseline. The standard error for county 2 has declined because the uncertainty about the intercept for county 2 is less than the uncertainty about the difference between counties 1 and 2. Moving to the bottom of the table, the residual standard deviation is unchanged, as is appropriate given that this is just a shifting of the constant term within an existing model. Oddly, however, the R^2 has increased from 29% to 77%—this is because the `lm()` function calculates explained variance differently for models with no constant term, an issue that does not concern us. (For our purposes, the correct R^2 for this model is 29%; see Section 21.5.)

Setting up a multilevel regression model in Bugs

We set up the following Bugs code for a multilevel model for the radon problem, saving it in the file `radon.1.bug` (in the same working directory that we are using for our R analyses). Section 16.4 explains this model in detail.

Bugs code

```
model {
  for (i in 1:n){
    y[i] ~ dnorm (y.hat[i], tau.y)
    y.hat[i] <- a[county[i]] + b*x[i]
  }
  b ~ dnorm (0, .0001)
  tau.y <- pow(sigma.y, -2)
  sigma.y ~ dunif (0, 100)
  for (j in 1:J){
    a[j] ~ dnorm (mu.a, tau.a)
  }
  mu.a ~ dnorm (0, .0001)
  tau.a <- pow(sigma.a, -2)
  sigma.a ~ dunif (0, 100)
}
```

Calling Bugs from R

Our R environment has already been set up to be ready to call Bugs (see Appendix C). We execute the following R code to set up the data, initial values, and parameters to save for the Bugs run:

R code

```
radon.data <- list ("n", "J", "y", "county", "x")
radon.inits <- function (){
  list (a=rnorm(J), b=rnorm(1), mu.a=rnorm(1),
        sigma.y=runif(1), sigma.a=runif(1))}
radon.parameters <- c ("a", "b", "mu.a", "sigma.y", "sigma.a")
```

Once again, these details will be explained in Section 16.4. The R code continues with a call to Bugs in "debug" mode:

R code

```
radon.1 <- bugs (radon.data, radon.inits, radon.parameters,
  "radon.1.bug", n.chains=3, n.iter=10, debug=TRUE)
```

Here, we have run Bugs for just 10 iterations in each chain. We can look at the output in the Bugs window. When we close the Bugs window, R resumes. In the R window, we can type

R code

```
plot (radon.1)
print (radon.1)
```

and inferences for `a, b, mu.a, sigma.y, sigma.a` (the parameters included in the `radon.parameters` vector that was passed to Bugs) are displayed in a graphics window and in the R console. Having ascertained that the program will run, we now run it longer:

R code

```
radon.1 <- bugs (radon.data, radon.inits, radon.parameters,
  "radon.1.bug", n.chains=3, n.iter=500)
```

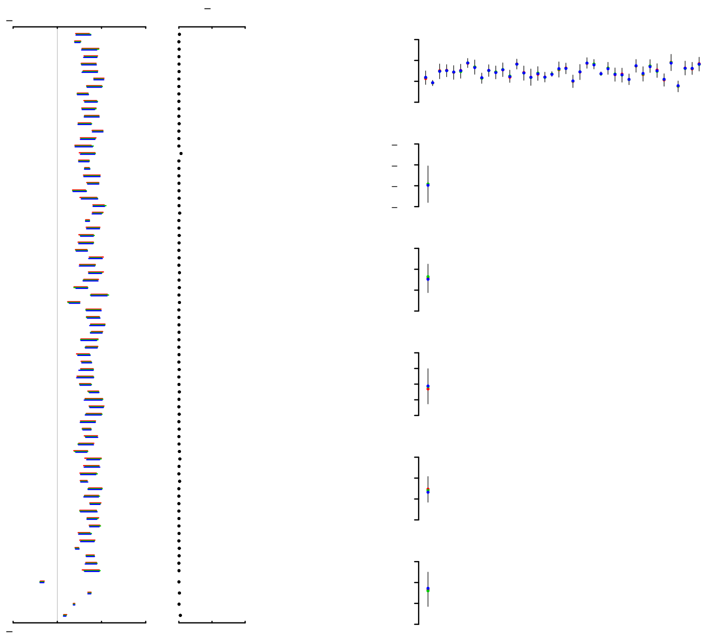

Figure 16.1 *Summary of Bugs simulations for the multilevel regression with varying intercepts and constant slope, fit to the Minnesota radon data.* `R-hat` *is near 1 for all parameters, indicating approximate convergence. The intervals for the county intercepts α_j indicate the uncertainty in these estimates (see also the error bars in Figure 12.3a on page 256).*

A Bugs window opens, and after about a minute, it closes and the R window becomes active again. Again, we can `plot` and `print` the fitted Bugs object, yielding the display shown in Figure 16.1, and the following text in the R window:

R output

```
Inference for Bugs model at "radon.1.bug"
 3 chains, each with 500 iterations (first 250 discarded)
 n.sims = 750 iterations saved
           mean   sd   2.5%    25%    50%    75%  97.5% Rhat n.eff
a[1]        1.2  0.3    0.7    1.0    1.2    1.3    1.7    1   230
a[2]        0.9  0.1    0.7    0.9    0.9    1.0    1.1    1   750
. . .
a[85]       1.4  0.3    0.8    1.2    1.4    1.6    1.9    1   750
b          -0.7  0.1   -0.8   -0.7   -0.7   -0.6   -0.5    1   750
mu.a        1.5  0.1    1.3    1.4    1.5    1.5    1.6    1   240
sigma.y     0.8  0.0    0.7    0.7    0.8    0.8    0.8    1   710
sigma.a     0.3  0.0    0.3    0.3    0.3    0.4    0.4    1    95
deviance 2093.4 12.5 2069.0 2085.0 2093.0 2101.7 2119.0    1   510

pD = 77.7 and DIC = 2171 (using the rule, pD = var(deviance)/2)
DIC is an estimate of expected predictive error (lower deviance is better).
```

```
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).
```

The first seven columns of numbers give inferences for the model parameters. For example, α_1 has a mean estimate of 1.2 and a standard error of 0.3. The median estimate of α_1 is 1.2, with a 50% uncertainty interval of $[1.0, 1.3]$ and a 95% interval of $[0.7, 1.7]$. Moving to the bottom of the table, the 50% interval for the floor coefficient, β, is $[-0.7, -0.6]$, and the average intercept, μ_α, is estimated at 1.5 (see Figure 12.4 on page 257). The within-county standard deviation σ_y is estimated to be much larger than the between-county standard deviation σ_α. (This can also be seen in Figure 12.4—the scatter between points within each county is much larger than the county-to-county variation among the estimated regression lines.)

At the bottom of the table, we see p_D, the estimated effective number of parameters in the model, and DIC, the deviance information criterion, an estimate of predictive error. We return to these in Section 24.3.

Finally, we consider the rightmost columns of the output from the Bugs fit. `Rhat` gives information about convergence of the algorithm. At convergence, the numbers in this column should equal 1; before convergence, it should be larger than 1. If `Rhat` is less than 1.1 for all parameters, then we judge the algorithm to have approximately converged, in the sense that the parallel chains have mixed well (see Section 18.6 for more context on this). The final column, `n.eff`, is the "effective sample size" of the simulations, also discussed in Section 18.6.

Summarizing classical and multilevel inferences graphically

We can use R to summarize our inferences obtained from Bugs. For example, to display individual-level regressions as in Figure 12.4 on page 257, we first choose the counties to display, construct jittered data, and compute the range of the data (so that all eight counties will be displayed on a common scale):

R code
```
display8 <- c (36, 1, 35, 21, 14, 71, 61, 70)   # choose 8 counties
x.jitter <- x + runif(n,-.05,.05)
x.range <- range (x.jitter)
y.range <- range (y[!is.na(match(county,display8))])
```

We then pull out the appropriate parameter estimates from the classical fits:

R code
```
a.pooled <- coef(lm.pooled)[1]             # complete-pooling intercept
b.pooled <- coef(lm.pooled)[2]             # complete-pooling slope
a.nopooled <- coef(lm.unpooled)[2:(J+1)]   # no-pooling intercepts
b.nopooled <- coef(lm.unpooled)[1]         # no-pooling slope
```

and summarize the parameters in the fitted multilevel model by their median estimates, first attaching the Bugs object and then computing the medians, component by component:

R code
```
attach.bugs (radon.1)
a.multilevel <- rep (NA, J)
for (j in 1:J){
  a.multilevel[j] <- median (a[,j])
}
b.multilevel <- median (b)
```

The computation for a is more complicated;[3] because a is a vector of length J, its `n.sims` simulations are saved as a matrix with dimensions `n.sims` $\times$ `J`.

