APPENDIX C

Example of computation in R and Bugs

We illustrate some of the practical issues of simulation by fitting a single example—the hierarchical normal model for the eight educational testing experiments described in Section 5.5. After some background in Section C.1, we show in Section C.2 how to fit the model using the Bayesian inference package `Bugs`, operating from within the general statistical package `R`. We proceed from data input, through statistical inference, to posterior predictive simulation and graphics. Section C.3 describes some alternative ways of parameterizing the model in Bugs. Section C.4 presents several different ways of programming the model directly in R. These algorithms require programming efforts that are unnecessary for the Bugs user but are useful knowledge for programming more advanced models for which Bugs might not work. We conclude in Section C.5 with some comments on practical issues of programming and debugging. It may also be helpful to read the computational tips at the end of Section 10.3.

C.1 Getting started with R and Bugs

Follow the instructions at `www.stat.columbia.edu/~gelman/bugsR/` to install the latest versions of `R` and `Bugs` (both of which are free of charge at the time of the writing of this book), as well as to configure the two programs appropriately and set up a working directory for R. Also download the functions at that website for running Bugs from R and postprocessing regression output. Once you have followed the instructions, your working directory will automatically be set every time you run R, and the functions for running Bugs will be automatically be loaded. You can call Bugs within R as illustrated in the example on the webpage.

R (the open-source version of S and S-Plus) is a general-purpose statistical package that is fully programmable and also has available a large range of statistical tools, including flexible graphics, simulation from probability distributions, numerical optimization, and automatic fitting of many standard probability models including linear regression and generalized linear models. For Bayesian computation, one can directly program Gibbs sampler and Metropolis algorithms, as we illustrate in Section C.4. Computationally intensive tasks can be programmed in Fortran or C and linked from R.

Bugs is a high-level language in which the user specifies a model and starting values, and then a Markov chain simulation is automatically implemented for the resulting posterior distribution. It is possible to set up models and run them entirely within Bugs, but in practice it is almost always necessary to process data before entering them into a model, and to process the inferences after the model is fitted, and so we run Bugs by calling it from R, using the `bugs()` function in R, as we illustrate in Section C.2. As of the writing of this book, the current version of Bugs is called `WinBUGS` and must be run in the environment of the Microsoft Windows operating system. When we mention the Bugs package, we are referring to the latest version of WinBUGS.

R and Bugs have online help, and further information is available at the webpages `www.r-project.org` and `www.mrc-bsu.cam.ac.uk/bugs/`. We anticipate continuing improvements in both packages in the years after this book is released (and we will update our webpage accordingly), but the general computational strategies presented here should remain relevant.

When working in Bugs and R, it is helpful to set up the computer to simultaneously display three windows: an R console and graphics window, and text editors with the Bugs model file and the R script. Rather than typing commands directly into R, we prefer to enter them into the text editor and then use cut-and-paste to transfer them to the R console. Using the text editor is convenient because it allows more flexibility in writing functions and loops.

C.2 Fitting a hierarchical model in Bugs

In this section, we describe all the steps by which we would use Bugs to fit the hierarchical normal model to the SAT coaching data in Section 5.5. These steps include writing the model in Bugs and using R to set up the data and starting values, call Bugs, create predictive simulations, and graph the results. Section C.3 gives some alternative parameterizations of the model in Bugs.

Bugs model file

The hierarchical model can be written in Bugs in the following form, which we save in the file `schools.txt` in our R working directory. (The file must be given a `.txt` extension.)

```
model {
  for (j in 1:J){
    y[j] ~ dnorm (theta[j], tau.y[j])
    theta[j] ~ dnorm (mu.theta, tau.theta)
    tau.y[j] <- pow(sigma.y[j], -2)
  }
  mu.theta ~ dnorm (0, 1.0E-6)
  tau.theta <- pow(sigma.theta, -2)
  sigma.theta ~ dunif (0, 1000)
}
```

This model specification looks similar to how it would be written in this book, with two differences. First, in Bugs, the normal distribution is parameterized by its precision (1/variance), rather than its standard deviation. When working in Bugs, we use the notation τ for precisions and σ for standard deviations (departing from our notation in the book, where we use σ and τ for data and prior standard deviations, respectively).

The second difference from our model in Chapter 5 is that Bugs requires proper prior distributions. Hence, we express noninformative prior information by proper distributions with large uncertainties: μ_θ is given a normal distribution with mean 0 and standard deviation 1000, and σ_θ has a uniform distribution from 0 to 1000. These are certainly noninformative, given that the data y all fall well below 100 in absolute value.

R script for data input, starting values, and running Bugs

We put the data into a file, `schools.dat`, in the R working directory, with headers describing the data:

```
school estimate sd
A  28  15
B   8  10
C  -3  16
D   7  11
E  -1   9
F   1  11
G  18  10
H  12  18
```

From R, we then execute the following script, which reads in the dataset, puts it in list format to be read by Bugs, sets up a function to compute initial values for the Bugs run, and identifies parameters to be saved.

```
schools <- read.table ("schools.dat", header=T)
J <- nrow (schools)
y <- schools$estimate
sigma.y <- schools$sd
data <- list ("J", "y", "sigma.y")
inits <- function()
  list (theta=rnorm(J,0,100), mu.theta=rnorm(1,0,100),
        sigma.theta=runif(1,0,100))
parameters <- c("theta", "mu.theta", "sigma.theta")
```

