CHAPTER

Developing Markov Chain Monte Carlo Samplers 17

In this chapter we will dive a little deeper into Markov chain Monte Carlo (MCMC) sampling. We will construct custom MCMC samplers in **R**, starting with easy-to-code Generalized Linear (Mixed) Models (GL(M)Ms) and moving on to basic CR and SCR models. This material might seem slightly out of place here, as it does not deal with specific aspects or modifications of SCR models, but rather, with a particular way of implementing them (and other models, too). Knowing how to build an MCMC sampler is not essential for any of the SCR models we have covered so far, but we will need these skills to implement some models that come up in the last few chapters of this book. The aim of this chapter is to provide you with some working knowledge of building MCMC samplers. To this end, we will *not* provide exhaustive background information on the theory and justification of MCMC sampling—there are entire books dedicated to that subject and we refer you to Robert and Casella (2004, 2010). Rather, we aim to provide you with enough background and technical know-how to start building your own MCMC samplers for SCR models in **R**. You will find that quite a few topics that come up in this chapter have already been covered in previous chapters, particularly the introduction into Bayesian analysis in Chapter 3. To keep you from having to leaf back and forth we will in some places briefly review aspects of Bayesian analysis, but we try to focus on the more technical issues of building MCMC samplers relevant to SCR models.

17.1 Why build your own MCMC algorithm?

The standard programs we have used so far to do MCMC analyses are **WinBUGS** (Gilks et al., 1994) and **JAGS** (Plummer, 2003). The wonderful thing about these **BUGS** engines is that they automatically use appropriate and, most of the time, reasonably efficient forms of MCMC sampling for the model specified by the user.

The fact that we have such a Swiss Army knife type of MCMC machine begs the question: Why would anyone want to build their own MCMC algorithm? For one, there are a limited number of distributions and functions implemented in **BUGS**. While **JAGS** provides more options, some more complex models may be impossible to build within these programs. A very simple example from spatial capture-recapture that can give you a headache in **WinBUGS** is when your state-space is an irregular-shaped polygon, rather than an ideal rectangle that can be characterized by four pairs of coordinates. It is easy to restrict activity centers to any arbitrary polygon in **R** using an ESRI shapefile (and we will show you an example in Section 17.7), but you cannot use a shapefile in a **BUGS** model. Similarly, models of space usage that take into account ecological distance (Chapter 12) cannot be implemented in the **BUGS** engines.

Sometimes, implementing an MCMC algorithm in **R** may be faster and more memory-efficient than in **WinBUGS**—especially if you want to run simulation studies where you have hundreds or more simulated data sets, several years' worth of data or other large models, this can be a big advantage. Further, writing your own sampler gives you more control over which kind of updater is used (see following sections). Finally, building your own MCMC algorithm is a great exercise to understand how MCMC sampling works. So while using the **BUGS** language requires you to understand the structure of your model, building an MCMC algorithm requires you to think about the relationship between your data, priors, and posteriors, and how these can be efficiently analyzed and characterized. However, if you don't think you will ever sit down and write your own MCMC sampler, consider skipping this chapter—apart from coding, it will not cover anything SCR-related that is not covered by other, more model-oriented chapters as well.

17.2 MCMC and posterior distributions

MCMC is a class of simulation methods for drawing (correlated) random numbers from a target distribution, which in Bayesian inference is the posterior distribution. As a reminder, the posterior distribution is a probability distribution for an unknown parameter, say θ, given observed data and its prior probability distribution (the probability distribution we assign to a parameter before we observe data). The great benefit of having the posterior distribution of θ is that it can be used to make probability statements about θ, such as the probability that θ is equal to some value, or the probability that θ falls within some range of values. The posterior distribution summarizes all we know about a parameter and thus, is the central object of interest in Bayesian analysis. Unfortunately, in many if not most practical applications, it is nearly impossible to compute the posterior directly. Recall Bayes' theorem:

$$[\theta|y] = \frac{[y|\theta][\theta]}{[y]}, \tag{17.2.1}$$

where θ is the parameter of interest, y is the observed data, $[\theta|y]$ is the posterior, $[y|\theta]$ is the likelihood of the data conditional on θ, $[\theta]$ is the prior probability distribution of θ, and, finally, $[y]$ is the marginal probability distribution of the data, defined as

$$[y] = \int [y|\theta][\theta]d\theta.$$

This marginal probability is a normalizing constant that ensures that the posterior integrates to 1. Often, the integral is difficult or impossible to evaluate, unless you are dealing with a really simple model. For example, consider a normal model, with a set of n observations, $y_i; i = 1, 2, \dots, n$:

$$y_i \sim \text{Normal}(\mu, \sigma),$$

where σ is known and our objective is to estimate μ. To fully specify the model in a Bayesian framework, we first have to define a prior distribution for μ. Recall from Chapter 3 that for certain data models, certain priors lead to conjugacy, i.e., if you choose a certain prior for your parameter, the posterior distribution will be of a known parametric form. More specifically, under conjugacy, the prior and posterior distributions are from the same parametric family. The conjugate prior for the mean of a normal model is also a normal distribution:

$$\mu \sim \text{Normal}(\mu_0, \sigma_0^2).$$

If μ_0 and σ_0^2 are fixed, the posterior for μ has the following form (for some of the algebra behind this, see Chapter 2 in Gelman et al. (2004)):

$$\mu | y \sim \text{Normal}(\mu_n, \sigma_n^2), \tag{17.2.2}$$

where

$$\mu_n = \left(\frac{\sigma^2}{\sigma^2 + n\sigma_0^2}\right) \times \left(\mu_0 + \frac{n\sigma_0^2}{\sigma^2 + n\sigma_0^2}\right) \times \bar{y}$$

and

$$\sigma_n^2 = \frac{\sigma^2 \sigma_0^2}{\sigma^2 + n\sigma_0^2}.$$

We can directly obtain estimates of interest from this normal posterior distribution, such as its mean $\hat{\mu}$ (which is equivalent to an estimate of μ_n) and variance; we do not need to apply MCMC, since we can recognize the posterior as a parametric distribution, including the normalizing constant $[y]$. But generally we will be interested in more complex models with several, say m, parameters. In this case, computing $[y]$ from Eq. (17.2.1) requires m-dimensional integration, which can be difficult or impossible. Thus, the posterior distribution is generally only known up to a constant of proportionality:

$$[\theta | y] \propto [y|\theta][\theta].$$

The power of MCMC is that it allows us to approximate the posterior using simulation without evaluating the highdimensional integrals, and to directly sample from the posterior, even when the posterior distribution is unknown! The price is that MCMC is computationally expensive. Although MCMC first appeared in the scientific literature in 1949 (Metropolis and Ulam, 1949), widespread use did not occur until the 1980s when computational power and speed increased (Gelfand and Smith, 1990). It is safe to say that the advent of practical MCMC methods is the primary reason why Bayesian inference has become so popular during the past three decades.

In a nutshell, MCMC lets us generate sequential draws of θ (the parameter(s) of interest) from distributions approximating the unknown posterior over T iterations. The distribution of the draw at t depends on the value drawn at $t-1$; hence, the draws form a Markov chain.[1] As T goes to infinity, the Markov chain converges to the desired distribution, in our case the posterior distribution for θ. Thus, once the Markov chain has reached its stationary distribution, the generated samples can be used to characterize the posterior distribution, $[\theta|y]$, and point estimates of θ, its standard error and confidence bounds, can be obtained directly from this approximation of the posterior.

17.3 Types of MCMC sampling

There are several general MCMC algorithms in widespread use, the most popular being Gibbs sampling and Metropolis-Hastings sampling, both of which were briefly introduced in Chapter 3. We will be dealing with these two classes in more detail and use them to construct MCMC algorithms for SCR models. Also, we will briefly review alternative techniques that are applicable in some situations.

17.3.1 Gibbs sampling

Gibbs sampling was named after the physicist J.W. Gibbs by Geman and Geman (1984), who applied the algorithm to a Gibbs distribution.[2] The roots of Gibbs sampling can be traced back to the work of Metropolis et al. (1953), and it is actually closely related to Metropolis sampling (see Section 11.5 in Gelman et al, 2004, for the link between the two samplers). We will focus on the technical aspects of this algorithm, but if you find yourself hungry for more background, Casella and George (1992) provide a more in-depth introduction to the Gibbs sampler.

Let's go back to our example from above to understand the motivation and functioning of Gibbs sampling. Recall that for a normal model with known variance and a normal prior for μ, the posterior distribution of μ is also normal. Conversely, with a fixed (known) μ, but unknown variance, the conjugate prior for σ^2 is an inverse-gamma distribution with shape and scale parameters a and b:

$$\sigma^2 \sim \text{Inverse-Gamma}(a, b).$$

With fixed a and b, algebra reveals that the posterior $[\sigma^2|\mu, y]$ is also an inverse-gamma distribution, namely:

$$\sigma^2|\mu, y \sim \text{Inverse-Gamma}(a_n, b_n), \tag{17.3.1}$$

[1]Remember that for T random samples $\theta^{(1)}, \ldots, \theta^{(T)}$ from a Markov chain the distribution of $\theta^{(t)}$ depends only on the immediately preceding value, $\theta^{(t-1)}$.

[2]A distribution from physics we are not going to worry about, since it has no immediate connection with Gibbs sampling other than giving its name.

where $a_n = n/2 + a$ and $b_n = (1/2)\sum_{i=1}^{n}(y_i - \mu)^2 + b$. However, what if we know neither μ nor σ^2, which is probably the more common case? The joint posterior distribution of μ and σ^2 now has the general structure

$$[\mu, \sigma^2|y] = \frac{[y|\mu, \sigma^2][\mu][\sigma^2]}{\int [y|\mu][\mu][\sigma^2] d\mu\, d\sigma^2}$$

or

$$[\mu, \sigma^2|y] \propto [y|\mu, \sigma^2][\mu][\sigma^2].$$

This cannot easily be reduced to a distribution we recognize. However, we can condition μ on σ^2 (i.e., we treat σ^2 as fixed) and remove all terms from the joint posterior distribution that do not involve μ to construct the full conditional distribution,

$$[\mu|\sigma^2, y] \propto [y|\mu][\mu].$$

The full conditional of μ again takes the form of the normal distribution shown in Eq. (17.2.2); similarly, $[\sigma^2|\mu, y]$ takes the form of the inverse-gamma distribution shown in Eq. (17.3.1), both distributions we can easily sample from. And this is precisely what we do when using Gibbs sampling: we break down high-dimensional problems into convenient one-dimensional problems by constructing the full conditional distributions for each model parameter separately; and we sample from these full conditionals, which, if we choose conjugate priors, are known parametric distributions. Let's put the concept of Gibbs sampling into the MCMC framework of generating successive samples, using our simple normal model with unknown μ and σ^2 and conjugate priors as an example. These are the steps you need in order to build a Gibbs sampler:

Step 0: Begin with some initial values for θ, say $\theta^{(0)}$. In our example, $\theta = (\mu, \sigma)$, so we have to specify initial values for μ and σ, for example by drawing a random number from some uniform distribution, or by setting them close to what we think they might be. (Note: This step is required in any MCMC sampling; chains have to start from somewhere. We will get back to these technical details a little later.)
Step 1: For iteration t, draw $\theta^{(t)}$ from the conditional distribution $[\theta_1^{(t)}|\theta_2^{(t-1)},\ldots, \theta_d^{(t-1)}]$. Here, θ_1 is μ, which we draw from the normal distribution in Eq. (17.2.2) using the $\sigma^{(t-1)}$ value for σ.
Step 2: Draw $\theta_2^{(t)}$ from the conditional distribution $[\theta_2^{(t)}|\theta_1^{(t)}, \theta_3^{(t-1)},\ldots, \theta_d^{(t-1)}]$. Here, θ_2 is σ, which we draw from the inverse-gamma distribution in Eq. (17.3.1), using the newly generated $\mu^{(t)}$ value for μ.
Step 3,…, d: Draw $\theta_3^{(t)}, \theta_4^{(t)},\ldots, \theta_d^{(t)}$ from their conditional distribution $[\theta_3^{(t)}|\theta_1^{(t)}, \theta_2^{(t)}, \theta_4^{(t-1)}, \ldots, \theta_d^{(t-1)}]$, …, $[\theta_d^{(t)}|\theta_1^{(t)},\ldots, \theta_{d-1}^{(t)}]$. In our example we have no additional parameters, so we only need Step 0 through to 2.

Repeat Steps 1 to d for T = a large number of samples.

```
Norm.Gibbs <- function(y=y,mu_0=mu_0,sigma2_0=sigma2_0,a=a,b=b,niter=niter){

ybar <- mean(y)
n <- length(y)
mu <- 1         #mean initial value
sigma2 <- 1     #sigma2 initial value
an <- n/2 + a   #shape parameter of IvGamma of sigma2
out <- matrix(nrow=niter, ncol=2)
colnames(out) <- c('mu', 'sig')

for (i in 1:niter) {

   #update mu
   mu_n <- ((sigma2/(sigma2+n*sigma2_0))*mu_0
   + (n*sigma2_0/(sigma2 + n*sigma2_0))*ybar)
   sigma2_n <- (sigma2*sigma2_0)/ (sigma2 + n*sigma2_0)
   mu <- rnorm(1,mu_n, sqrt(sigma2_n))

   #update sigma2
   bn <- 0.5 * (sum((y-mu)^2)) + b
   sigma2 <- 1/rgamma(1,shape=an, rate=bn)
   out[i,] <- c(mu,sqrt(sigma2))
}
return(out)
}
```

PANEL 17.1

R code for a Gibbs sampler for a normal model with unknown μ and σ and conjugate priors (normal and inverse-gamma, respectively) for both parameters.

Note that the order in which we update the parameters within the Gibbs algorithm does not matter. In terms of **R** coding, this means we have to write Gibbs updaters for μ and σ^2 and embed them into a loop over T iterations. The final code in the form of an **R** function is shown in Panel 17.1.

This is it! You can go ahead and simulate some data, $y \sim \text{Normal}(5, 0.5)$ and then use the function NormGibbs in the **R** package scrbook to run your first Gibbs sampler (note that the **R** function rnorm requires you to supply the standard deviation σ and we have written NormGibbs so that it returns σ instead of σ^2 so you can easily compare your input value and parameter estimate):

```
> set.seed(13)
#true mean and sd are 5 and 0.5
> y <- rnorm(1000, 5,0.5) #data

> mu_0 <- 0 #prior mean
> sigma2_0 <- 100 #prior variance
```

```
#inverse-gamma hyperparameters
> a <- 0.1
> b <- 0.1
```

```
> mod = Norm.Gibbs(y, mu_0, sigma2_0, a,b,niter=10000)
```

Your output, mod, will be a table with two columns, one per parameter, and T rows, one per iteration. For this 2-parameter example you can visualize the joint posterior by plotting samples of μ against samples of σ (Figure 17.1):

```
> plot(out[,1], out[,2])
```

The marginal distribution of each parameter is approximated by examining the samples of this particular parameter. You can visualize it by plotting a histogram of the samples (Figure 17.2 upper left and right):

```
> par(mfrow=c(1,2))
> hist(out[,1]); hist(out[,2])
```

Finally, recall an important characteristic of MCMC, namely, that the chain has to have converged (reached its stationary distribution) in order to regard samples as being from the posterior distribution. In practice, that means you have to throw out some of the initial samples called the burn-in. We will talk about this in more detail when we talk about convergence diagnostics. For now, you can use the `plot(out[,1])` or `plot(out[,2])` command to make a time series plot of the samples of each parameter and visually assess how many of the initial samples you should discard.