We can now make the graphs in Figure 12.4:

R code
```
par (mfrow=c(2,4))
for (j in display8){
  plot (x.jitter[county==j], y[county==j], xlim=c(-.05,1.05),
    ylim=y.range, xlab="floor", ylab="log radon level", main=uniq[j])
  curve (a.pooled + b.pooled*x, lwd=.5, lty=2, col="gray10", add=TRUE)
  curve (a.nopooled[j] + b.nopooled*x, lwd=.5, col="gray10", add=TRUE)
  curve (a.multilevel[j] + b.multilevel*x, lwd=1, col="black", add=TRUE)
}
```

(This can be compared with the code on page 261 for making this graph using the point estimates from `lmer()`.)

To display the estimates and uncertainties versus sample size as in Figure 12.3b on page 256, we first set up the sample size variable,

R code
```
sample.size <- as.vector (table (county))
sample.size.jitter <- sample.size*exp(runif(J,-.1,.1))
```

and then plot the estimates and standard errors:

R code
```
plot (sample.size.jitter, a.multilevel, xlab="sample size in county j",
  ylim=range(y), ylab=expression (paste ("intercept, ", alpha[j],
  "   (multilevel inference)")), pch=20, log="x")
for (j in 1:J){
  lines (rep(sample.size.jitter[j],2), median(a[,j])+c(-1,1)*sd(a[,j]))
}
abline (a.pooled, 0, lwd=.5)
```

The last line of code above places a thin horizontal line at the complete-pooling estimate, as can be seen in Figure 12.3b.

16.4 Step by step through a Bugs model, as called from R

A Bugs model must include a specification for every data point, every group-level parameter, and every hyperparameter. We illustrate here for the radon model shown in the previous section. For help in programming models in Bugs in general, open the Bugs window and click on Help, then Examples. Chapter 19 discusses some methods for running Bugs faster and more reliably.

The individual-level model

The varying-intercept model on page 350 starts with a probability distribution for each data point; hence the looping from 1 to n:

Bugs code
```
model {
  for (i in 1:n){
    y[i] ~ dnorm (y.hat[i], tau.y)
    y.hat[i] <- a[county[i]] + b*x[i]
  }
```

[3] The three lines computing `a.multilevel` could be compressed into the single command, `a.multilevel <- apply (a, 2, median)`, but we find it clearer to compute the components one at a time.

The Bugs code is abstracted in several steps from the model

$$y_i \sim \mathrm{N}(\alpha_{j[i]} + \beta x_i, \sigma_y^2). \tag{16.2}$$

First, Bugs does not allow composite expressions in its distribution specifications (we cannot write `y[i] ~ dnorm (a[county[i]] + b*x[i], tau.y)` because the first argument to `dnorm` is too complex), and so we split (16.2) into two lines:

$$\begin{aligned} y_i &\sim \mathrm{N}(\hat{y}_i, \sigma_y^2) \\ \hat{y}_i &= \alpha_{j[i]} + \beta x_i. \end{aligned} \tag{16.3}$$

In the notation of our variables in R, this second line becomes

$$\hat{y}_i = \mathtt{a}_{\mathtt{county}[i]} + \mathtt{b} \cdot \mathtt{x}_\mathtt{i}. \tag{16.4}$$

(In the program we use a, b rather than α, β to make the code easier to follow.)

Second, as noted above, we replace the subscripts $j[i]$ in the model with the variable `county[i]` in R and Bugs, thus freeing j to be a looping index.

Finally, Bugs parameterizes normal distributions in terms of the inverse-variance, $\tau = 1/\sigma^2$, a point we shall return to shortly.

The group-level model

The next step is to model the group-level parameters. For our example here, these are the county intercepts α_j, which have a common mean μ_α and standard deviation σ_α:

$$\alpha_j \sim \mathrm{N}(\mu_\alpha, \sigma_\alpha^2).$$

This is expressed almost identically in Bugs:

Bugs code
```
for (j in 1:J){
  a[j] ~ dnorm (mu.a, tau.a)
}
```

The only difference from the preceding statistical formula is the use of the inverse-variance parameter $\tau_\alpha = 1/\sigma_\alpha^2$.

Prior distributions

Every parameter in a Bugs model must be given either an assignment (as is done for the temporary parameters `y.hat[i]` defined within the data model) or a distribution. The parameters `a[j]` were given a distribution as part of the group-level model, but this still leaves `b` and `tau.y` from the data model and `mu.a` and `tau.a` from the group-level model to be defined.

The specifications for these parameters are called *prior distributions* because they must be specified before the model is fit to data. In the radon example we follow common practice and use noninformative prior distributions:

Bugs code
```
b ~ dnorm (0, .0001)
mu.a ~ dnorm (0, .0001)
```

The regression coefficients μ_α and β are each given normal prior distributions with mean 0 and standard deviation 100 (thus, they each have inverse-variance $1/100^2 = 10^{-4}$). This states, roughly, that we expect these coefficients to be in the range $(-100, 100)$, and if the estimates are in this range, the prior distribution is providing very little information in the inference.

Bugs code

```
    tau.y <- pow(sigma.y, -2)
    sigma.y ~ dunif (0, 100)
    tau.a <- pow(sigma.a, -2)
    sigma.a ~ dunif (0, 100)
```

We define the inverse-variances τ_y and τ_α in terms of the standard-deviation parameters, σ_y and σ_α, which are each given uniform prior distributions on the range $(0, 100)$.

Scale of prior distributions

Constraining the absolute values of the parameters to be less than 100 is not a serious restriction—the model is on the log scale, and there is no way we will see effects as extreme as -100 or 100 on the log scale, which would correspond to multiplicative effects of e^{-100} or e^{100}.

Here are two examples where the prior distributions with scale 100 would *not* be noninformative:

- In the regression of earnings (in dollars) on height, the coefficient estimate is 1300 (see model (4.1) on page 53). Fitting the model with a $\mathrm{N}(0, 100^2)$ prior distribution (in Bugs, `dnorm(0,.0001)`) would pull the coefficient toward zero, completely inappropriately.
- In the model on page 88 of the probability of switching wells, given distance to the nearest safe well (in 100-meter units), the logistic regression coefficient of distance is -0.62. A normal prior distribution with mean 0 and standard deviation 100 would be no problem here. If, however, distance were measured in 100-kilometer units, its coefficient would become -620, and its estimate would be strongly and inappropriately affected by the prior distribution with scale 100.

Noninformative prior distributions

To summarize the above discussion: for a prior distribution to be noninformative, its range of uncertainty should be clearly wider than the range of reasonable values of the parameters. Our starting point is regression in which the outcome y and the predictors x have variation that is of the order of magnitude of 1. Simple examples are binary variables (these are 0 or 1 by definition), subjective measurement scales (for example, 1–5, or 1–10, or -3 to $+3$), and proportions. In other cases, it makes sense to transform predictors to a more reasonable scale—for example, taking a 0–100 score and dividing by 100 so the range is from 0 to 1, or taking the logarithm of earnings or height. One of the advantages of logarithmic and logistic regressions is that these automatically put the outcomes on scales for which typical changes are 0.1, or 1, but not 10 or 100. As long as the predictors are also on a reasonable scale, one would not expect to see coefficients much higher than 10 in absolute value, and so prior distributions with scale 100 are noninformative.