In general, initial values can be set from crude estimates, as discussed in Section 10.1. In this particular example, we use the range of the data to construct roughly overdispersed distributions for the θ_j's, μ_θ, and σ_θ. Bugs does not require all parameters to be initialized, but it is a good idea to do so. We prefer to explicitly construct the starting points randomly as above, but it would also be possible to start the simulations with simple initial values; for example,

```
inits1 <- list (theta=y, mu.theta=0, sigma.theta=1)
inits2 <- list (theta=y, mu.theta=0, sigma.theta=10)
inits3 <- list (theta=y, mu.theta=0, sigma.theta=100)
inits <- list (inits1, inits2, inits3)
```

```
Inference for Bugs model at "c:/bugsR/schools.txt"
 3 chains, each with 1000 iterations (first 500 discarded)
 n.sims = 1500 iterations saved

            mean  sd  2.5%  25%  50%  75% 97.5% Rhat n.eff
theta[1]    11.0 8.5  -2.1  5.4  9.9 15.2  30.5  1.0    76
theta[2]     7.4 6.2  -5.0  3.7  7.1 11.4  20.1  1.0   290
theta[3]     5.5 7.8 -12.3  1.5  6.1 10.1  19.9  1.0  1400
theta[4]     7.4 6.6  -5.7  3.7  7.2 11.3  21.2  1.0   170
theta[5]     4.6 6.4  -9.6  0.5  5.4  8.8  16.0  1.0   390
theta[6]     5.6 6.6  -8.0  1.4  6.0 10.1  18.1  1.0   430
theta[7]    10.0 6.7  -2.1  5.6  9.4 14.0  25.0  1.0    73
theta[8]     8.0 7.8  -6.5  3.8  7.4 12.1  24.9  1.0   140
mu.theta     7.5 5.4  -2.8  4.3  7.3 10.7  19.0  1.0   160
sigma.theta  6.5 5.7   0.2  2.7  4.9  9.1  21.4  1.1    23
deviance    60.3 2.2  56.9 58.9 59.8 61.1  66.0  1.0    92
 pD = 2.4 and DIC = 62.7 (using the rule, pD = var(deviance)/2)
```

Figure C.1 *Numerical output from the* `bugs()` *function after being fitted to the hierarchical model for the educational testing example. For each parameter, n_{eff} is a rough measure of effective sample size, and $\widehat{R}$ is the potential scale reduction factor defined in Section 11.6 (at convergence, $\widehat{R} = 1$). The effective number of parameters p_D and the expected predictive error DIC are defined in Section 6.7.*

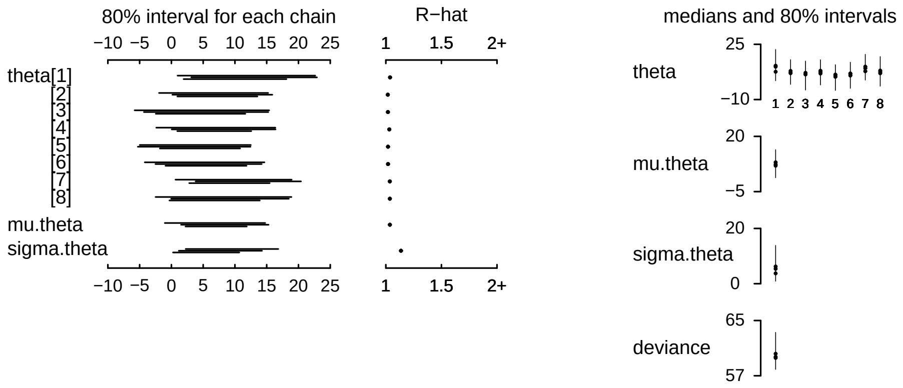

Figure C.2 *Graphical output from the* `bugs()` *function after being fitted to the hierarchical model for the educational testing example. The left side of the display shows the overlap of the three parallel chains, and the right side shows the posterior inference for each parameter and the deviance (−2 times the log likelihood). The on-screen display uses colors to distinguish the different chains.*

In any case, we can now run Bugs with 3 chains for 1000 iterations:

```
schools.sim <- bugs (data, inits, parameters, "schools.txt",
  n.chains=3, n.iter=1000)
```

While Bugs is running, it opens a new window and freezes R. When the computations are finished, summaries of the inferences and convergence are displayed as a table in the R console (see Figure C.1) and in an R graphics window (see Figure C.2). In this example, the sequences have mixed well—the estimated potential scale reduction factors $\widehat{R}$ are close to 1 for all parameters—and so we stop. (If some of the $\widehat{R}$ factors were still large, we would try running the simulation a bit longer—perhaps 2000 or 5000 iterations per chain—and if convergence were still a problem, we would consider reparameterizing the model. We give some examples of reparameterizing this model in Section C.3.) Running three chains for 1000 iterations and saving the second half of each chain (the default option) yields 1500 simulation draws of the parameter vector.

The output in Figure C.1 also gives information on the effective sample size n_{eff} for each parameter (as defined at the end of Section 11.6) and the deviance and DIC (see Section 6.7).

Accessing the posterior simulations in R

The output of the R function `bugs()` is a list which includes several components, most notably the summary of the inferences and convergence and a list containing the simulation draws of all the saved parameters. In the example above, the `bugs()` call is assigned to the R object `schools.sim`, and so typing `schools.sim$summary` from the R console will display the summary shown in Figure C.1.