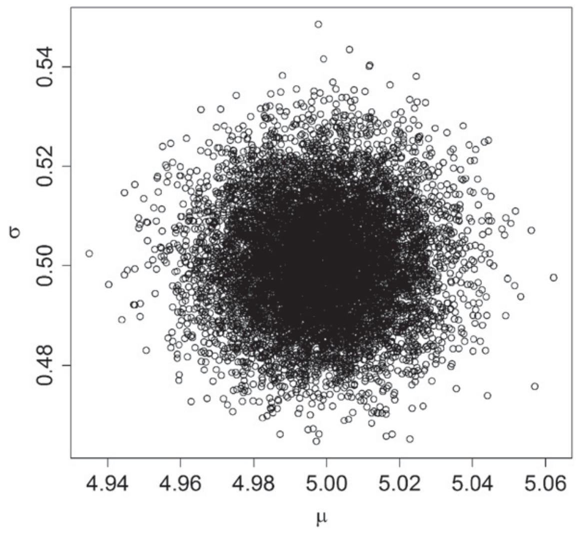

FIGURE 17.1

Joint posterior distribution of μ and σ from a normal model.

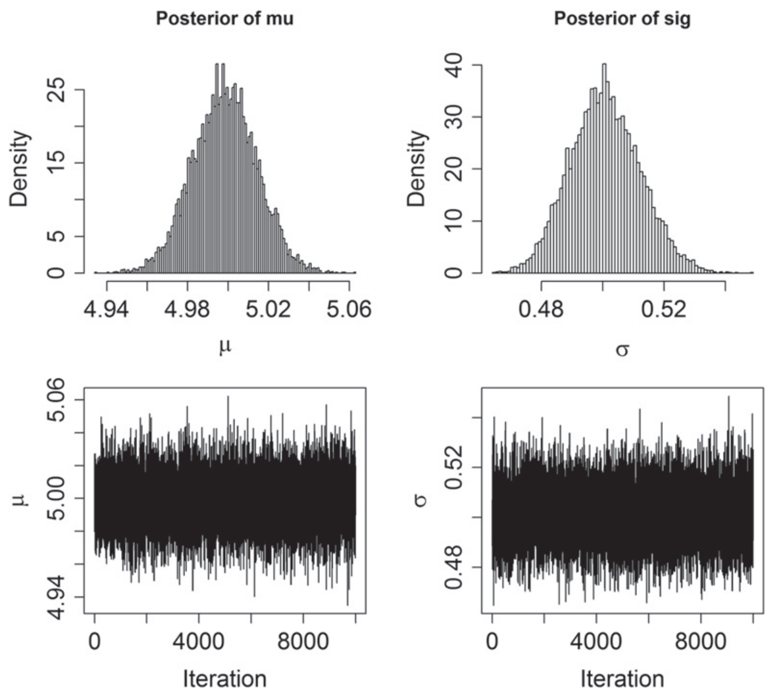

FIGURE 17.2

Plots of the posterior distributions of μ (upper left) and σ (upper right) from a normal model and time series plots of μ (lower left) and σ (lower right).

Figure 17.2 bottom left and right shows plots for the samples of μ and σ from our simulated data set; you see that in this simple example the Markov chain apparently reaches its stationary distribution very quickly—the chains look "grassy" seemingly from the start. It is hard to discern a burn-in phase visually (but we will see examples further on where the burn-in is clearer), and you may just discard the first 500 draws to be sure you only use samples from the posterior distribution. The mean of the remaining samples are your estimates of μ and σ, which, as we see below, are almost identical to the input values:

```
> summary(mod[501:10000,])
       mu              sig        
 Min.   :4.935  Min.   :0.4652  
 1st Qu.:4.988  1st Qu.:0.4930  
 Median :4.998  Median :0.5006  
 Mean   :4.998  Mean   :0.5008  
 3rd Qu.:5.009  3rd Qu.:0.5084  
 Max.   :5.062  Max.   :0.5486  
```

17.3.2 Metropolis-Hastings sampling

Although it is applicable to a wide range of problems, the limitations of Gibbs sampling are obvious: what if we do not want to use conjugate priors or what if we cannot recognize the full conditional distribution as a parametric distribution, or simply do not want to worry about these issues? The most general solution is to use the Metropolis-Hastings (MH) algorithm, which also goes back to the work by Metropolis et al. (1953). You saw the basics of this algorithm in Chapter 3. In a nutshell, because we do not recognize the posterior $[\theta|y]$ as a parametric distribution, the MH algorithm generates samples from a known proposal distribution, say $h(\theta)$, that depends on the value of θ at the previous time step, $\theta^{(t-1)}$. The candidate value θ^* is accepted with probability

$$r = \min\left(1, \frac{[\theta^*|y]h(\theta^{(t-1)}|\theta^*)}{[\theta^{(t-1)}|y]h(\theta^*|\theta^{(t-1)})}\right).$$