At this point one might ask, why not simply set a prior distribution with mean 0 and a huge standard deviation such as 100,000 (this would be `dnorm(0,1.E-10)` in Bugs) to completely ensure noninformativeness? We do not do this for two reasons. First, Bugs can be computationally unstable when parameters have extremely wide ranges. It is safer to keep the values near 1. (This is why we typically use $\mathrm{N}(0, 1)$ and $\mathrm{Uniform}(0, 1)$ distributions for initial values, as we shall discuss.)

The second reason for avoiding extremely wide prior distributions is that we do not actually want to work with coefficients that are orders of magnitude away from zero. Part of this is for ease of interpretation (just as we transformed to `dist100` in the arsenic example on page 88). This can also be interpreted as a form of model checking—we set up a model so that parameters should not be much greater than 1 in absolute value; if they are, this indicates some aspect of the problem that we do not understand.

Data, initial values, and parameters

To return to the implementation of the radon example, we set up and call Bugs using the following sequence of commands in R:

R code
```
radon.data <- list ("n", "J", "y", "county", "x")
radon.inits <- function (){
  list (a=rnorm(J), b=rnorm(1), mu.a=rnorm(1),
        sigma.y=runif(1), sigma.a=runif(1))}
radon.parameters <- c ("a", "b", "mu.a", "sigma.y", "sigma.a")
radon.1 <- bugs (radon.data, radon.inits, radon.parameters,
  "radon.1.bug", n.chains=3, n.iter=500)
```

The first argument to the `bugs()` function lists the data—including outcomes, inputs, and indexing parameters such as n and J—that are written to a file to be read by Bugs.

The second argument to `bugs()` is a function that returns a list of the starting values for the algorithm. Within the list are random-number generators—for example, `rnorm(J)` is a vector of length J of random numbers from the $\mathrm{N}(0,1)$ distribution, and these random numbers are assigned to α to start off the Bugs iterations. In this example, we follow our usual practice and assign random numbers from normal distributions for all the parameters—except those constrained to be positive (here, σ_y and σ_α), to which we assign uniformly distributed random numbers (which, by default in R, fall in the range $[0,1]$). For more details on the random number functions, type `?rnorm` and `?runif` in R.

We generally supply initial values (using random numbers) for all the parameters in the model. (When initial values are not specified, Bugs generates them itself; however, Bugs often crashes when using its self-generated initial values.)

The third argument to the `bugs()` function is a vector of the names of the parameters that we want to save from the Bugs run. For example, the vector of α_j parameters is represented by `"a"`.

Number of sequences and number of iterations

Bugs uses an iterative algorithm that runs several Markov chains in parallel, each starting with some list of initial values and endlessly wandering through a distribution of parameter estimates. We would like to run the algorithm until the simulations from separate initial values converge to a common distribution, as Figure 16.2 illustrates. Specifying initial values using random distributions (as described above) ensures that different chains start at different points.

We assess convergence by checking whether the distributions of the different simulated chains mix; we thus need to simulate *at least 2 chains.* We also need to run the simulations "long enough," although it is generally difficult to know ahead of time how long is necessary. The `bugs()` function is set up to run for `n.iter` iterations and discard the first half of each chain (to lose the influence of the starting values). Thus, in the example presented here, Bugs ran `n.chains` = 3 sequences, each for `n.iter` = 500 iterations, with the first 250 from each sequence discarded.

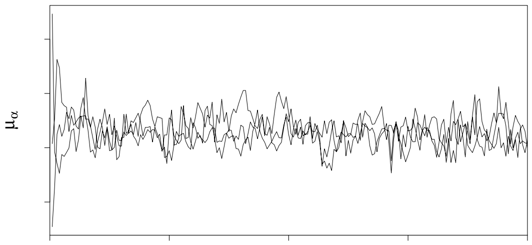

Figure 16.2 *Illustration of convergence for the parameter μ_α from the Bugs simulations for the radon model. The three chains start at different (random) points and, for the first 50 iterations or so, have not completely mixed. By 200 iterations, the chains have mixed fairly well. Inference is based on the last halves of the simulated sequences. Compare to Figure 16.3, which shows poor mixing.*

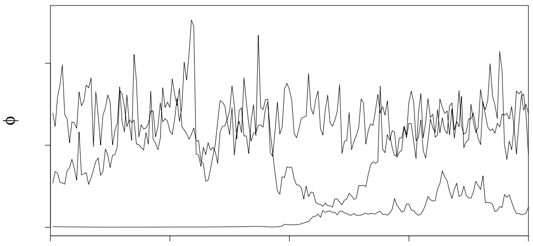

Figure 16.3 *Illustration of poor convergence for the parameter ϕ from a slowly converging Bugs simulation. Compare to Figure 16.2, which shows rapid mixing.*

We offer the following general advice on how long to run the simulations.

1. When first setting up a model, set `n.iter` to a low value such as 10 or 50—the model probably will not run correctly at first, so there is no need to waste time waiting for computations.

2. If the simulations have not reached approximate convergence (see Section 16.4), run longer—perhaps 500 or 1000 iterations—so that running Bugs takes between a few seconds and a few minutes.

3. If your Bugs run is taking a long time (more than a few minutes for the examples of the size presented in this book, or longer for larger datasets or more elaborate models), and the simulations are still far from convergence, then play around with your model to get it to converge faster; see Chapter 19 for more on this.

4. Another useful trick to speed computations, especially when in the exploratory model-fitting stage, is to work with a subset of your data—perhaps half, or a fifth, or a tenth. For example, analyze the radon data from a sample of 20 counties, rather than the full set of 85. Bugs will run faster with smaller datasets and fewer parameters.

Summary and convergence

From a Bugs run, you will see means, standard deviations, and quantiles for all the parameters that are saved. You also get, for each parameter, a convergence statistic, $\hat{R}$, and an effective number of independent simulation draws, n_{eff}. We typically monitor convergence using $\hat{R}$, which we call the *potential scale reduction factor*—for each parameter, the possible reduction in the width of its confidence interval, were the simulations to be run forever. Our usual practice is to run simulations until $\hat{R}$ is no greater than 1.1 for all parameters.

For example, here was the output after `n.iter` = 50 iterations:

R output
```
Inference for Bugs model at "radon.1.bug"
 3 chains, each with 50 iterations (first 25 discarded)
 n.sims = 75 iterations saved
          mean   sd   2.5%    25%    50%    75%  97.5% Rhat n.eff
. . .
mu.a       1.1  0.1   1.4    1.5    1.6    1.7    1.8  1.7     6
. . .
```

$\widehat{R} = 1.7$ indicates that the simulations were still far from convergence.

After `n.iter` = 500 iterations, however, we achieved approximate convergence with the radon model, with $\widehat{R} < 1.1$ for all parameters.

$\widehat{R}$ and n_{eff}

For each parameter that is saved, $\widehat{R}$ is, approximately, the square root of the variance of the mixture of all the chains, divided by the average within-chain variance. If $\widehat{R}$ is much greater than 1, the chains have not mixed well. We usually wait until $\widehat{R} \leq 1.1$ for all parameters, although, if simulations are proceeding slowly, we might work provisionally with simulations that have still not completely mixed, for example, with $\widehat{R} = 1.5$ for some parameters.

Printing a Bugs object also reports n_{eff}, the "effective number of simulation draws." If the simulation draws were independent, then n_{eff} would be the number of saved draws, which is $n_{\text{chains}} \cdot n_{\text{iter}}/2$ (dividing by 2 because our programs automatically discard the first half of the simulations from each chain). Actual Markov chain simulations tend to be autocorrelated and so the effective number of simulation draws is smaller. We usually like to have n_{eff} to be at least 100 for typical estimates and confidence intervals.