The posterior simulations are saved in the list, `schools.sim$sims.list`. We we can directly access them by typing `attach.all(schools.sim$sims.list)` from the R console, at which point `mu.theta` is a vector of 1500 simulations of μ_θ, `sigma.theta` is a vector of 1500 simulations of σ_θ, and `theta` is a 1500×8 matrix of simulations of θ. Other output from the `bugs()` function is explained in comment lines within the function itself and can be viewed by typing `bugs` from the R console or examining the file `bugs.R`.

Posterior predictive simulations and graphs in R

Replicated data in the existing schools. Having run Bugs to successful convergence, we can work directly in R with the saved parameters, $\theta, \mu_\theta, \sigma_\theta$. For example, we can simulate posterior predictive replicated data in the original 8 schools:

```
y.rep <- array (NA, c(n.sims, J))
for (sim in 1:n.sims)
  y.rep[sim,] <- rnorm (J, theta[sim,], sigma.y)
```

We now illustrate a graphical posterior predictive check. There are not many ways to display a set of eight numbers. One possibility is as a histogram; the possible values of y^{rep} are then represented by an array of histograms as in Figure 6.2 on page 160. In R, this could be programmed as

```
par (mfrow=c(5,4), mar=c(4,4,2,2))
hist (y, xlab="", main="y")
for (sim in 1:19)
  hist (y.rep[sim,], xlab="", main=paste("y.rep",sim))
```

The upper-right histogram displays the observed data, and the other 19 histograms are posterior predictive replications, which in this example look similar to the data.

We could also compute a numerical test statistic such as the difference between the best and second-best of the 8 coaching programs:

```
test <- function (y){
  y.sort <- rev(sort(y))
  return (y.sort[1] - y.sort[2])
}
t.y <- test(y)
t.rep <- rep (NA, n.sims)
for (sim in 1:n.sims)
  t.rep[sim] <- test(y.rep[sim,])
```

We then can summarize the posterior predictive check. The following R code gives a numerical comparison of the test statistic to its replication distribution, a p-value, and a graph like those on pages 161 and 163:

```
cat ("T(y) =", round(t.y,1), "and T(y.rep) has mean",
     round(mean(t.rep),1), "and sd", round(sd(t.rep),1),
     "\nPr (T(y.rep) > T(y)) =", round(mean(t.rep>t.y),2), "\n")
hist0 <- hist (t.rep, xlim=range(t.y,t.rep), xlab="T(y.rep)")
lines (rep(t.y,2), c(0,1e6))
text (t.y, .9*max(hist0$count), "T(y)", adj=0)
```

Replicated data in new schools. As discussed in Section 6.8, another form of replication would simulate new parameter values and new data for eight *new* schools. To simulate data $y_j \sim \mathrm{N}(\theta_j, \sigma_j^2)$ from new schools, it is necessary to make some assumption or model for the data variances σ_j^2. For the purpose of illustration, we assume these are repeated from the original 8 schools.

```
theta.rep <- array (NA, c(n.sims, J))
y.rep <- array (NA, c(n.sims, J))
for (sim in 1:n.sims){
  theta.rep[sim,] <- rnorm (J, mu.theta[sim], sigma.theta[sim])
  y.rep[sim,] <- rnorm (J, theta.rep[sim,], sigma.y)
}
```

Numerical and graphical comparisons can be performed as before.

C.3 Options in the Bugs implementation

We next explore alternative ways that the model can be expressed in Bugs.

Alternative prior distributions

The model as programmed above has a uniform prior distribution on the hyperparameters μ_θ and σ_θ. An alternative is to parameterize in terms of τ_θ, the precision parameter, for which a gamma distribution is conditionally conjugate in this model. For example, the last two lines of the Bugs model in Section C.3 can be replaced by,

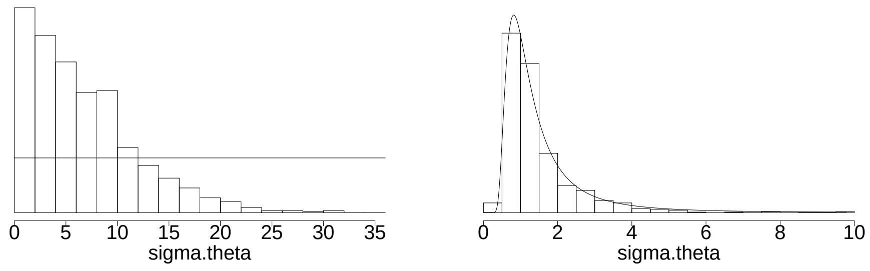

Figure C.3 *Histograms of posterior simulations of the between-school standard deviation, σ_θ, from models with two different prior distributions: (a) uniform prior distribution on σ_θ, and (b) conjugate Gamma$(1,1)$ prior distribution on the precision parameter $1/\sigma_\theta^2$. The two histograms are on different scales. Overlain on each is the corresponding prior density function for σ_θ. (For model (b), the density for σ_θ is calculated using the gamma density function from Appendix A multiplied by the Jacobian of the $1/\sigma_\theta^2$ transformation.) In model (b), posterior inferences are highly constrained by the prior distribution.*

```
  tau.theta ~ dgamma (1, 1)
  sigma.theta <- 1/sqrt(tau.theta)
```

The Gamma$(1,1)$ prior distribution is an attempt at noninformativeness within the conjugate family. (Recall that Bugs does not allow improper prior distributions.)