Proposal distributions must be chosen so that reversibility is ensured. That means, it must be possible to go from any one value to any other. But within that criterion the proposal distribution can be absolutely anything! You can generate candidate values from a Normal(0, 1) distribution, from a Uniform($-3455, 3455$) distribution, or anything of proper support. Note, however, that good choices of $h(\theta)$ are those that approximate the posterior distribution. Obviously, if $h(\theta) = [\theta|y]$ (i.e., the posterior) then you always accept the candidate value and it stands to reason that proposal distributions that are more similar to $[\theta|y]$ will lead to higher acceptance probabilities. Actually, when $h(\theta) = [\theta|y]$ we can draw samples of θ directly from $h(\theta)$, which brings us back to Gibbs sampling. Thus, Gibbs sampling is a special case of Metropolis-Hastings sampling.

The original Metropolis algorithm required $h(\theta)$ to be symmetric so that

$$h(\theta^*|\theta^{(t-1)}) = h(\theta^{(t-1)}|\theta^*).$$

In that case these two terms just cancel out from the MH acceptance probability and r is then just the ratio of the target density evaluated at the candidate value to that evaluated at the current value. A later development of the algorithm by Hastings (1970) lifted this condition. Since using a symmetric proposal distribution makes life a little easier, we are going to focus on this specific case. A type of symmetric proposal useful in many situations is the so-called *random walk* proposal distribution where candidate values are drawn from a normal distribution with the mean equal to the current value and some standard deviation, say δ, which is prescribed by the user (see below for further explanation).

Parameters with bounded support: Many models contain parameters that have bounded support, e.g., variance parameters live on $[0, \infty]$, parameters that represent probabilities live on $[0, 1]$, etc. For such cases, it is still sometimes convenient to use a random walk proposal distribution that can generate any real number (e.g., a normal random walk proposal). Under these circumstances you should not constrain the proposal distribution itself, but you can just reject parameters that are outside of the parameter space (Section 6.4.1 in Robert and Casella, 2010). You will see plenty of examples of updating parameters with bounded support in this chapter.

It is worth knowing that there are alternatives to the random walk MH algorithm. For example, in the independent MH, the proposal distribution h does not depend on $\theta^{(t-1)}$, while the Langevin algorithm (Roberts and Rosenthal, 1998) aims at avoiding the random walk by favoring moves toward regions of higher posterior probability density. The interested reader should look up these algorithms in Robert and Casella (2004~~) or (~~2010).

Building a MH sampler can be broken down into several steps. We are going to demonstrate these steps using a different but still simple and common model: the logit-normal or logistic regression model. For simplicity, assume that

$$y|\theta \sim \text{Bernoulli}\left(\frac{\exp(\theta)}{1+\exp(\theta)}\right)$$

and

$$\theta \sim \text{Normal}(\mu, \sigma).$$

The following steps are required to set up a random walk MH algorithm:

Step 0: Choose initial values, $\theta^{(0)}$.

Step 1: Generate a proposed value of θ from $h(\theta|\theta^{(t-1)})$. We will use the random walk MH algorithm, so we draw θ^* from Normal$(\theta^{(t-1)}, \delta)$, where δ is the standard deviation of the normal proposal distribution, the tuning parameter that we have to set.

Step 2: Calculate the ratio of posterior densities for the proposed and the original value for θ:

$$r = \frac{[\theta^*|y]}{[\theta^{(t-1)}|y]}.$$

In our example,

$$r = \frac{\text{Bernoulli}(y|\theta^*) \times \text{Normal}(\theta^*|\mu, \sigma)}{\text{Bernoulli}(y|\theta^{(t-1)}) \times \text{Normal}(\theta^{(t-1)}|\mu, \sigma)}.$$

Step 3: Set

$$\begin{aligned}\theta^t &= \theta^* \text{ with probability } \min(r, 1)\\ &= \theta^{(t-1)} \text{ otherwise.}\end{aligned}$$

We can do this last step by drawing a random number u from a Uniform(0, 1) and accept θ^* if $u < r$. This is repeated for $t = 1, 2, \ldots, T$ a large number of samples. As in Gibbs sampling, the order in which we update parameters does not matter. The **R** code for this MH sampler is provided in Panel 17.2.

The reason why, in the **R** code, we sum the logs of the likelihood and the prior, rather than multiplying the original values, is simply computational. The product of small probabilities can be numbers very close to 0, which computers do not handle well. Thus, we exponentiate the sum of the logarithms to achieve the desired result. Similarly, in case you have forgotten, $x/y = \exp(\log(x) - \log(y))$, with the latter being favored for computational reasons.

```
Logreg.MH <- function(y=y, mu0=mu0, sig0=sig0, delta=delta, niter=niter) {

out <- c()

theta <- runif(1, -3,3) #initial value

for (iter in 1:niter){
   theta.cand <- rnorm(1, theta, delta)

   loglike <- sum(dbinom(y, 1, exp(theta)/(1+exp(theta)), log=TRUE))
   logprior <- dnorm(theta,mu0 ,sig0, log=TRUE)
   loglike.cand <- sum(dbinom(y, 1, exp(theta.cand)/(1+exp(theta.cand)),
   log=TRUE))
   logprior.cand <- dnorm(theta.cand, mu0, sig0, log=TRUE)

   if (runif(1)<exp((loglike.cand+logprior.cand)-(loglike+logprior))){
   theta <- theta.cand
}
out[iter] <- theta
}

return(out)
}
```

PANEL 17.2

R code to run a Metropolis sampler on a simple logit-normal model.

