Appendix C

Computation in R and Stan

We illustrate some practical issues of simulation by fitting a single example—the hierarchical normal model for the eight schools 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 Stan, operating from within the general statistical package R. Sections C.3 and C.4 present several different ways of programming the model directly in R. These algorithms require programming efforts that are unnecessary for the Stan user but are useful knowledge for programming more advanced models for which Stan 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 in Section 10.7 and the discussion of Hamiltonian Monte Carlo and Stan in Sections 12.4–12.6.

C.1 Getting started with R and Stan

Go to `http://www.r-project.org/` and `http://mc-stan.org/`. Further information including links to help lists are available at these webpages. We anticipate continuing improvements in both packages in the years after this book is released, but the general computational strategies presented here should remain relevant.

R 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 and Metropolis algorithms (as we illustrate in Section C.3) or Hamiltonian Monte Carlo (as shown in Section C.4). Computationally intensive tasks can be programmed in Fortran or C and linked from R.

Stan is a high-level language in which the user specifies a model and has the option to provide starting values, and then a Markov chain simulation is automatically implemented for the resulting posterior distribution. It is possible to set up and fit models entirely within Stan, 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 Stan by calling it from R using the `stan()` function, as illustrated in Section C.2. Again, the details of these function calls might change as Stan continues to be developed, so refer to `http://mc-stan.org/` for the latest documentation.

When working in R and Stan, it is helpful to set up the computer to simultaneously display four windows: the R console, an R graphics window, a text editor with the R script, and a text editor with Stan code. Rather than typing directly into R, we prefer to enter the R code into the editor and then source the file to run the commands in the R console. Using the text editor is convenient because it allows more flexibility in writing functions and loops. Another alternative is to use a workspace such as RStudio (`http://rstudio.org/`) which maintains several windows within a single environment.

C.2 Fitting a hierarchical model in Stan

In this section, we describe all the steps by which we would use Stan to fit the hierarchical normal model to the educational testing experiments in Section 5.5. These steps include writing the model in Stan and using R to set up the data and starting values, call Stan, create predictive simulations, and graph the results.

Stan program

The hierarchical model can be written in Stan in the following form, which we save as a file, `schools.stan`, in our working directory:

```
data {
  int<lower=0> J;          // number of schools
  real y[J];               // estimated treatment effects
  real<lower=0> sigma[J];  // s.e.'s of effect estimates
}
parameters {
  real mu;                 // population mean
  real<lower=0> tau;       // population sd
  vector[J] eta;           // school-level errors
}
transformed parameters {
  vector[J] theta;         // school effects
  theta <- mu + tau*eta;
}
model {
  eta ~ normal(0, 1);
  y ~ normal(theta, sigma);
}
```

The first paragraph of the above code specifies the data: the number of schools, J; the estimates, $y_1, \ldots, y_J$; and the standard errors, $\sigma_1, \ldots \sigma_J$. Data are labeled as integer or real and can be vectors (or, more generally, arrays) if dimensions are specified. Data can also be constrained; for example, in the above model J has been restricted to be nonnegative and the components of σ_y must all be positive.

The code next introduces the parameters: the unknowns to be estimated in the model fit. These are the school effects, θ_j; the mean, μ, and standard deviation, τ, of the population of school effects, the school-level errors η, and the effects, θ. In this model, we let θ be a transformation of μ, τ, and η instead of directly declaring θ as a parameter. By parameterizing this way, the sampler runs more efficiently; the resulting multivariate geometry is better behaved for Hamiltonian Monte Carlo.

Finally comes the model, which looks similar to how it would be written in this book. (Just be careful: in our book, the second argument to the $\mathrm{N}(\cdot,\cdot)$ distribution is the variance; Stan parameterizes using the standard deviation.) We have written the model in vector notation, which is cleaner and also runs faster in Stan by making use of more efficient autodifferentiation. It would also be possible to write the model more explicitly, for example replacing `y ~ normal(theta,sigma);` with a loop over the J schools, `for (j in 1:J) y[j] ~ normal(theta[j],sigma[j]);`.