The initial values in the call to Bugs from R must now be redefined in terms of τ_θ rather than σ_θ; for example,

```
inits <- function ()
  list (theta=rnorm(J,0,100), mu.theta=rnorm(1,0,100),
        tau.theta=runif(1,0,100))
```

Otherwise, we fit the model as before. This new prior distribution leads to changed inferences. In particular, the posterior mean and median of σ_θ are lower and shrinkage of the θ_j's is greater than in the previously-fitted model with a uniform prior distribution on σ_θ. To understand this, it helps to graph the prior distribution in the range for which the posterior distribution is substantial. Figure C.3b shows that the prior distribution is concentrated in the range $[0.5, 5]$, a narrow zone in which the likelihood is close to flat (we can see this because the distribution of the posterior simulations of σ_θ closely matches the prior distribution, $p(\sigma_\theta)$). By comparison, in Figure C.3a, the uniform prior distribution on σ_θ seems closer to 'noninformative' for this problem, in the sense that it does not appear to be constraining the posterior inference.

Parameter expansion

A different direction to go in the Bugs programming is toward more efficient simulation by parameterizing the model so that the Gibbs sampler runs more effectively. Such tools are discussed in Section 11.8; here we illustrate how to set them up in Bugs. Introducing the new multiplicative parameter α, the model becomes

$$\begin{aligned} y_j &\sim \mathrm{N}(\mu + \alpha\gamma_j, \sigma_j^2) \\ \gamma_j &\sim \mathrm{N}(0, \sigma_\gamma^2) \\ p(\mu, \alpha, \sigma_\gamma) &\propto 1, \end{aligned} \tag{C.1}$$

which we code in Bugs as follows:

```
model {
  for (j in 1:J){
    y[j] ~ dnorm (theta[j], tau.y[j])
    theta[j] <- mu.theta + alpha*gamma[j]
    gamma[j] ~ dnorm (0, tau.gamma)
    tau.y[j] <- pow(sigma.y[j], -2)
  }
  mu.theta ~ dnorm (0.0, 1.0E-6)
  tau.gamma <- pow(sigma.gamma, -2)
  sigma.gamma ~ dunif (0, 1000)
  alpha ~ dunif(0,1000)
  sigma.theta <- abs(alpha)*sigma.gamma
}
```

The parameters of interest, θ and σ_θ, are defined in the Bugs model in terms of the expanded parameter set, $\gamma, \alpha, \mu_\theta, \sigma_\gamma$. The call from R must then specify initial values in the new parameterization; for example

```
inits <- function ()
  list (gamma=rnorm(J,0,100), alpha=runif(1,0,100),
        mu.theta=rnorm(1,0,100), sigma.gamma=runif(1,0,100))
```

and then the rest of the R program is unchanged.

Posterior predictive simulations

At the end of Section C.2 we illustrated the computation of posterior predictive simulations in R. It is also possible to perform these simulations in Bugs. For example, to simulate new data from the existing schools, we can add the following line in the Bugs model, inside the '`for (j in 1:J)`' loop (see the model on page 592, for example):

```
    y.rep[j] ~ dnorm (theta[j], tau.y[j])
```

Or, to simulate new data from new schools:

```
    theta.rep[j] ~ dnorm (mu.theta, tau.theta)
    y.rep[j] ~ dnorm (theta.rep[j], tau.y[j])
```

We can also include in the Bugs model the computation of a test statistic, for example, the difference between the two best coaching programs:

```
  J.1 <- J-1
  t.rep <- ranked(y.rep[],J) - ranked(y.rep[],J.1)
```

We must then add these derived quantities to the list of parameters saved in R:

```
parameters <- c("theta", "mu.theta", "sigma.theta", "y.rep", "t.rep")
```

We can then call the Bugs program from R as before.

Using the t model

It is straightforward to expand the hierarchical normal distribution for the coaching effects to a t distribution as discussed in Section 17.4. For simplicity, we return to the original form of the model without parameter expansion.

```
model {
  for (j in 1:J){
    y[j] ~ dnorm (theta[j], tau.y[j])
    theta[j] ~ dt (mu.theta, tau.theta, nu.theta)
    tau.y[j] <- pow(sigma.y[j], -2)
  }
  mu.theta ~ dnorm (0.0, 1.0E-6)
  tau.theta <- pow(sigma.theta, -2)
  sigma.theta ~ dunif (0, 1000)
  nu.theta <- 1/nu.inv.theta
  nu.inv.theta ~ dunif (0, .5)
}
```

Here, we are assigning a uniform distribution to the inverse of the shape parameter ν_θ, as in Section 17.4. In addition, Bugs requires the t degrees of freedom to be at least 2, so we have implemented that restriction on ν_θ. We run this model from R as before, adding `nu.inv.theta` to the set of initial values and `nu.theta` to the parameters saved:

```
inits <- function()
  list (theta=rnorm(J,0,100), mu.theta=rnorm(1,0,100),
        sigma.theta=runif(1,0,100), nu.inv.theta=runif(1,0,.5))
parameters <- c("theta", "mu.theta", "sigma.theta", "nu.theta")
```

Alternatively, the t model can be expressed as a normal model with a scaled inverse-χ^2 distribution for the variances, which corresponds to a gamma distributions for the precisions $\tau_{\theta\, j}$:

```
model {
  for (j in 1:J){
    y[j] ~ dnorm (theta[j], tau.y[j])
    theta[j] ~ dnorm (mu.theta, tau.theta[j])
    tau.y[j] <- pow(sigma.y[j], -2)
    tau.theta[j] ~ dgamma (a, b)
  }
  mu.theta ~ dnorm (0.0, 1.0E-6)
  a <- nu.theta/2
  b <- (nu.theta/2)*pow(sigma.theta, 2)
  sigma.theta ~ dunif (0, 1000)
  nu.theta <- 1/nu.inv.theta
  nu.inv.theta ~ dunif (0, 1)
}
```

In this indirect parameterization, Bugs allows the parameter ν_θ to be any positive value, and so we can apply a $U(0,1)$ prior distribution to $1/\nu_\theta$.