Unlike Gibbs sampling, where all draws from the conditional distribution are used, the MH algorithm discards a portion of the candidate values, which inherently makes it less efficient than Gibbs sampling—the price you pay for its increased generality. In Step 1 of the MH sampler we had to choose a ~~variance~~, δ, for the normal proposal distribution. Choice of the parameters that define our candidate distribution is also referred to as "tuning," and it is important since adequate tuning will make your algorithm more efficient. δ should be chosen (a) large enough so that each step of drawing a new proposal value for θ can cover a reasonable distance in the parameter space; otherwise, mixing of the Markov chain is inefficient and chains will tend to have strong autocorrelation; and (b) small enough so that proposal values are not rejected too often; otherwise the random walk will "get stuck" at specific values for too long. As a rule of thumb, the candidate value should be accepted in about 40% of all cases. Acceptance rates of 20–80% are probably ok, but anything below or above may well render your algorithm inefficient (this does not mean that it will give you wrong results, only that you will need more iterations to converge to the posterior

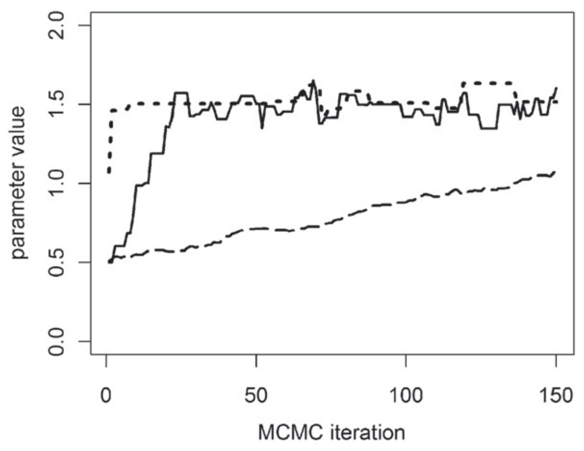

FIGURE 17.3

Time series plots of θ from a MH algorithm with tuning parameter $\delta = 0.01$ (dashed line), 0.2 (solid line), and 1 (dotted line).

distribution). In practice, tuning will require some "trial-and-error," some common sense and, with enough experience, some intuition. Or, one can use an adaptive phase, where the tuning parameter is automatically adjusted until it reaches a user-defined acceptance rate, at which point the adaptive phase ends and the actual Markov chain begins. This is computationally a little more advanced. Link and Barker (2010) discuss this in more detail. It is important that the samples drawn during the adaptive phase are discarded.

To illustrate the effects of tuning, we ran the Metropolis-Hastings algorithm in Panel 17.2 with $\delta = 0.01$, $\delta = 0.2$, and $\delta = 1$. The first 150 iterations for θ are shown in Figure 17.3. We see that for a very small δ (the dashed line) the burn-in is extremely slow—after 150 iterations the chain isn't even halfway there, while for the other two values of δ (solid and dotted lines) the burn-in phase seems to be over after only about 10 iterations. While $\delta = 0.2$ leads to reasonably good mixing, the chain clearly gets stuck on certain values with $\delta = 1$.

Other than graphically, you can easily check acceptance rates for the parameters you monitor (i.e., that are part of your output) using the `rejectionRate` function of the package `coda` (we will talk more about this package a little later on). Do not let the term "rejection rate" confuse you; it is simply 1—acceptance rate. There may be parameters—for example, individual values of a random effect or latent variables—that you do not want to save, though, and in our next example we will show you a way to monitor their acceptance rates with a few extra lines of code.

17.3.3 Metropolis-within-Gibbs

One weakness of the MH sampler is that formulating the joint posterior when evaluating whether to accept or reject the candidate values for θ becomes increasingly complex or inefficient as the number of parameters in a model increases. As you already saw in Chapter 3, in these cases you can combine MH sampling and Gibbs sampling. You can use the principles of Gibbs sampling to break down your high-dimensional parameter space into easy-to-handle one-dimensional conditional distributions and use MH sampling for these conditional distributions. Better yet, if you have some conjugacy in your model, you can use the more efficient Gibbs sampling for these parameters and one-dimensional MH for all the others. You have already seen the basics of how to build both types of algorithms, so we can jump straight into an example here and build a Metropolis-within-Gibbs algorithm.