Accessing the simulations

We can use the simulations for predictions and uncertainty intervals for any functions of parameters, as with the propagation of error in classical regressions in Chapters 2 and 3. To access the simulations, we must first `attach` them in R. In the example above, we saved the Bugs output into the R object `radon.1` (see page 356), and we can load in the relevant information with the command

R code
```
attach.bugs (radon.1)
```

Each variable that was saved in the Bugs computation now lives as an R object, with its 750 simulation draws (3 chains × 500 iterations × last half of the iterations are saved = 750). Each of the scalar parameters β, μ_α, σ_y, and σ_α is represented by a vector of length 750, and the vector parameter α is saved as a 750×85 matrix. Extending this, a 10×20 matrix parameter would be saved as a $750 \times 10 \times 20$ array, and so forth.

We can access the parameters directly. For example, a 90% interval for β would be computed by

R code
```
quantile (b, c(0.05,0.95))
```

For another example, what is the probability that average radon levels (after controlling for floor-of-measurement effects) are higher in county 36 (Lac Qui Parle) than in county 26 (Hennepin)?

R code
```
mean (a[,36] > a[,26])
```

Fitted values, residuals, and other calculations

We can calculate fitted values and residuals from the multilevel model:

R code
```
y.hat <- a.multilevel[county] + b.multilevel*x
y.resid <- y - y.hat
```

and then plot them:

R code
```
plot (y.hat, y.resid)
```

Alternatively, we can add `y.hat` to the vector of parameters to save in the Bugs call, and then access the simulations of `y.hat` after the Bugs run and the call to `attach.bugs`.

We can also perform numerical calculations, such as the predictions described in Section 12.8 or anything that might be of interest. For example, what is the distribution of the difference in absolute (not log) radon level in a house with no basement in county 36 (Lac Qui Parle), compared to a house with no basement in county 26 (Hennepin)?

R code
```
lqp.radon <- rep (NA, n.sims)
hennepin.radon <- rep (NA, n.sims)
for (s in 1:n.sims){
  lqp.radon[s] <- exp (rnorm (1, a[s,36] + b[s], sigma.y[s]))
  hennepin.radon[s] <- exp (rnorm (1, a[s,26] + b[s], sigma.y[s]))
}
radon.diff <- lqp.radon - hennepin.radon
hist (radon.diff)
print (mean(radon.diff))
print (sd(radon.diff))
```

The expected difference comes to 2.0 picoCuries per liter, with a standard deviation of 4.6 and a wide range of uncertainty. Here we have compared two randomly selected houses, not the two county averages. If we wanted inference for the difference between the two county averages, we could simply take `exp(a[,36]+b) - exp(a[,26]+b)`.

We further discuss multilevel predictions in Section 16.6.

16.5 Adding individual- and group-level predictors

Classical complete-pooling and no-pooling regressions

Classical regressions and generalized linear models can be fit easily enough using R, but it can sometimes be useful also to estimate them using Bugs—often as a step toward fitting more complicated models. We illustrate with the radon example.

Complete pooling. The complete-pooling model is a simple linear regression of log radon on basement status and can be written in Bugs as

Bugs code
```
model {
  for (i in 1:n){
    y[i] ~ dnorm (y.hat[i], tau.y)
    y.hat[i] <- a + b*x[i]
  }
  a ~ dnorm (0, .0001)
  b ~ dnorm (0, .0001)
  tau.y <- pow(sigma.y, -2)
  sigma.y ~ dunif (0, 100)
}
```

No pooling. The no-pooling model can be fit in two ways: either by fitting the above regression separately to the data in each county (thus, running a loop in R for the 85 counties), or else by allowing the intercept α to vary but with a noninformative prior distribution for each α_j (so that this is still a classical regression):

Bugs code
```
model {
  for (i in 1:n){
    y[i] ~ dnorm (y.hat[i], tau.y)
    y.hat[i] <- a[county[i]] + b*x[i]
  }
  b ~ dnorm (0, .0001)
  tau.y <- pow(sigma.y, -2)
  sigma.y ~ dunif (0, 100)

  for (j in 1:J){
    a[j] ~ dnorm (0, .0001)
  }
}
```

Classical regression with multiple predictors

The Bugs model can easily include multiple predictors in `y.hat`. For example, we can add an indicator for whether the measurement was taken in winter (when windows are closed, trapping radon indoors):

Bugs code
```
    y.hat[i] <- a + b.x*x[i] + b.winter*winter[i]
```

and add an interaction:

Bugs code
```
    y.hat[i] <- a + b.x*x[i] + b.winter*winter[i] +
                b.x.winter*x[i]*winter[i]
```

In each case, we would set up these new coefficients with `dnorm(0,.0001)` prior distributions.

As the number of predictors increases, it can be simpler to set up a vector β of regression coefficients:

Bugs code
```
    y.hat[i] <- a + b[1]*x[i] + b[2]*winter[i] + b[3]*x[i]*winter[i]
```

and then assign these noninformative prior distributions:

Bugs code
```
  for (k in 1:K){
    b[k] ~ dnorm (0, .0001)
  }
```

with `K` added to the list of data in the call of `bugs()` from R.

Vector-matrix notation in Bugs. One can go further by creating a matrix of predictors in R:

R code
```
X <- cbind (x, winter, x*winter)
K <- ncol (X)
```

and then in the Bugs model, using the inner-product function:

Bugs code
```
    y.hat[i] <- a + inprod(b[],X[i,])
```

Finally, one could include the intercept in the list of β's, first including a constant term in the predictor matrix:

R code
```
ones <- rep (1, n)
X <- cbind (ones, x, winter, x*winter)
K <- ncol (X)
```

and then simplify the expression for `y.hat` in the Bugs model:

Bugs code
```
    y.hat[i] <- inprod(b[],X[i,])
```

with the coefficients being `b[1]`, . . . , `b[4]`.

In a varying-intercept model, it can be convenient to keep the intercept α separate from the other coefficients β. However, in model with a varying intercept and several varying slopes, it can make sense to use the unified notation including all of them in a matrix B, as we discuss in Sections 17.1 and 17.2.

Multilevel model with a group-level predictor

Here is the Bugs code for model (12.15) on page 266, which includes a group-level predictor, u_j (the county-level uranium measure in the radon example):

Bugs code
```
model {
  for (i in 1:n){
    y[i] ~ dnorm (y.hat[i], tau.y)
    y.hat[i] <- a[county[i]] + b*x[i]
  }
  b ~ dnorm (0, .0001)
  tau.y <- pow(sigma.y, -2)
  sigma.y ~ dunif (0, 100)

  for (j in 1:J){
    a[j] ~ dnorm (a.hat[j], tau.a)
    a.hat[j] <- g.0 + g.1*u[j]
  }
  g.0 ~ dnorm (0, .0001)
  g.1 ~ dnorm (0, .0001)
  tau.a <- pow(sigma.a, -2)
  sigma.a ~ dunif (0, 100)
}
```

16.6 Predictions for new observations and new groups

We can simulate predictions in two ways: directly in Bugs, by adding additional units or groups to the model, or in R, by drawing simulations based on the model fit to existing data. We demonstrate both approaches for the radon example, in each case demonstrating how to forecast for new houses in existing counties and for new houses in new counties.

Predicting a new unit in an existing group using Bugs

Bugs automatically makes predictions for modeled data with `NA` values. Thus, to predict the log radon level for a new house in county 26 with no basement, we merely need to extend the dataset by one point. We first save the current dataset to an external file and then extend it:

R code
```
save ("n", "y", "county", "x", file="radon.data")
n <- n + 1
y <- c (y, NA)
county <- c (county, 26)
x <- c (x, 1)
```

For convenience, we then add a line at the end of the Bugs model to flag the predicted measurement:

Bugs code
```
  y.tilde <- y[n]
```

and rename this file as a new model, `radon2a.bug`. We now fit this Bugs model, after alerting it to save the inferences for the additional data point:

R code
```
radon.parameters <- c (radon.parameters, "y.tilde")
radon.2a <- bugs (radon.data, radon.inits, radon.parameters,
  "radon.2a.bug", n.chains=3, n.iter=500)
```

The prediction, $\tilde{y}$, now appears if `radon.2a` is printed or plotted, or we can access it directly; for example,

R code
```
attach.bugs (radon.2a)
quantile (exp (y.tilde), c(.25,.75))
```

gives a 50% confidence interval for the (unlogged) radon level in this new house.