It would be possible to further elaborate the model in Bugs by applying parameter expansion to the t model, but we do not consider any further extensions here.

C.4 Fitting a hierarchical model in R

In this section we demonstrate several different computational approaches for analyzing the SAT coaching data by directly programming the computations in R. Compared to using Bugs, computation in R requires more programming effort but gives us direct control of the simulation algorithm, which is helpful in settings where the Gibbs sampler or Metropolis algorithm is slow to converge.

Marginal and conditional simulation for the normal model

We begin by programming the calculations in Section 5.4. The programs provided here return to the notation of Chapter 5 (for example, τ is the population standard deviation of the θ's) as this allows for easy identification of some of the variables in the programs (for example, `mu.hat` and `V.mu` are the quantities denoted by the corresponding symbols in (5.20)).

We assume that the dataset has been read into R as in Section C.2, with `J` the number of schools, `y` the vector of data values, and `sigma.y` the vector of standard deviations. Then the first step of our programming is to set up a grid for τ, evaluate the marginal posterior distribution (5.21) for τ at each grid point, and sample 1000 draws from the grid. The grid here is `n.grid`=2000 points equally spread from 0 to 40. Here we use the grid as a discrete approximation to the posterior distribution of τ. We first define $\hat{\mu}$ and V_μ of (5.20) as functions of τ and the data, as these quantities are needed here and in later steps, and then compute the log density for τ.

```
mu.hat <- function (tau, y, sigma.y){
  sum(y/(sigma.y^2 + tau^2))/sum(1/(sigma.y^2 + tau^2))
}
V.mu <- function (tau, y, sigma.y){
  1/sum(1/(tau^2 + sigma.y^2))
}
n.grid <- 2000
tau.grid <- seq (.01, 40, length=n.grid)
log.p.tau <- rep (NA, n.grid)
for (i in 1:n.grid){
  mu <- mu.hat (tau.grid[i], y, sigma.y)
  V <- V.mu (tau.grid[i], y, sigma.y)
  log.p.tau[i] <- .5*log(V) -
    .5*sum(log(sigma.y^2 + tau.grid[i]^2)) -
    .5*sum((y-mu)^2/(sigma.y^2 + tau.grid[i]^2))
}
```

We compute the posterior density for τ on the log scale and rescale it to eliminate the possibility of computational overflow or underflow that can occur when multiplying many factors.

```
log.p.tau <- log.p.tau - max(log.p.tau)
p.tau <- exp(log.p.tau)
p.tau <- p.tau/sum(p.tau)
n.sims <- 1000
tau <- sample (tau.grid, n.sims, replace=T, prob=p.tau)
```

The last step draws the simulations of τ from the approximate discrete distribution. The remaining steps are sampling from normal conditional distributions for μ and the θ_j's as in Section 5.4. The sampled values of the eight θ_j's are collected in an array.

```
mu <- rep (NA, n.sims)
theta <- array (NA, c(n.sims,J))
for (i in 1:n.sims){
  mu[i] <- rnorm (1, mu.hat(tau[i],y,sigma.y),
    sqrt(V.mu(tau[i],y,sigma.y)))
  theta.mean <- (mu[i]/tau[i]^2 + y/sigma.y^2)/
    (1/tau[i]^2 + 1/sigma.y^2)
  theta.sd <- sqrt(1/(1/tau[i]^2 + 1/sigma.y^2))
  theta[i,] <- rnorm (J, theta.mean, theta.sd)
}
```

We now have created 1000 draws from the joint posterior distribution of τ, μ, θ. Posterior predictive distributions are easily generated using the random number generation capabilities of R as described above in the Bugs context.

Gibbs sampler for the normal model

Another approach, actually simpler to program, is to use the Gibbs sampler. This computational approach follows the outline of Section 11.7 with the simplification that the observation variances σ_j^2 are known.

```
theta.update <- function (){
  theta.hat <- (mu/tau^2 + y/sigma.y^2)/(1/tau^2 + 1/sigma.y^2)
  V.theta <- 1/(1/tau^2 + 1/sigma.y^2)
  rnorm (J, theta.hat, sqrt(V.theta))
}
mu.update <- function (){
  rnorm (1, mean(theta), tau/sqrt(J))
}
tau.update <- function (){
  sqrt(sum((theta-mu)^2)/rchisq(1,J-1))
}
```