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

We put the data into a file, `schools.csv`, 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 to read in the data:

```
schools <- read.csv("schools.csv", header=TRUE)
J <- nrow(schools)
y <- schools$estimate
sigma <- schools$sd
```

We load in the `rstan` package:

```
library("rstan")
```

We now run Stan with 4 chains of 1000 iterations each and display the results numerically and graphically:

```
schools_fit <- stan(file="schools.stan",
  data=c("J","y","sigma"), iter=1000, chains=4)
print(schools_fit)
plot(schools_fit)
```

When the computations are finished, summaries of the inferences and convergence are displayed in the R console (see Figure C.1) and in an R graphics window (not shown here).

In this example, the sequences appear to have mixed well—the estimated potential scale reduction factor $\widehat{R}$ is below 1.1 for all the parameters and quantities of interest displayed.

Stan uses a stochastic algorithm and so results will not be identical when re-running it. For example, here is the line of output for the parameter θ_1 in our first Stan run (repeated from Figure C.1):

```
          mean se_mean   sd  25%  50%  75%  n_eff Rhat
theta[1] 12.1     1.3 11.1  5.8 10.0 15.4     72    1
```

and here is the corresponding result from the second run:

```
          mean se_mean   sd  25%  50%  75%  n_eff Rhat
theta[1] 11.4     0.3  8.2  5.8 10.4 15.7    830    1
```

The inferences are similar but not identical. The simulation estimate for $\mathrm{E}(\theta_1|y)$ is 12.1 under one simulation and 11.4 under the other, not much of a difference considering that the posterior standard deviation is about 10 (more precisely, estimated to be 11.0 under one simulation and 8.2 under the other). The quantiles have the same general feel, but one must beware of overinterpretation. For example, the 95% posterior interval for θ_1 is $[-3.0, 40.1]$ in one simulation and $[-1.9, 31.5]$ in the other. (The 95% interval can be obtained from the function call, `print(schools_fit,"theta[1]",probs=c(.025,.975));` it is not shown in the default display in Figure C.1.) In practice, the intervals from the two different simulation runs contain simular information but their variability indicates that, even after approximate convergence, the tail quantiles of posterior quantities have a fair

```
Inference for Stan model: schools.
4 chains, each with iter=1000; warmup=500; thin=1;
post-warmup draws per chain=500, total post-warmup draws=2000.

         mean se_mean   sd  25%  50%  75% n_eff Rhat
mu        7.4     0.2  4.8  4.5  7.4 10.5   534    1
tau       6.9     0.5  6.1  2.4  5.4  9.3   138    1
eta[1]    0.4     0.1  0.9 -0.2  0.4  1.1   332    1
eta[2]    0.0     0.0  0.9 -0.5  0.1  0.6  1052    1
eta[3]   -0.2     0.0  1.0 -0.9 -0.3  0.4   820    1
eta[4]    0.0     0.0  0.8 -0.5 -0.1  0.5   848    1
eta[5]   -0.3     0.0  0.8 -0.9 -0.3  0.1  1051    1
eta[6]   -0.2     0.0  0.9 -0.8 -0.2  0.4   676    1
eta[7]    0.3     0.0  0.9 -0.2  0.4  1.0   793    1
eta[8]    0.1     0.0  0.9 -0.6  0.1  0.6   902    1
theta[1] 12.1     1.3 11.1  5.8 10.0 15.4    72    1
theta[2]  7.8     0.2  5.9  3.9  7.7 11.6   934    1
theta[3]  4.8     0.5  9.0  1.0  6.2 10.3   301    1
theta[4]  7.0     0.3  6.7  3.0  6.9 11.3   512    1
theta[5]  4.5     0.3  6.4  0.2  5.0  8.9   604    1
theta[6]  5.6     0.6  7.7  1.9  6.5 10.3   142    1
theta[7] 10.5     0.3  7.0  5.4  9.8 14.6   636    1
theta[8]  8.3     0.4  8.2  3.6  8.0 12.8   532    1
lp__     -4.9     0.2  2.6 -6.6 -4.8 -3.0   201    1

Samples were drawn using NUTS2 at Wed Apr 24 13:36:13 2013.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Figure C.1 *Numerical output from the* `print()` *function applied to the Stan code of the hierarchical model for the educational testing example. For each parameter,* `mean` *is the estimated posterior mean (computed as the average of the saved simulation draws),* `se_mean` *is the estimated standard error (that is, Monte Carlo uncertainty) of the mean of the simulations, and* `sd` *is the standard deviation. Thus, as the number of simulation draws approaches infinity,* `se_mean` *approaches zero while* `sd` *approaches the posterior standard deviation of the parameter. Then come several quantiles, then the effective sample size* n_{eff} *(formula (11.8) on page 287) and the potential scale reduction factor* $\widehat{R}$ *(see (11.4) on page 285). When all the simulated chains have mixed,* $\widehat{R} = 1$*. Beyond this, the effective sample size and standard errors give a sense of whether the simulations suffice for practical purposes. Each line of the table shows inference for a single scalar parameter in the model, with the last line displaying inference for the unnormalized log posterior density calculated at each step in Stan.*

amount of simulation variability. The importance of this depends on what the simulation will be used for.

Both simulations show good mixing ($\widehat{R} \approx 1$), but the effective sample sizes are much different. This sort of variation is expected, as n_{eff} is itself a random variable estimated from simulation draws. The simulation with higher effective sample size has a lower standard error of the mean and more stable estimates.

Accessing the posterior simulations in R

The output of the R function `stan()` is an object from which can be extracted various information regarding convergence and performance of the algorithm as well as a matrix of simulation draws of all the parameters, following the basic idea of Figure 1.1 on page 24.

For example:

```
schools_sim <- extract(schools_fit1)
```

The result is a list with four elements corresponding to the five quantities saved in the model: `theta`, `eta`, `mu`, `tau`, `lp__`. The vector θ of length 8 becomes a 20,000 $\times$ 8 matrix of simulations, the vector η similarly becomes a 20,000 $\times$ 8 matrix, the scalars μ and τ each become a vector of 20,000 draws, and the 20,000 draws of the unnormalized log posterior density are saved as the fourth element of the list.

For example, we can display posterior inference for τ:

```
hist(schools_sim$tau)
```

Or compute the posterior probability that the effect is larger in school A than in school C:

```
mean(schools_sim$theta[,1] > schools_sim$theta[,3])
```

Posterior predictive simulations and graphs in R

Replicated data in the existing schools. Having run Stan to successful convergence, we can work directly in R with the saved parameters, θ, μ, τ. For example, we can simulate posterior predictive replicated data in the original 8 schools:

```
n_sims <- length(schools_sim$lp__)
y_rep <- array(NA, c(n_sims, J))
for (s in 1:n_sims)
  y_rep[s,] <- rnorm(J, schools_sim$theta[s,], sigma)
```

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 144. In R, this could be programmed as

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

The upper-left 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(s in 1:n_sims)
  t_rep[s] <- test(y_rep[s,])
```

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 144 and 148:

```
par(mfrow=c(1,1))
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 6.5, 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 (s in 1:n_sims){
  theta_rep[s,] <- rnorm(J, schools_sim$mu[s], schools_sim$tau[s])
  y_rep[s,] <- rnorm(J, theta_rep[s,], sigma)
}
```

Numerical and graphical comparisons can be performed as before.

Alternative prior distributions

The model as programmed above has nearly uniform prior distributions on the hyperparameters μ_θ and σ_θ. An alternative is a half-Cauchy for σ_θ, which we could implement by taking the Stan model on page 592 and adding the line, `tau ~ cauchy(0,25);`.

We can fit the model as before. This new hyperprior distribution leads to changed inferences. In particular, the posterior mean and median of τ are lower and shrinkage of the θ_j's is greater than in the previously fitted model with a uniform prior distribution on τ. To understand this, it helps to graph the prior density in the range for which the posterior distribution is substantial. Figure 5.9 on page 131 shows that the prior density is a decreasing function of τ which has the effect of shortening the tail of the posterior density.

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, by replacing `eta ~ normal(0,1);` with `eta ~ student_t(nu,0,1);` and declaring `nu` as a parameter that takes on a value of 1 or greater (`real<lower=1> nu;`) and assigning it a prior distribution.

C.3 Direct simulation, Gibbs, and Metropolis in R

In this section we demonstrate several different ways to fit the 8-schools model by directly programming the computations in R.