We can similarly make predictions for any number of new houses by adding additional `NA`'s to the end of the data vector. It is necessary to specify the predictors for these new houses; if you set `county` or `x` to `NA` for any of the data, the Bugs model will not run. Bugs allows missing data in modeled, but not unmodeled, data (a distinction we discuss further in Section 16.8).

Predicting a new unit in a new group using Bugs

The same approach can be used to make predictions for houses in unmeasured counties (ignoring for the moment that this particular survey included all the counties in Minnesota, or else considering an unmeasured county in a similar neighboring state). We merely need to extend the number of counties in the hypothetical dataset and specify the group-level predictor u_j (in this case, the county uranium measurement) for this new county. For simplicity we here consider a hypothetical new county with uranium measurement equal to the average of the 85 existing counties:

R code
```
u.tilde <- mean (u)
```

We now load back the original data, save everything, and extend to a new house in a new county:

R code
```
load ("radon.data")
save ("n", "y", "county", "x", "J", "u", file="radon.data")
n <- n + 1
y <- c (y, NA)
county <- c (county, J+1)
x <- c (x, 1)
J <- J + 1
u <- c (u, u.tilde)
```

We can then run the model as before, again using the simulations of $\tilde{y}$ to summarize the uncertainty about the radon level in this new house, this time in a new county.

Prediction using R

The other approach to prediction is to use R to simulate from the distribution of the new data, conditional on the estimated parameters from the model. Section 12.8 laid out how to do this using estimates from `lmer()`; here we do the same thing using Bugs output. The only difference from before is that instead of working with functions such as `coef()` and `sim()` applied to objects created from `lmer()`, we work directly with simulated parameters after attaching Bugs fits.

For a new house with no basement in county 26:

R code

```
attach.bugs (radon.2)
y.tilde <- rnorm (n.sims, a[,26] + b*1, sigma.y)
```

This creates a vector of n_{sims} simulations of $\tilde{y}$. The only tricky thing here is that we need to use matrix notation for `a` (its 26^{th} column contains the simulations for a_{26}), but we can write `b` and `sigma.y` directly, since as scalar parameters these are saved as vectors of simulations. The simplest way to understand this is to perform the calculations on your own computer, running Bugs for just 10 iterations so that the saved objects are small and can be understood by simply typing `a`, `b`, and so forth in the R console.

Continuing, prediction for a new house with no basement in a new county with uranium level $\tilde{u}$ requires simulation first of the new $\tilde{\alpha}_j$, then of the radon measurement in the house within this county:

R code

```
a.tilde <- rnorm (n.sims, g.0 + g.1*u.tilde, sigma.a)
y.tilde <- rnorm (n.sims, a.tilde + b*1, sigma.y)
```

16.7 Fake-data simulation

As discussed in Sections 8.1–8.2, a good way to understand a model-fitting procedure is by simulating and then fitting a model to fake data:

1. Specify a reasonable "true" value for each of the parameters in the model. Label the vector of specified parameters as θ^{true}; these values should be reasonable and be consistent with the model.
2. Simulate a fake dataset y^{fake} using the model itself along with the assumed θ^{true}.
3. Fit the model to the fake data, and check that the inferences for the parameters θ are consistent with the "true" θ^{true}.

We illustrated this process for classical regression with numerical checks in Section 8.1 and graphical checks in Section 8.2. Here we demonstrate fake-data checking for the varying-intercept radon model (12.15) on page 266 with floor and uranium as individual- and county-level predictors, respectively.

Specifying "true" parameter values

For a classical regression, one must simply specify the coefficients β and residual standard deviation σ to begin a fake-data simulation. Multilevel modeling is more complicated: one must first specify the hyperparameters, then simulate the modeled parameters from the group-level distributions.

Specifying the unmodeled parameters. We start by specifying β^{true}, γ_0^{true}, γ_1^{true}, σ_y^{true}, and $\sigma_\alpha^{\text{true}}$: these are the parameters that do not vary by group, and they get the simulation started.

We do *not* want to choose round numbers such as 0 or 1 for the parameters, since these can mask potential programming errors. For example, a variance is specified as σ rather than σ^2 would not show up as an inconsistency if σ were set to 1.

There are two natural ways to set the parameters. The first approach is just to pick values that seem reasonable; for example, $\beta^{\text{true}} = -0.5$, $\gamma_0^{\text{true}} = 0.5$, $\gamma_1^{\text{true}} = 2.0$, $\sigma_y^{\text{true}} = 3.0$, $\sigma_\alpha^{\text{true}} = 1.5$. (Variation among groups is usually less than variation among individuals within a group, and so it makes sense to set $\sigma_\alpha^{\text{true}}$ to a smaller value than σ_y^{true}. The model is on the log scale, so it would not make sense to choose numbers such as 500 or -400 that are high in absolute value; see the discussion on page 355 on the scaling of prior distributions.)

The other way to choose the parameters is to use estimates from a fitted model. For this example, we could set $\beta^{\text{true}} = -0.7$, $\gamma_0^{\text{true}} = 1.5$, $\gamma_1^{\text{true}} = 0.7$, $\sigma_y^{\text{true}} = 0.8$, $\sigma_\alpha^{\text{true}} = 0.2$, which are the median estimates of the parameters from model (12.15) as fitted in Bugs (see code at the very end of Section 16.5). Supposing we have labeled this `bugs` object as `radon.2`, then we can set the "true" values in R:

R code
```
attach.bugs (radon.2)
b.true <- median (b)
g.0.true <- median (g.0)
g.1.true <- median (g.1)
sigma.y.true <- median (sigma.y)
sigma.a.true <- median (sigma.a)
```

Simulating the varying coefficients. We now simulate the α_j's from the group-level model given the "true" parameters using a loop:[4]

R code
```
a.true <- rep (NA, J)
for (j in 1:J){
  a.true[j] <- rnorm (1, g.0.true + g.1.true*u[j], sigma.a.true)
}
```

Simulating fake data

We can now simulate the dataset y^{fake}:[5]

R code
```
y.fake <- rep (NA, n)
for (i in 1:n){
  y.fake[i] <- rnorm (1, a.true[county[i]] + b.true*x[i], sigma.y.true)
}
```

Inference and comparison to "true" values

We can now fit the model in Bugs using the fake data. The fitting procedure is the same except that we must pass `y.fake` rather than `y` to Bugs, which we can do by explicitly specifying the data to be passed:[6]

[4] Or, more compactly but perhaps less clearly, in vector form:
`a.true <- rnorm (J, g.0.true + g.1.true*u, sigma.a.true).`

[5] Again, an experienced R programmer would use the vector form:
`y.fake <- rnorm (n, a.true[county] + b.true*x, sigma.y.true).`

[6] Alternatively, we could use the existing data object after saving the real data and renaming the fake data:
`y.save <- y; y <- y.fake.`

R code
```
radon.data <- list (n=n, J=J, y=y.fake, county=county, x=x, u=u)
```

We can then specify the rest of the inputs, run Bugs, and save it into a new R object:

R code
```
radon.inits <- function (){
  list (a=rnorm(J), b=rnorm(1), g.0=rnorm(1), g.1=rnorm(1),
        sigma.y=runif(1), sigma.a=runif(1))}
radon.parameters <- c ("a", "b", "g.0", "g.1", "sigma.y", "sigma.a")
radon.2.fake <- bugs (radon.data, radon.inits, radon.parameters,
  "radon.2.bug", n.chains=3, n.iter=500)
```

We are now ready to compare the inferences to the true parameter values. To start, we can display the fitted model (`print(radon.2.fake)`) and compare inferences to the true values. Approximately half the 50% intervals and approximately 95% of the 95% intervals should contain the true values; about half of the median estimates should be above the true parameter values and about half should be below. In our example, the output looks like:

R output
```
             mean   sd   2.5%    25%    50%    75%  97.5% Rhat n.eff
a[1]          1.0  0.2    0.7    0.9    1.0    1.1    1.4  1.0   280
a[2]          0.9  0.1    0.7    0.8    0.9    0.9    1.0  1.0   190
. . .
a[85]         1.8  0.2    1.4    1.7    1.8    1.9    2.2  1.0   470
b            -0.6  0.1   -0.7   -0.6   -0.6   -0.6   -0.5  1.0   750
g.0           1.5  0.0    1.4    1.4    1.5    1.5    1.5  1.0   140
g.1           0.8  0.1    0.6    0.7    0.8    0.9    1.0  1.0    68
sigma.y       0.8  0.0    0.7    0.7    0.8    0.8    0.8  1.0   240
sigma.a       0.2  0.0    0.1    0.2    0.2    0.2    0.3  1.1    36
```

In the particular simulation we ran, the 85 values of α^{true} were 1.2, 0.9, …, 1.8 (as we can see by simply typing `a.true` in the R console), and the "true" values of the other parameters, are $\beta^{\text{true}} = -0.7$, $\gamma_0^{\text{true}} = 1.5$, $\gamma_1^{\text{true}} = 0.7$, $\sigma_y^{\text{true}} = 0.8$, $\sigma_\alpha^{\text{true}} = 0.2$, as noted earlier. About half of these fall within the 50% intervals, as predicted.

Checking coverage of 50% intervals

For a more formal comparison, we can measure the coverage of the intervals. For example, to check the coverage of the 50% interval for α_1:

R code
```
attach.bugs (radon.2.fake)
a.true[1] > quantile (a[,1], .25) & a.true[1] < quantile (a[,1], .75)
```

which, for our particular simulation, yields the value `FALSE`. We can write a loop to check the coverage for all 85 α_j's:[7]

R code
```
cover.50 <- rep (NA, J)
for (j in 1:J){
  cover.50[j] <- a.true[j] > quantile (a, .25) &
                 a.true[j] < quantile (a, .75)
}
mean (cover.50)
```

which comes to 0.51 in our example—well within the margin of error for a random simulation.

Other numerical and graphical checks are also possible, following the principles of Chapter 8.

[7] Again, we could write the calculation more compactly in vectorized form as `cover.50 <- a.true > quantile (a, .25) & a.true < quantile (a, .75)`.

Category	List of objects
Modeled data	`y`
Unmodeled data	`n`, `J`, `county`, `x`, `u`
Modeled parameters	`a`
Unmodeled parameters	`b`, `g.0`, `g.1`, `sigma.y`, `sigma.a`
Derived quantities	`y.hat`, `tau.y`, `a.hat`, `tau.a`
Looping indexes	`i`, `j`

Figure 16.4 *Classes of objects in the Bugs model and R code of Section 16.8.* Data *are specified in the list of data sent to Bugs,* parameters *are nondata objects that are given distributions ("∼" statements in Bugs),* derived quantities *are defined deterministically ("<-" statements in Bugs), and* looping indexes *are defined in* `for` *loops in the Bugs model.*

16.8 The principles of modeling in Bugs

Data, parameters, and derived quantities

Every object in a Bugs model is data, parameter, or derived quantity. Data are specified in the `bugs()` call, parameters are modeled, and derived quantities are given assignments. Some but not all data are modeled.

We illustrate with the varying-intercept Bugs model on page 361, stored in the file `radon3.bug`:

Bugs code
```
model {
  for (i in 1:n){
    y[i] ~ dnorm (y.hat[i], tau.y)
    y.hat[i] <- a[county[i]] + b*x[i]
  }
  b ~ dnorm (0, .0001)
  tau.y <- pow(sigma.y, -2)
  sigma.y ~ dunif (0, 100)

  for (j in 1:J){
    a[j] ~ dnorm (a.hat[j], tau.a)
    a.hat[j] <- g.0 + g.1*u[j]
  }
  g.0 ~ dnorm (0, .0001)
  g.1 ~ dnorm (0, .0001)
  tau.a <- pow(sigma.a, -2)
  sigma.a ~ dunif (0, 100)
}
```

as called from R as follows:

R code
```
radon.data <- list ("n", "J", "y", "county", "x", "u")
radon.inits <- function (){
  list (a=rnorm(J), b=rnorm(1), g.0=rnorm(1), g.1=rnorm(1),
        sigma.y=runif(1), sigma.a=runif(1))}
radon.parameters <- c ("a", "b", "sigma.y", "sigma.a")
radon.3 <- bugs (radon.data, radon.inits, radon.parameters,
  "radon.3.bug", n.chains=3, n.iter=500)
```

This model has the following classes of objects, which we summarize in Figure 16.4.

- The *data* are the objects that are specified by the `data` input to the `bugs()` call:

  - *Modeled data*: These are the data objects that are assigned probability distributions (that is, they are to the left of a "∼" in a line of Bugs code). In our example, the only modeled data are the components of `y`.
  - *Unmodeled data*: These are the data that are not assigned any distribution in the Bugs code. The unmodeled data objects in our example are `n`, `J`, `county`, `x`, `u`.

- Next come the *parameters*, which are assigned probability distributions (that is, they are to the left of a "∼" in a line of Bugs code) but are not specified as data in the `bugs()` call:
  - *Modeled parameters*: We label parameters as "modeled" if they are assigned informative prior distributions that depend on hyperparameters (which typically are themselves estimated from the data). The only modeled parameters in our example are the elements of `a`, whose distribution depends on `a.hat` and `tau.a`.
  - *Unmodeled parameters*: These are the parameters with noninformative prior distributions (typically `dnorm(0,.0001)` or `dunif(0,100)`); in our example, these are `b`, `g.0`, `g.1`, `sigma.y`, `sigma.a`. Strictly speaking, what we call "unmodeled parameters" are actually modeled with wide, "noninformative" prior distributions. What is important here is that their distributions in the Bugs model are specified not based on other parameters in the model but rather based on constants such as .0001.
- *Derived quantities*: These are objects that are defined deterministically (that is, with "`<-`" in the Bugs code); in our example: `y.hat`, `tau.y`, `a.hat`, `tau.a`.
- *Looping indexes*: These are integers (in our example, `i` and `j`) that are defined in `for` loops in the Bugs model.

To figure out whether an object is a parameter or a derived quantity, it can be helpful to scan down the code to see how it is defined. For example, reading the model on page 361 from the top, it is not clear at first whether `tau.y` is a parameter or a derived quantity. (We know it is not data since it is not specified in the `data` list supplied in the call to `bugs()`.) But reading through the model, we see the line `tau.y <- ...`, so we know it is a derived quantity.

Missing data

Modeled data can have elements with NA values, in which case these elements are implicitly treated as parameters by the Bugs model—that is, they are estimated stochastically along with the other uncertain quantities in the model. Bugs will not accept *unmodeled* data (for example, the regression predictors `x` and `u` in our example) with NA values, because these objects in the model have no distributions specified and thus cannot be estimated stochastically. If predictors have missing data, they must either be imputed before being entered into the Bugs model, or they must themselves be modeled.

Changing what is included in the data

Any combination of data, parameters, or derived quantities can be saved as parameters in the `bugs()` call. But only parameters and missing data, not observed data and not derived quantities, can be given initial values.

Conversely, the status of the objects in a given Bugs model can change by changing the corresponding call in R. For example, we can call the model without specifying `y`:

R code
```
radon.data <- list ("n", "J", "county", "x", "u")
```

and the components of `y` become modeled parameters—they are specified by distributions in the Bugs model and not included as data. Bugs then draws from the "prior predictive distribution"—the model unconditional on y. Alternatively, if `b, g.0, g.1, sigma.y, sigma.a` are included in the data list but `y` is not, Bugs will perform "forward simulation" for `y` given these specified parameters.