We now generate five independent Gibbs sampling sequences of length 1000. We initialize μ and τ with overdispersed values based on the range of the data y and then run the Gibbs sampler, saving the output in a large array, `sims`, that contains posterior simulation draws for θ, μ, τ.

```
n.chains <- 5
n.iter <- 1000
sims <- array (NA, c(n.iter, n.chains, J+2))
dimnames (sims) <- list (NULL, NULL,
  c (paste ("theta[", 1:8, "]", sep=""), "mu", "tau"))
for (m in 1:n.chains){
  mu <- rnorm (1, mean(y), sd(y))
  tau <- runif (1, 0, sd(y))
  for (t in 1:n.iter){
    theta <- theta.update ()
    mu <- mu.update ()
    tau <- tau.update ()
    sims[t,m,] <- c (theta, mu, tau)
  }
}
```

We then check the mixing of the sequences using the R function `monitor` that carries out the multiple-chain diagnostic described in Section 11.6. We round to one decimal place to make the results more readable:

```
round (monitor (sims), 1)
```

The `monitor` function has automatically been loaded if you have followed the instructions for setting up R at the beginning of this chapter. The function takes as input an array of posterior simulations from multiple chains, and it returns an estimate of the potential scale reduction $\widehat{R}$, effective sample size n_{eff}, and summary statistics for the posterior distribution (based on the last half of the simulated Markov chains).

The model can also be computed using the alternative parameterizations and prior distributions that we implemented in Bugs in Section C.3. For example, in the parameter-expanded model (C.1), the Gibbs sampler steps can be programmed as

```
gamma.update <- function (){
  gamma.hat <- (alpha*(y-mu)/sigma.y^2)/(1/tau^2 + alpha^2/sigma.y^2)
  V.gamma <- 1/(1/tau^2 + alpha^2/sigma.y^2)
  rnorm (J, gamma.hat, sqrt(V.gamma))
}
alpha.update <- function (){
  alpha.hat <- sum(gamma*(y-mu)/sigma.y^2)/sum(gamma^2/sigma.y^2)
  V.alpha <- 1/sum(gamma^2/sigma.y^2)
  rnorm (1, alpha.hat, sqrt(V.alpha))
}
mu.update <- function (){
  mu.hat <- sum((y-alpha*gamma)/sigma.y^2)/sum(1/sigma.y^2)
  V.mu <- 1/sum(1/sigma.y^2)
  rnorm (1, mu.hat, sqrt(V.mu))
}
tau.update <- function (){
  sqrt(sum(gamma^2)/rchisq(1,J-1))
}
```

The Gibbs sampler can then be implemented as

```
sims <- array (NA, c(n.iter, n.chains, J+2))
dimnames (sims) <- list (NULL, NULL,
  c (paste ("theta[", 1:8, "]", sep=""), "mu", "tau"))
for (m in 1:n.chains){
  alpha <- 1
  mu <- rnorm (1, mean(y), sd(y))
  tau <- runif (1, 0, sd(y))
  for (t in 1:n.iter){
    gamma <- gamma.update ()
    alpha <- alpha.update ()
    mu <- mu.update ()
    tau <- tau.update ()
    sims[t,m,] <- c (mu+alpha*gamma, mu, abs(alpha)*tau)
  }
}
round (monitor (sims), 1)
```

Gibbs sampling for the t model with fixed degrees of freedom

As described in Chapter 17, the t model can be implemented using the Gibbs sampler using the normal/inverse-χ^2 parameterization for the θ_j's and their variances. Following the notation of that chapter, we take V_j to be the variance for θ_j and model the V_j's as draws from an inverse-χ^2 distribution with degrees of freedom ν and scale τ. As with the normal model, we use a uniform prior distribution on (μ, τ).

As before, we first create the separate updating functions, including a new function to update the individual-school variances V_j.

```
theta.update <- function (){
  theta.hat <- (mu/V + y/sigma.y^2)/(1/V + 1/sigma.y^2)
  V.theta <- 1/(1/V + 1/sigma.y^2)
  rnorm (J, theta.hat, sqrt(V.theta))
}
mu.update <- function (){
  mu.hat <- sum(theta/V)/sum(1/V)
  V.mu <- 1/sum(1/V)
  rnorm (1, mu.hat, sqrt(V.mu))
}
tau.update <- function (){
  sqrt (rgamma (1, J*nu/2+1, (nu/2)*sum(1/V)))
}
V.update <- function (){
  (nu*tau^2 + (theta-mu)^2)/rchisq(J,nu+1)
}
```

Initially we fix the degrees of freedom at 4 to provide a robust analysis of the data.

```
sims <- array (NA, c(n.iter, n.chains, J+2))
dimnames (sims) <- list (NULL, NULL,
  c (paste ("theta[", 1:8, "]", sep=""), "mu", "tau"))
nu <- 4
for (m in 1:n.chains){
  mu <- rnorm (1, mean(y), sd(y))
  tau <- runif (1, 0, sd(y))
  V <- runif (J, 0, sd(y))^2
  for (t in 1:n.iter){
    theta <- theta.update ()
    V <- V.update ()
    mu <- mu.update ()
    tau <- tau.update ()
    sims[t,m,] <- c (theta, mu, tau)
  }
}
round (monitor (sims), 1)
```

Gibbs-Metropolis sampling for the t model with unknown degrees of freedom

We can also include ν, the degrees of freedom in the above analysis, as an unknown parameter and update it conditional on all the others using the Metropolis algorithm. We follow the discussion in Chapter 17 and use a uniform prior distribution on $(\mu, \tau, 1/\nu)$.