GLMMs: Poisson regression with a random effect. Let's a model that gets us closer to the problem we ultimately want to deal with—a GLMM. Here, we assume we have Poisson counts, y_{ij}, from $j = 1, 2, \ldots, n$ plots in i different study sites, and we believe that the counts are influenced by some plot-specific covariate, $\mathbf{x}$, but that there is also a random site effect. So our model is:

$$\begin{aligned} y_{ij} &\sim \text{Poisson}(\lambda_{ij}), \\ \lambda_{ij} &= \exp(\alpha_i + \beta x_{ij}). \end{aligned}$$

Let's place normal priors on α and β,

$$\alpha_i \sim \text{Normal}(\mu_\alpha, \sigma_\alpha)$$

and

$$\beta \sim \text{Normal}(\mu_\beta, \sigma_\beta).$$

In this model, we do not specify μ_α and σ_α, but instead, estimate them as well, so we have to specify hyperpriors for these parameters:

$$\begin{aligned} \mu_\alpha &\sim \text{Normal}(\mu_0, \sigma_0), \\ \sigma_\alpha^2 &\sim \text{Inverse-Gamma}(a_0, b_0). \end{aligned}$$

Note that for simplicity we assume that β is constant across the i study sites, and for analysis we set μ_β and σ_β (i.e., we don't estimate these parameters from the data). With the model completely specified, we can compile the full conditionals, breaking the multi-dimensional parameter space into one-dimensional components:

$$\begin{aligned} [\alpha_1 | \alpha_2, \alpha_3, \ldots, \alpha_i, \beta, \mathbf{y}_1] &\propto [\mathbf{y}_1 | \alpha_1, \beta][\alpha_1] \\ &\propto \text{Poisson}(\mathbf{y}_1 | \exp(\alpha_1 + \beta \mathbf{x}_1)) \times \text{Normal}(\alpha_1 | \mu_\alpha, \sigma_\alpha), \end{aligned}$$

where $\mathbf{y}_1 = (y_{11}, y_{12}, \ldots, y_{1n})$ is the vector of observed counts for site $i = 1$ and, in general, $\mathbf{y}_i$ is the vector of all counts for site i; analogously, $\mathbf{x}_i$ is the vector of all observations of the covariate for site i. The other full conditionals for each α_i are constructed similarly:

$$\begin{aligned}[\alpha_2|\alpha_1, \alpha_3, \ldots, \alpha_i, \beta, \mathbf{y}_2] &\propto [\mathbf{y}_2|\alpha_2, \beta][\alpha_2] \\ &\propto \text{Poisson}(\mathbf{y}_2|\exp(\alpha_2 + \beta\mathbf{x}_2)) \times \text{Normal}(\alpha_2|\mu_\alpha, \sigma_\alpha),\end{aligned}$$

and so on for all elements of $\boldsymbol{\alpha}$. The full conditional for β is:

$$\begin{aligned}[\beta|\boldsymbol{\alpha}, \mathbf{y}] &\propto [\mathbf{y}|\boldsymbol{\alpha}, \beta][\beta] \\ &\propto \text{Poisson}(\mathbf{y}|\exp(\boldsymbol{\alpha} + \beta\mathbf{x})) \times \text{Normal}(\beta|\mu_\beta, \sigma_\beta).\end{aligned}$$

Finally, we need to update the hyperparameters for the random effects vector $\boldsymbol{\alpha}$:

$$\begin{aligned}[\mu_\alpha|\boldsymbol{\alpha}] &\propto [\boldsymbol{\alpha}|\mu_\alpha, \sigma_\alpha][\mu_\alpha], \\ [\sigma_\alpha|\boldsymbol{\alpha}] &\propto [\boldsymbol{\alpha}|\mu_\alpha, \sigma_\alpha][\sigma_\alpha].\end{aligned}$$

Note that the likelihood contributions of the counts $\mathbf{y}$ at each site, when conditioned on $\boldsymbol{\alpha}$, do not depend on the hyperparameters μ_α and σ_α. As such, the full conditionals for these hyperparameters only depend on the collection of all $\boldsymbol{\alpha}$, not the data. Since we assumed $\boldsymbol{\alpha}$ to come from a normal distribution, the choice of priors for μ_α (normal) and σ_α^2 (inverse-gamma) leads to the same conjugacy we observed in our initial normal model, so that both hyperparameters can be updated using Gibbs sampling.

Now let's build the updating steps for these full conditionals. Again, for the MH steps that update $\boldsymbol{\alpha}$ and β we use normal proposal distributions with standard deviations δ_α and δ_β.

First, we set the initial values $\alpha^{(0)}$ and $\beta^{(0)}$. Then, starting with α_1, we draw $\alpha_1^{(1)}$ from Normal($\alpha_1^{(0)}, \delta_\alpha$), calculate the conditional posterior density of $\alpha_1^{(0)}$ and $\alpha_1^{(1)}$ and compare their ratios,

$$r = \frac{\text{Poisson}(\mathbf{y}_1|\exp(\alpha_1^{(1)} + \beta\mathbf{x}_1)) \times \text{Normal}(\alpha_1^{(1)}|\mu_\alpha, \sigma_\alpha)}{\text{Poisson}(\mathbf{y}_1|\exp(\alpha_1^{(0)} + \beta\mathbf{x}_1)) \times \text{Normal}(\alpha_1^{(0)}|\mu_\alpha, \sigma_\alpha)},$$

and accept $\alpha_1^{(1)}$ with probability $\min(r, 1)$. We repeat this for all $\boldsymbol{\alpha}$.