We cannot remove the other objects—`n`, `J`, `county`, `x`, `u`—from the data list, because these are not specified in the Bugs model. For example, there is no line in the model of the form, `n` $\sim$ `...` If we did want to remove any of these data objects, we would need to include them in the model, either as parameters (defined by $\sim$) or derived quantities (defined by `<-`).

We can, however, specify the values of parameters in the model. For example, suppose we wanted to fit the model with the variance parameters set to known values, for example, $\sigma_y = 0.7$ and $\sigma_\alpha = 0.4$. We can simply define them in the R code and include them in the data:

R code
```
sigma.y <- .7
sigma.alpha <- .4
radon.data <- list("n","J","y","county","x","u","sigma.y","sigma.alpha")
```

and remove them from the `inits()` function.

Each object can be defined at most once

Every object is modeled or defined at most once in a Bugs model (except that certain transformations can be done by declaring a variable twice). For example, the example we have been discussing includes the lines

Bugs code
```
  for (i in 1:n){
    y[i] ~ dnorm (y.hat[i], tau.y)
    y.hat[i] <- a[county[i]] + b*x[i]
  }
```

It might seem more compact to express this as

Bugs code
```
  for (i in 1:n){
    y[i] ~ dnorm (y.hat, tau.y)
    y.hat <- a[county[i]] + b*x[i]
  }
```

thus "overwriting" the intermediate quantity `y.hat` at each step of the loop. Such code would work in R but is not acceptable in Bugs. The reason is that lines of Bugs code are *specifications* of a model, not *instructions* to be executed. In particular, these lines define the single variable `y.hat` multiple times, which is not allowed in Bugs.

For another example, we have been coding the inverse-variance in terms of the standard-deviation parameter:

Bugs code
```
  tau.y <- pow(sigma.y, -2)
  sigma.y ~ dunif (0, 100)
```

It might seem natural to write this transformation as

```
sigma.y <- sqrt(1/tau.y)
sigma.y ~ dunif (0, 100)
```
Bugs code

but this will not work: in this model, `sigma.y` is defined twice and `tau.y` is defined not at all. We must use the earlier formulation in which `tau.y` (which is specified in the data model) is defined in terms of `sigma.y`, which is then given its own prior distribution.

16.9 Practical issues of implementation

In almost any application, a good starting point is to run simple classical models in R (for example, complete-pooling and no-pooling regressions) and then replicate them in Bugs, checking that the estimates and standard errors are approximately unchanged. Then gradually complexify the model, adding multilevel structures, group-level predictors, varying slopes, non-nested structures, and so forth, as appropriate.

We suggest a simple start for both statistical and computational reasons. Even in classical regression, it is a good idea to include the most important predictors first and then see what happens when further predictors and interactions are added. Multilevel models can be even more difficult to understand, and so it makes sense to build up gradually. In addition, it is usually a mistake in Bugs to program a complicated model all at once; it typically will not run, and then you have to go back to simpler models anyway until you can get the program working.

If a model does not run, you can use the `debug=TRUE` option in the call to `bugs()`. Then the Bugs window will stay open and you might be able to figure out what's going wrong, as we discuss in Section 19.1.

How many chains and how long to run?

We usually run a small number of chains such as 3. This is governed by the `n.chains` argument of the `bugs()` function.

In deciding how long to run the simulations, we balance the goals of speed and convergence. We start by running Bugs for only a few iterations in debug mode until we can get our script to run without crashing. Once it works, we will do a fairly short run—for example, `n.iter` = 100 or 500. At this point:

- If approximate convergence has been reached ($\widehat{R} < 1.1$ for all parameters), we stop.
- If the sequences seem close to convergence (for example, $\widehat{R} < 1.5$ for all parameters), then we repeat, running longer (for example, 1000 or 2000 iterations).
- If our Bugs run takes more than a few minutes, and the sequences are still far from convergence, we step back and consider our options, which include:
  - altering the Bugs model to run more efficiently (see the tips in Chapter 17),
  - fitting the Bugs model to a sample of the data (see Section 19.2),
  - fitting a simpler model.

In some settings with complicated models, it may be necessary to run Bugs for a huge number of iterations, but in the model-building stage, we generally recommend *against* the "brute force" approach of simply running for 50,000 or 100,000 iterations. Even if this tactic yields convergence, it is typically not a good long-run solution, since it ensures long waits for fitting the inevitable alterations of the model (for example, from adding new predictors).

Initial values

The `bugs()` function takes an `inits` argument, which is a function that must be written for creating starting points. It is not too important exactly what these starting points are, as long as they are dispersed (so the different chains start at different points; see Figure 16.2 on page 357) and reasonable. If parameters get started at values such as 10^{-4} or 10^6, Bugs can drift out of range and crash.

It is convenient to specify initial values as a function using random numbers, so that running it for several chains will automatically give different starting points. Typically we start parameters from random $\mathrm{N}(0,1)$ distributions, unless they are restricted to be positive, in which case we typically use random numbers that are uniformly distributed between 0 and 1 (the $\mathrm{Uniform}(0,1)$ distribution). Vector and matrix parameters must be set up as vectors and matrices; for example, if a is a scalar parameter, b is a vector of length J, C is a $J \times K$ matrix, and σ is a scalar parameter constrained to be positive:

R code
```
inits <- function() {list (a=rnorm(1), b=rnorm(J),
  C=array(rnorm(J*K), c(J,K)), sigma=runif(1))}
```

Here we have used the defaults of the `rnorm()` and `runif()` functions. If instead, for example, we want to use $\mathrm{N}(0,2^2)$ and $\mathrm{Uniform}(0.1,10)$ distributions, we can write

R code
```
inits <- function() {list (a=rnorm(1,0,2), b=rnorm(J,0,2),
  C=array(rnorm(J*K,0,2), c(J,K)), sigma=runif(1,.1,10))}
```

16.10 Open-ended modeling in Bugs

This book focuses on the most standard models, beginning with linear regression and then adding various complications one step at a time. From this perspective, Bugs is useful because it accounts for uncertainty in all parameters when fitting multilevel models. However, a quite different advantage of Bugs is its modular structure, which allows us to fit models of nearly arbitrary complexity.

We demonstrate here by considering a hypothetical study of a new teaching method applied in J different classrooms containing a total of n students. Our data for this example will be the treatment indicator T (defined at the school level) and, for each student, a pre-treatment assessment, x (on a 1–10 scale, say) and a post-treatment test score, y (on a 0–100 scale).

The minimal model for such data is a hierarchical regression with varying intercepts for schools:

Bugs code
```
model {
  for (i in 1:n){
    y[i] ~ dnorm (y.hat[i], tau.y)
    y.hat[i] <- a[school[i]] + b*x[i]
  }
  b ~ dnorm (0, .0001)
  tau.y <- pow(sigma.y, -2)
  sigma.y ~ dunif (0, 100)

  for (j in 1:J){
    a[j] ~ dnorm (a.hat[j], tau.a)
    a.hat[j] <- g.0 + g.1*T[j]
  }
  g.0 ~ dnorm (0, .0001)
```

```
    g.1 ~ dnorm (0, .0001)
    tau.a <- pow(sigma.a, -2)
    sigma.a ~ dunif (0, 100)
  }
```

The natural extension here is to allow the coefficient β for the pre-treatment assessment to vary by group. This is a varying-intercept, varying-slope model, the implementation of which we shall discuss in Sections 17.1–17.2. Here, however, we shall consider some less standard extensions, to demonstrate the flexibility of Bugs.