The Metropolis updating function calls a function `log.post` to calculate the logarithm of the conditional posterior distribution of $1/\nu$ given all of the other parameters. (We work on the logarithmic scale to avoid computational overflows, as mentioned in Section 10.3.) The log posterior density function for this model has three terms—the logarithm of a normal density for the data points y_j, the logarithm of a normal density for the school effects θ_j, and the logarithm of an inverse-χ^2 density for the variances V_j. Actually, only the last term involves ν, but for generality we compute the entire log-posterior density in the `log.post` function.

```
log.post <- function (theta, V, mu, tau, nu, y, sigma.y){
  sum(dnorm(y,theta,sigma.y,log=T)) +
    sum(dnorm(theta,mu,sqrt(V),log=T)) +
    sum (.5*nu*log(nu/2) + nu*log(tau) -
         lgamma(nu/2) - (nu/2+1)*log(V) - .5*nu*tau^2/V)
}
```

We introduce the function that performs the Metropolis step and then describe how to alter the R code given earlier to incorporate the Metropolis step. The following function performs the Metropolis step for the degrees of freedom (recall that we work with the reciprocal of the degrees of freedom). The jumping distribution is normal with mean at the current value and standard deviation `sigma.jump.nu` (which is set as described below). We compute the jumping probability as described on page 289, setting it to zero if the proposed value of $1/\nu$ is outside the interval $(0, 1]$ to ensure that such proposals are rejected.

```
nu.update <- function (sigma.jump.nu){
  nu.inv.star <- rnorm(1, 1/nu, sigma.jump.nu)
  if (nu.inv.star<=0 | nu.inv.star>1)
    p.jump <- 0
  else {
    nu.star <- 1/nu.inv.star
    log.post.old <- log.post (theta, V, mu, tau, nu, y, sigma.y)
    log.post.star <- log.post (theta, V, mu, tau, nu.star,y,sigma.y)
    r <- exp (log.post.star - log.post.old)
    nu <- ifelse (runif(1) < r,  nu.star, nu)
    p.jump <- min(r,1)
  }
  return (nu=nu, p.jump=p.jump)
}
```

This updating function stores the acceptance probability `p.jump.nu` which is used in adaptively setting the jumping scale `sigma.jump.nu`, as we discuss when describing the Gibbs-Metropolis loop.

Given these functions, it is relatively easy to modify the R code that we have already written for the t model with fixed degrees of freedom. When computing the Metropolis updates, we store the acceptance probabilities in an array, `p.jump.nu`, to monitor the efficiency of the jumping. Theoretical results given in Chapter 11 suggest that for a single parameter the optimal acceptance rate—that is, the average probability of successfully jumping—is approximately 44%. Thus we can vary `sigma.jump.nu` and run a pilot study to determine an acceptable value. In this case we can settle on a value such as `sigma.jump.nu`=1, which has an average jumping probability of about 0.4 for these data.

```
sigma.jump.nu <- 1
p.jump.nu <- array (NA, c(n.iter, n.chains))
sims <- array (NA, c(n.iter, n.chains, J+3))
dimnames (sims) <- list (NULL, NULL,
  c (paste ("theta[", 1:8, "]", sep=""), "mu", "tau", "nu"))
for (m in 1:n.chains){
  mu <- rnorm (1, mean(y), sd(y))
  tau <- runif (1, 0, sd(y))
  V <- runif (J, 0, sd(y))^2
  nu <- 1/runif(1, 0, 1)
  for (t in 1:n.iter){
    theta <- theta.update ()
    V <- V.update ()
    mu <- mu.update ()
    tau <- tau.update ()
    temp <- nu.update (sigma.jump.nu)
    nu <- temp$nu
    p.jump.nu[t,m] <- temp$p.jump
    sims[t,m,] <- c (theta, mu, tau, nu)
  }
}
print (mean (p.jump.nu))
round (monitor (sims), 1)
```

Parameter expansion for the t model

Finally, we can make the computations for the t model more efficient by applying parameter expansion. In the expanded parameterization, the new Gibbs sampler steps can be programmed in R as

```
gamma.update <- function (){
  gamma.hat <- (alpha*(y-mu)/sigma.y^2)/(1/V + alpha^2/sigma.y^2)
  V.gamma <- 1/(1/V + alpha^2/sigma.y^2)
  rnorm (J, gamma.hat, sqrt(V.gamma))
}
alpha.update <- function (){
  alpha.hat <- sum(gamma*(y-mu)/sigma.y^2)/sum(gamma^2/sigma.y^2)
  V.alpha <- 1/sum(gamma^2/sigma.y^2)
  rnorm (1, alpha.hat, sqrt(V.alpha))
}
mu.update <- function (){
  mu.hat <- sum((y-alpha*gamma)/sigma.y^2)/sum(1/sigma.y^2)
  V.mu <- 1/sum(1/sigma.y^2)
  rnorm (1, mu.hat, sqrt(V.mu))
}
tau.update <- function (){
  sqrt (rgamma (1, J*nu/2+1, (nu/2)*sum(1/V)))
}
V.update <- function (){
  (nu*tau^2 + gamma^2)/rchisq(J,nu+1)
}
nu.update <- function (sigma.jump){
  nu.inv.star <- rnorm(1, 1/nu, sigma.jump)
  if (nu.inv.star<=0 | nu.inv.star>1)
    p.jump <- 0
  else {
    nu.star <- 1/nu.inv.star
    log.post.old <- log.post (mu+alpha*gamma, alpha^2*V, mu,
                              abs(alpha)*tau, nu, y, sigma.y)
    log.post.star <- log.post (mu+alpha*gamma, alpha^2*V, mu,
                               abs(alpha)*tau, nu.star, y, sigma.y)
    r <- exp (log.post.star - log.post.old)
    nu <- ifelse (runif(1) < r,  nu.star, nu)
    p.jump <- min(r,1)
  }
  return (nu=nu, p.jump=p.jump)
}
```