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` 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){
  sum(y/(sigma^2 + tau^2))/sum(1/(sigma^2 + tau^2))
}
V_mu <- function(tau, y, sigma){
  1/sum(1/(tau^2 + sigma^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)
  V <- V_mu(tau_grid[i], y, sigma)
  log_p_tau[i] <- .5*log(V) - .5*sum(log(sigma^2 + tau_grid[i]^2)) -
    .5*sum((y-mu)^2/(sigma^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=TRUE, 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), sqrt(V_mu(tau[i],y,sigma)))
  theta_mean <- (mu[i]/tau[i]^2 + y/sigma^2)/(1/tau[i]^2 + 1/sigma^2)
  theta_sd <- sqrt(1/(1/tau[i]^2 + 1/sigma^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 Stan 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 11.6 with the simplification that the observation variances σ_j^2 are known.

```
theta_update <- function(){
  theta_hat <- (mu/tau^2 + y/sigma^2)/(1/tau^2 + 1/sigma^2)
  V_theta <- 1/(1/tau^2 + 1/sigma^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.

```
chains <- 5
iter <- 1000
sims <- array(NA, c(iter, chains, J+2))
dimnames(sims) <- list(NULL, NULL,
  c(paste("theta[", 1:8, "]", sep=""), "mu", "tau"))
for (m in 1:chains){
  mu <- rnorm(1, mean(y), sd(y))
  tau <- runif(1, 0, sd(y))
  for (t in 1: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 convergence diagnostic and effective sample size computation described in Section 11.4:

```
monitor(sims)
```

The `monitor` function is part of the `rstan` package and thus is already loaded if you have entered `library("rstan")` in your current R session. 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 alternative parameterizations. For example, in a parameter-expanded model, the Gibbs sampler steps can be programmed as

```
gamma_update <- function(){
  gamma_hat <- (alpha*(y-mu)/sigma^2)/(1/tau^2 + alpha^2/sigma^2)
  V_gamma <- 1/(1/tau^2 + alpha^2/sigma^2)
  rnorm(J, gamma_hat, sqrt(V_gamma))
}
alpha_update <- function(){
  alpha_hat <- sum(gamma*(y-mu)/sigma^2)/sum(gamma^2/sigma^2)
  V_alpha <- 1/sum(gamma^2/sigma^2)
```

```
    rnorm(1, alpha_hat, sqrt(V_alpha))
  }
  mu_update <- function(){
    mu_hat <- sum((y-alpha*gamma)/sigma^2)/sum(1/sigma^2)
    V_mu <- 1/sum(1/sigma^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(iter, chains, J+2))
  dimnames(sims) <- list(NULL, NULL,
    c(paste("theta[", 1:8, "]", sep=""), "mu", "tau"))
  for (m in 1:chains){
    alpha <- 1
    mu <- rnorm(1, mean(y), sd(y))
    tau <- runif(1, 0, sd(y))
    for (t in 1:iter){
      gamma <- gamma_update()
      alpha <- alpha_update()
      mu <- mu_update()
      tau <- tau_update()
      sims[t,m,] <- c(mu + alpha*gamma, mu, abs(alpha)*tau)
    }
  }
  monitor(sims)
```

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^2)/(1/V + 1/sigma^2)
    V_theta <- 1/(1/V + 1/sigma^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(iter, chains, J+2))
dimnames(sims) <- list(NULL, NULL,
  c(paste("theta[", 1:8, "]", sep=""), "mu", "tau"))
nu <- 4
for (m in 1: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:iter){
    theta <- theta_update()
    V <- V_update()
    mu <- mu_update()
    tau <- tau_update()
    sims[t,m,] <- c(theta, mu, tau)
  }
}
monitor(sims)
```

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 $(\mu, \tau, 1/\nu)$.

To do Metropolis updating function, we write 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 10.7.) 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:

```
log_post <- function(theta, V, mu, tau, nu, y, sigma){
  sum(dnorm(y, theta, sigma, log=TRUE)) +
    sum(dnorm(theta, mu, sqrt(V), log=TRUE)) +
    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 278, 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)
    log_post_star <- log_post(theta, V, mu, tau, nu_star,y,sigma)
    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%. We can vary `sigma_jump_nu` in a pilot study to aim for this rate. For this example we can settle on a value such as `sigma_jump_nu`=1, which has an average jumping probability of about 0.4.

```
sigma_jump_nu <- 1
p_jump_nu <- array(NA, c(iter, chains))
sims <- array(NA, c(iter, chains, J+3))
dimnames(sims) <- list(NULL, NULL,
  c(paste("theta[", 1:8, "]", sep=""), "mu", "tau", "nu"))
for (m in 1: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: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))
monitor(sims)
```