Nonlinear and nonadditive models

The relation between inputs and regression predictor need not be linear. From classical regression we are already familiar with the inclusion of interactions and transformed inputs as additional predictors, for example, altering the expression for $\hat{y}$:

```
y.hat[i] <- a[school[i]] + b[1]*x[i] + b[2]*pow(x[i],2) +
  b[3]*T[school[i]]*x[i]
```
Bugs code

Another option is to define transformed variables in R and then include them as predictors in the Bugs model; for example, for the squared term

```
x.sq <- x^2
```
R code

and then, in the Bugs model:

```
y.hat[i] <- a[school[i]] + b[1]*x[i] + b[2]*x.sq[i]
```
Bugs code

Nonlinear functions of data and parameters. More interesting are models that cannot simply be expressed as regressions. For example, suppose we wanted to fit a model with diminishing returns for the pre-treatment assessment, such as $y = \alpha - \beta e^{-\gamma x}$. We can simultaneously estimate the linear parameters α, β and the nonlinear γ in a Bugs model:

```
y.hat[i] <- a[school[i]] + b*exp(-g*x[i])
```
Bugs code

and also add a noninformative prior distribution for γ. The parameters β and γ could also be allowed to vary by group. More complicated expressions are also possible, for example,

```
y.hat[i] <- a + b[1]*exp(-g[1]*x1[i]) + b[2]*exp(-g[2]*x2[i])
```
Bugs code

or

```
y.hat[i] <- (a + b*x1[i])/(1 + g*x2[i])
```
Bugs code

or whatever. The point here is not to try an endless variety of models but to be able to fit models that might be suggested by theoretical considerations, and to have the flexibility to alter functional forms as appropriate.

Unequal variances

Perhaps the data-level variance should be different for students in the treated and control groups. (The variance in the treatment group could be higher, for example, if the treatment worked very well on some students and poorly on others. Or, in the other direction, the treated students could show lower variance if the effect of the treatment is to pull all students up to a common level of expertise.)

Different variances for treated and control units. We can allow for either of these possibilities by changing the data distribution to

Bugs code
```
    y[i] ~ dnorm (y.hat[i], tau.y[T[school[i]+1]])
```

(Adding 1 to T allows the index to take on the values 1 and 2.) The specification of `tau.y` in the model is then expanded to

Bugs code
```
  for (k in 1:2){
    tau.y[k] <- pow(sigma.y[k], -2)
    sigma.y[k] ~ dunif (0, 100)
  }
```

Different variance within each group. Similarly, we can simply let the data variance vary by school by changing the data model to,

Bugs code
```
    y[i] ~ dnorm (y.hat[i], tau.y[school[i]])
```

and then specifying the distribution of `tau.y` within the `for (j in 1:J) loop`:

Bugs code
```
    tau.y[j] <- pow(sigma.y[j], -2)
    sigma.y[j] ~ dunif (0, 100)
```

Modeling the varying variances. Once we have a parameter that varies by group, it makes sense to model it, for example using a lognormal distribution:

Bugs code
```
    tau.y[j] <- pow(sigma.y[j], -2)
    sigma.y[j] ~ dlnorm (mu.lsigma.y, tau.lsigma.y)
```

This extra stage of modeling allows us to better adjust for unequal variances when the sample size within groups is small (so that within-group variances cannot be precisely estimated individually). We also must specify noninformative prior distributions for these hyperparameters, outside the `for (j in 1:j)` loop:

Bugs code
```
  mu.lsigma.y ~ dnorm (0, .0001)
  tau.lsigma.y <- pow(sigma.lsigma.y, -2)
  sigma.lsigma.y ~ dunif (0, 100)
```

Variances that also differ systematically between treatment and control. We can extend the hierarchical model for the variances to include treatment as a group-level predictor for the variance model:

Bugs code
```
    tau.y[j] <- pow(sigma.y[j], -2)
    sigma.y[j] ~ dlnorm (log.sigma.hat[j], sigma.log.sigma.y)
    log.sigma.hat[j] <- d.0 + d.1*T[j]
```

and again specifying noninformative prior distributions outside the loop.

Other distributional forms

There is no need to restrict ourselves to the normal distribution. For example, the t distribution allows for occasional extreme values. Here is how it can be set up in Bugs with degrees of freedom ν_y estimated from the data:

Bugs code
```
    y[i] ~ dt (y.hat[i], tau.y, nu.y)
```

and then, outside the loop, we must put in a prior distribution for ν_y. Bugs restricts this degrees-of-freedom parameter to be at least 2, so it is convenient to assign a uniform distribution on its inverse:

Bugs code
```
  nu.y <- 1/nu.inv.y
  nu.inv.y ~ dunif (0, .5)
```

The t model could similarly be applied to the group-level model for the α_j's as well.

16.11 Bibliographic note

Details on Bugs are in Appendix C. For fitting models in Bugs, the two volumes of examples in the online help are a good starting point, and the textbooks by Lancaster (2004) and Congdon (2001, 2003) are also helpful because they use Bugs for all their examples. Gelman and Rubin (1992) and Brooks and Gelman (1998) discuss the use of multiple chains to monitor the convergence of iterative simulations. Kass et al. (1998) present a lively discussion of practical issues in implementing iterative simulation.

16.12 Exercises

1. Elements of a Bugs model: list the elements of the model on page 370 by category: modeled data, unmodeled data, modeled parameters, unmodeled parameters, derived quantities, and looping indexes (as in Figure 16.4).
2. Find all the errors in the following Bugs model:

Bugs code

```
model {
  for (i in 1:n){
    y[i] ~ dnorm (a[state[i]] + theta*treat[i] + b*hispanic, tau.y)
  }
    theta ~ dnorm (0, .0001)
    b ~ dnorm (0, 1000)
  for (j in 1:J){
    a[j] ~ rnorm (mu.a, tau.a^2)
  }
  mu.a ~ dnorm (0, .0001)
  tau.a <- pow (sigma.a, -2)
  sigma.a ~ dunif (0, 100)
  tau.y <- pow (sigma.y, -2)
  sigma.y <- dunif (0, 100)
}
```

3. Using the data in folder `cd4` regarding CD4 percentages for young children with HIV, we shall revisit Exercise 12.2.
   (a) Use Bugs to fit the model in Exercise 12.2(a). Interpret the results.
   (b) Use Bugs to fit the model in Exercise 12.2(b). Interpret the results.
   (c) How do the results from these models compare to the fits from `lmer()`?
   (d) Summarize the results graphically as in Section 16.3.
4. Repeat the predictions described in Exercise 12.3 using the output from the Bugs fits from Exercise 16.3 instead.
5. Scaling of coefficients: again using the data in folder `cd4`, fit the model you formulated in Exercise 12.2(b), just as you did in Exercise 16.3(b). What happens if you rescale time so that it is in units of days rather than years? How does this influence your prior distributions and starting values?
6. Convergence of iterative simulation: return to the beauty and teaching evaluations example introduced in Exercise 3.5 and revisited in Exercises 12.6 and 13.1.
   (a) Write a varying-intercept model for these data with no group-level predictors. Fit this model using Bugs but allow for only 10 iterations. What do the $\widehat{R}$ values look like?

(b) Now fit the model again allowing for enough iterations to achieve convergence. How many iterations were required? Interpret the results from this model.

(c) Write a varying-intercept model that you would like to fit to these data that includes three group-level predictors. Fit this model using Bugs. How many iterations were required for convergence for this model? Interpret the results of the model.

(d) Create fake-data simulations to check the fit of the models in (b) and (c).

7. This exercise will use the data you found for Exercise 4.7. This time, rather than repeating the same analysis across each year, or country (or whatever group the data vary across), fit a multilevel model using Bugs instead. Compare the results to those obtained in your earlier analysis.

8. Impact of the prior distribution: you will use Bugs to fit several versions of the varying-intercept model to the radon data using floor as a house-level predictor and uranium as a county-level predictor.

(a) How do the inferences change if you assign normal prior distributions with mean 5 and standard deviation 1000 to the coefficients for floor and uranium.

(b) How do the inferences change if you switch to normal prior distributions with mean 0 and standard deviation 0.1?

(c) Now try normal prior distributions with mean 5 and standard deviation 1.

(d) Now try t prior distributions with mean 5, standard deviation 1, and 4 degrees of freedom.

(e) Now try Uniform(-100,100) prior distributions, then Uniform(-1,1) prior distributions.

(f) Discuss the impact of the prior distributions on the inferences.