The posterior density can conveniently be calculated in terms of the original parameterization, as shown in the function `nu.update()` above. We can then run the Gibbs-Metropolis algorithm as before (see the program on the bottom part of page 605 and the very top of page 606), adding an initialization step for α just before the '`for (t in 1:n.iter)`' loop:

```
alpha <- rnorm (1, 0, 1)
```

adding an updating step for α inside the loop,

```
alpha <- alpha.update ()
```

and replacing the last line inside the loop with simulations transformed to the original θ, μ, τ parameterization:

```
sims[t,m,] <- c (mu+alpha*gamma, mu, abs(alpha)*tau, nu)
```

We must once again tune the scale of the Metropolis jumps. We started for convenience at `sigma.jump.nu`$= 1$, and this time the average jumping probability for the Metropolis step is 17%. This is quite a bit lower than the optimal rate of 44% for one-dimensional jumping, and so we would expect to get a more efficient algorithm by decreasing the scale of the jumps (see Section 11.9). Reducing `sigma.jump.nu` to 0.5 yields an average acceptance probability `p.jump.nu` of 32%, and `sigma.jump.nu`$= 0.3$ yields an average jumping probability of 46% and somewhat more efficient simulations—that is, the draws of ν from the Gibbs-Metropolis algorithm are less correlated and yield a more accurate estimate of the posterior distribution. Decreasing `sigma.jump.nu` any further would make the acceptance rate too high and reduce the efficiency of the algorithm.

C.5 Further comments on computation

We have already given general computational tips at the end of Section 10.3: start by computing with simple models and compare to previous inferences when adding complexity. We also recommend getting started with smaller or simplified datasets, but this strategy was not really relevant to the current example with only 8 data points. Other practical issues that arise, and which we have discussed in Part III, include starting values, the choice of simulation algorithm, and methods for increasing simulation efficiency.

There are various ways in which the programs in this appendix could be made more computationally efficient. For example, in the Metropolis updating function `nu.update` for the t degrees of freedom in Section C.4, the log posterior density can be saved so that it does not need to be calculated twice at each step. It would also probably be good to use a more structured programming style in our R code (for example, in our updating functions `mu.update()`, `tau.update()`, and so forth) and perhaps to store the parameters and data as lists and pass them directly to the functions. We expect that there are many other ways in which our programs could be improved. Our general approach is to start with transparent (and possibly inefficient) code and then reprogram more efficiently once we know it is working.

We made several mistakes in the process of implementing the computations described in this appendix. Simplest were syntax errors in Bugs and related problems such as feeding in the wrong inputs when calling the `bugs()` function from R. We discovered and fixed these problems by using the `debug` option in `bugs()`; for example,

```
schools.sim <- bugs (data, inits, parameters, "schools.txt",
  n.chains=3, n.iter=10, debug=T)
```

and then inspecting the log file within the Bugs window.

We fixed syntax errors and other minor problems in the R code by cutting and pasting to run the scripts one line at a time, and by inserting print statements inside the R functions to display intermediate values.

We debugged the Bugs and R programs in this appendix by comparing them against each other, and by comparing each model to previously-fitted simpler models. We found many errors, including treating variances as standard deviations (for example, the command `rnorm(1,alpha.hat,V.alpha)` instead of `rnorm(1,alpha.hat,sqrt(V.alpha))` when simulating from a normal distribution in R), confusion between ν and $1/\nu$, forgetting a term in the log-posterior density, miscalculating the Metropolis updating condition, and saving the wrong output in the `sims` array in the Gibbs sampling loop.

More serious conceptual errors included the poor choice of conjugate prior distribution for τ_θ in the Bugs model at the beginning of Section C.3, which we realized was a problem by comparing to the posterior simulations as shown in Figure C.3b. We also originally had an error in the programming of the reparameterized model (C.1), both in R and Bugs (we included the parameter μ in the model for γ_j rather than for y_j). We discovered this mistake because the inferences differed dramatically from the simpler parameterization.

As the examples in this appendix illustrate, Bayesian computation is not always easy, even for relatively simple models. However, once a model has been debugged, it can be applied and then generalized to work for a range of problems. Ultimately, we find Bayesian simulation to be a flexible tool for fitting realistic models to simple and complex data structures, and the steps required for debugging are often parallel to the steps required to build confidence in a model. We can use R to graphically display posterior inferences and predictive checks.

C.6 Bibliographic note

`R` is available at R Project (2002), and its parent software package `S` is described by Becker, Chambers, and Wilks (1988). Two statistics texts that use `R` extensively are Fox (2002) and Venables and Ripley (2002). Information about `Bugs` appears at Spiegelhalter et al. (1994, 2003), and many examples appear in the textbooks of Congdon (2001, 2003). `R` and `Bugs` have online documentation, and their websites have pointers to various help files and examples. Several efforts are currently underway to develop Bayesian inference tools using `R`, for example Martin and Quinn (2002b), Plummer (2003), and Warnes (2003).