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^2)/(1/V + alpha^2/sigma^2)
  V_gamma <- 1/(1/V + alpha^2/sigma^2)
  rnorm(J, gamma_hat, sqrt(V_gamma))
}
alpha_update <- function(){
  alpha_hat <- sum(gamma*(y-mu)/sigma^2)/sum(gamma^2/sigma^2)
  V_alpha <- 1/sum(gamma^2/sigma^2)
  rnorm(1, alpha_hat, sqrt(V_alpha))
}
mu_update <- function(){
  mu_hat <- sum((y-alpha*gamma)/sigma^2)/sum(1/sigma^2)
  V_mu <- 1/sum(1/sigma^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)
    log_post_star <- log_post(mu+alpha*gamma, alpha^2*V, mu,
                              abs(alpha)*tau, nu_star, y, sigma)
    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 601 and the top of page 601), adding initialization steps for γ and α just before the '`for (t in 1:iter)`' loop:

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

adding updating steps for γ and α inside the loop,

```
  gamma <- gamma_update()
  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 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 12.2). 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.4 Programming Hamiltonian Monte Carlo in R

We demonstrate Hamiltonian Monte Carlo (HMC) by programming the basic eight-schools model. For this particular problem, HMC is overkill but it might help to have this code as a template.

We begin by reading in and setting up the data:

```
schools <- read.csv("schools.csv", header=TRUE)
J <- nrow(schools)
y <- schools$estimate
sigma <- schools$sd
```

Our model has 10 parameters, which we string into a single vector which we label as `th = (theta[1],...,theta[8],mu,tau)`. In the HMC program we work with this ten-dimensional vector, extracting its components as needed. First we program the log posterior density:

```
log_p_th <- function(th, y, sigma){
  J <- length(th) - 2
  theta <- th[1:J]
  mu <- th[J+1]
  tau <- th[J+2]
  if (is.nan(tau) | tau<=0)
    return(-Inf)
  else{
    log_hyperprior <- 1
    log_prior <- sum(dnorm(theta, mu, tau, log=TRUE))
    log_likelihood <- sum(dnorm(y, theta, sigma, log=TRUE))
    return(log_hyperprior + log_prior + log_likelihood)
  }
}
```

The scale parameter τ is restricted under the model to be positive, hence the if-statement above which has the effect of setting the posterior density to zero if τ jumps below zero.

Next we program the analytical gradient, the derivative of the log posterior with respect to each parameter:

```
gradient_th <- function(th, y, sigma){
  J <- length(th) - 2
  theta <- th[1:J]
  mu <- th[J+1]
  tau <- th[J+2]
  if (tau<=0)
    return(c(0,0,0))
```

```
    else {
      d_theta <- - (theta-y)/sigma^2 - (theta-mu)/tau^2
      d_mu <- -sum(mu-theta)/tau^2
      d_tau <- -J/tau + sum((mu-theta)^2)/tau^3
      return(c(d_theta, d_mu, d_tau))
    }
  }
```

If τ is less than zero, we have set the gradient to zero.

For debugging purposes we also write a numerical gradient function based on first differences:

```
gradient_th_numerical <- function(th, y, sigma){
  d <- length(th)
  e <- .0001
  diff <- rep(NA, d)
  for (k in 1:d){
    th_hi <- th
    th_lo <- th
    th_hi[k] <- th[k] + e
    th_lo[k] <- th[k] - e
    diff[k]<-(log_p_th(th_hi,y,sigma)-log_p_th(th_lo,y,sigma))/(2*e)
  }
  return(diff)
}
```

Next we program a single HMC iteration that takes as inputs the parameter vector θ, the data y, σ_y, the step size ϵ, the number of leapfrog steps L per iteration, and a diagonal mass matrix, expressed as a vector, M:

```
hmc_iteration <- function(th, y, sigma, epsilon, L, M) {
  M_inv <- 1/M
  d <- length(th)
  phi <- rnorm(d, 0, sqrt(M))
  th_old <- th
  log_p_old <- log_p_th(th,y,sigma) - 0.5*sum(M_inv*phi^2)
  phi <- phi + 0.5*epsilon*gradient_th(th, y, sigma)
  for (l in 1:L){
    th <- th + epsilon*M_inv*phi
    phi <- phi + (if (l==L) 0.5 else 1)*epsilon*gradient_th(th,y,sigma)
  }
  phi <- -phi
  log_p_star <- log_p_th(th,y,sigma) - 0.5*sum(M_inv*phi^2)
  r <- exp(log_p_star - log_p_old)
  if (is.nan(r)) r <- 0
  p_jump <- min(r,1)
  th_new <- if (runif(1) < p_jump) th else th_old
  return(list(th=th_new, p_jump=p_jump))
}
```

The above function performs the L leapfrog steps of an HMC iteration and returns the new value of θ (which is the same as the old value if the trajectory was rejected) and the acceptance probability, which can be useful in monitoring the efficiency of the algorithm.

Our next function is a wrapper that runs `hmc_iteration` and takes several arguments:

an $m \times d$ matrix of starting values (corresponding to m sequences and a vector of d parameters to start each chain); the number of iterations to run each chain; the baseline step size ϵ_0 and number of steps L_0, and the mass vector M. After setting up empty arrays to store the results, our HMC function, `hmc_run` runs the chains one at a time. Within each sequence, ϵ and L are randomly drawn at each iteration in order to mix up the algorithm and give it the opportunity to explore differently curved areas of the joint distribution. At the end, inferences are obtained using the last halves of the simulated sequences, summaries are printed out, and the simulations and acceptance probabilities are returned:

```
hmc_run <- function(starting_values, iter, epsilon_0, L_0, M) {
  chains <- nrow(starting_values)
  d <- ncol(starting_values)
  sims <- array(NA, c(iter, chains, d),
    dimnames=list(NULL, NULL, colnames(starting_values)))
  warmup <- 0.5*iter
  p_jump <- array(NA, c(iter, chains))
  for (j in 1:chains){
    th <- starting_values[j,]
    for (t in 1:iter){
      epsilon <- runif(1, 0, 2*epsilon_0)
      L <- ceiling(2*L_0*runif(1))
      temp <- hmc_iteration(th, y, sigma, epsilon, L, M)
      p_jump[t,j] <- temp$p_jump
      sims[t,j,] <- temp$th
      th <- temp$th
    }
  }
  monitor(sims, warmup)
  cat("Avg acceptance probs:",
    fround(colMeans(p_jump[(warmup+1):iter,]),2),"\n")
  return(list(sims=sims, p_jump=p_jump))
}
```

Now it is time to get ready to run the algorithm. We define a vector with the names of the parameters and set the number of chains to 4:

```
parameter_names <- c(paste("theta[",1:8,"]",sep=""), "mu", "tau")
d <- length(parameter_names)
chains <- 4
```

Next we define a diagonal mass matrix to be on the rough scale of the inverse variance matrix of the posterior distribution. Given the estimates and standard errors in Table 5.2 on page 120, we can crudely approximate this scale as 15 for each of the parameters; thus,

```
mass_vector <- rep(1/15^2, d)
```

Then we set up an array of random starting points defined on roughly the same scale, being careful to restrict the starting points for τ to be positive:

```
starts <- array(NA,c(chains,d),dimnames=list(NULL,parameter_names))
for (j in 1:chains){
  starts[j,] <- rnorm(d,0,15)
  starts[j,10] <- runif(1,0,15)
}
```

We are finally ready to go! We start with our default values $\epsilon_0 = 0.1, L_0 = 10$, running for only 20 iterations to make sure the program does not crash:

```
M1 <- hmc_run (starting_values=starts, iter=20,
  epsilon_0=.1, L_0=10, M=mass_vector)
```

The program runs fine so we go back for 100 iterations:

```
M2 <- hmc_run(starting_values=starts, iter=100,
  epsilon_0=.1, L_0=10, M=mass_vector)
```

Here are the results:

```
Inference for the input samples (4 chains: each with iter=100; warmup=50):

         mean se_mean  sd 25% 50%  75% n_eff Rhat
theta[1]  9.8     2.4 7.0 3.6 9.7 14.6     9  1.4
theta[2]  6.9     0.9 5.6 2.3 8.0 10.4    37  1.1
theta[3]  6.7     1.0 6.8 2.0 6.5 10.7    44  1.1
theta[4]  8.3     1.4 7.1 4.0 9.4 11.5    25  1.1
theta[5]  4.9     1.6 5.4 2.6 5.6  8.7    12  1.1
theta[6]  4.5     0.8 4.7 2.0 5.0  8.1    36  1.1
theta[7]  9.6     1.1 6.3 4.9 9.3 11.9    30  1.1
theta[8]  9.2     1.4 7.2 5.5 9.4 13.6    28  1.1
mu        7.4     0.7 3.9 4.4 7.2 10.2    27  1.1
tau       6.9     1.8 4.4 4.0 6.1  8.2     6  1.5

For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Avg acceptance probs: 0.27 0.42 0.59 0.68
```

The acceptance rates seem low (recall our goal of approximately 65% acceptances), and so we decrease the base step size from 0.1 to 0.05 and increase the base number of steps from 10 to 20:

```
M3 <- hmc_run(starting_values=starts, iter=100,
  epsilon_0=.05, L_0=20, M=mass_vector)
```

This looks better:

```
Inference for the input samples (4 chains: each with iter=100; warmup=50):

         mean se_mean   sd  25%  50%  75% n_eff Rhat
theta[1] 16.5     2.7 11.5  9.1 14.6 23.3    18  1.1
theta[2]  9.2     0.7  7.7  5.4  8.9 14.0   110  1.0
theta[3]  6.6     0.8  8.4  1.9  6.2 12.4    99  1.0
theta[4]  8.1     0.9  8.8  3.7  7.2 13.5    96  1.0
theta[5]  3.7     0.8  7.3 -2.1  3.5  8.9    75  1.0
theta[6]  6.2     2.2  7.7  2.1  5.9 11.4    13  1.1
theta[7] 12.9     1.1  8.8  7.2 13.2 19.1    61  1.1
theta[8]  9.4     2.9  9.2  4.3  9.2 15.1    10  1.1
mu        8.7     0.8  5.4  5.0  8.0 12.5    49  1.1
tau      10.0     1.9  6.3  6.1  8.4 12.6    11  1.3

For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Avg acceptance probs: 0.81 0.75 0.59 0.82
```

We re-run for 1000 and then 10,000 iterations and obtain stable inferences:

```
Inference for the input samples (4 chains: each with iter=10000; warmup=5000):

         mean se_mean  sd 25%  50%  75% n_eff Rhat
theta[1] 11.5     0.3 8.5 5.8 10.4 15.8  1129    1
theta[2]  7.9     0.2 6.5 3.7  7.8 12.1  1853    1
theta[3]  6.1     0.2 8.0 2.0  6.5 11.0  2434    1
theta[4]  7.6     0.2 6.8 3.4  7.6 11.8  1907    1
theta[5]  4.8     0.2 6.5 1.1  5.2  9.1  1492    1
theta[6]  6.1     0.1 6.8 2.1  6.3 10.4  2364    1
theta[7] 10.9     0.2 7.1 6.0 10.3 15.0  1161    1
theta[8]  8.5     0.2 8.1 3.6  8.1 13.1  1778    1
mu        8.0     0.2 5.4 4.4  7.9 11.3  1226    1
tau       6.9     0.2 5.5 2.9  5.6  9.4   565    1

For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Avg acceptance probs: 0.57 0.62 0.62 0.66
```

For this little hierarchical model, setting up and running HMC was costly both in programming effort and computation time, compared to the Gibbs sampler. More generally, though, Hamiltonian Monte Carlo can work in complicated problems where Gibbs and Metropolis fail, which is why in Stan we implemented HMC (via the no-U-turn sampler). In addition, this sort of hierarchical model exhibits better HMC convergence when parameterized in terms of the group-level errors (that is, the vector η, where $\theta_j = \mu + \tau\eta_j$ for $j = 1, \ldots, J$), as demonstrated in the Stan program in Section C.2.

C.5 Further comments on computation

We have already given general computational tips in Section 10.7: start by computing with simple models and compare to previous inferences when complexity is adding. We also recommend getting started with smaller or simplified datasets, but this strategy was not really relevant to the current example with only eight data points.

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.3, 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 Stan programs and related problems such as feeding in the wrong inputs when calling the `stan()` function from R.

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 Stan 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 expression `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 a poor choice of prior distribution, which we realized was a problem by comparing to posterior simulations computed using a different algorithm. We also originally had an error in the programming of a reparameterized model, a mistake we discovered 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). For more on Stan, see Stan Development Team (2012). R and Stan have online documentation, and their websites have pointers to various help files and examples.