For β, we draw $\beta^{(1)}$ from Normal($\beta^{(0)}, \delta_\beta$), compare the posterior densities of $\beta^{(0)}$ and $\beta^{(1)}$,

$$r = \frac{\text{Poisson}(\mathbf{y}|\exp(\boldsymbol{\alpha} + \beta^{(1)}\mathbf{x})) \times \text{Normal}(\beta^{(1)}|\mu_\beta, \sigma_\beta)}{\text{Poisson}(\mathbf{y}|\exp(\boldsymbol{\alpha} + \beta^{(0)}\mathbf{x})) \times \text{Normal}(\beta^{(0)}|\mu_\beta, \sigma_\beta)},$$

and accept $\beta^{(1)}$ with probability $\min(r, 1)$.

For μ_α and σ_α^2, we sample directly from the full conditional distributions (Eq. (17.2.2) and Eq. (17.3.1)):

$$\mu_\alpha^{(1)} \sim \text{Normal}(\mu_n, \sigma_n^2),$$

where

$$\mu_n = \frac{\sigma_\alpha^{2(0)}}{\sigma_\alpha^{2(0)} + n_\alpha\sigma_0^2} \times \mu_0 + \frac{n_\alpha\sigma_0^2}{\sigma_\alpha^{2(0)} + n_\alpha\sigma_0^2} \times \bar{\alpha}^{(1)}$$

and

$$\sigma_n^2 = \frac{\sigma_\alpha^{2(0)}\sigma_0}{\sigma_\alpha^{2(0)} + n\sigma_0^2}.$$

Here, $\bar{\alpha}$ is the current mean of the vector $\boldsymbol{\alpha}$, which we updated before, and n_α is the length of $\boldsymbol{\alpha}$. For σ_α^2 we use

$$\sigma_\alpha^{2(1)} \sim \text{Inverse-Gamma}(a_n, b_n),$$

where

$$a_n = n_a/2 + a_0,$$

and

$$b_n = 0.5\sum_{i=1}^{n_\alpha}(\alpha_i^{(1)} - \mu_\alpha^{(1)})^2 + b_0.$$

We repeat these steps over T iterations of the MCMC algorithm. Call the function `PoisGLMM` in `scrbook` to check out what this algorithm looks like in **R**.

In this example we may not want to save each individual α_i, but are only interested in their mean and standard deviation. Since these two parameters will change as soon as the value for one element in $\boldsymbol{\alpha}$ changes, their acceptance rates will always be close to 1 and are not representative of how well your algorithm performs. To monitor the acceptance rates of parameters you do not want to save, you simply need to add a few lines of code into your updater to see how often the individual parameters are accepted. The code for updating $\boldsymbol{\alpha}$ from our Poisson GLMM below shows one way to monitor acceptance of individual α_i.

```
#initiate counter for acceptance rate of alpha
alphaUps <- 0

#loop over sites, update intercepts alpha one at a time;
#only data at site i contributes information
#lev is the number of sites i
for (i in 1:lev) {
      alpha.cand <- rnorm(1, alpha[i], delta_alpha)
      loglike <- sum(dpois (y[site==i], exp(alpha[i] + beta*x[site==i]),
        log=TRUE))
      logprior <- dnorm(alpha[i], mu_alpha,sig_alpha, log=TRUE)
      loglike.cand <- sum(dpois (y[site==i], exp(alpha.cand + beta *x[site==i]),
        log=TRUE))
      logprior.cand <- dnorm(alpha.cand, mu_alpha,sig_alpha, log=TRUE)
      if (runif(1)< exp((loglike.cand+logprior.cand) -(loglike+logprior))) {
      alpha[i] <- alpha.cand
      alphaUps <- alphaUps+1
```

```
     }
}

#lets you check the acceptance rate of alpha at every 100th iteration
if(iter %% 100 == 0) {
     cat("    Acceptance rates\n")
     cat("      alpha =", alphaUps/lev, "\n")
}
```

17.3.4 Rejection sampling and slice sampling

While MH and Gibbs sampling are probably the most widely applied algorithms for posterior approximation, there are other options that work under certain circumstances and may be more efficient when applicable. **WinBUGS** applies these algorithms and we want you to be aware that there are more algorithms out there for approximating posterior distributions than just Gibbs and MH. One alternative algorithm is rejection sampling. Rejection sampling is not an MCMC method, since each draw is independent of the others. The method can be used when the posterior $[\theta|y]$ is not a known parametric distribution but can be expressed in closed form. Then, we can use a so-called envelope function, say, $g(\theta)$, that we can easily sample from, with the restriction that $[\theta|y] < Mg(\theta)$. We then sample a candidate value for θ from $g(\theta)$, calculate $r = [\theta|y]/Mg(\theta)$, and keep the sample with the probability r. M is a constant that has to be picked so that r lies between 0 and 1, for example by evaluating both $[\theta|y]$ and $g(\theta)$ at n points and looking at their ratios. Rejection sampling only works well if $g(\theta)$ is similar to $[\theta|y]$, and packages like **WinBUGS** use adaptive rejection sampling (Gilks and Wild, 1992), where a complex algorithm is used to fit an adequate and efficient $g(\theta)$ based on the first few draws. Though efficient in some situations, rejection sampling does not work well with high-dimensional problems, since it becomes increasingly hard to define a reasonable envelope function. For an example of rejection sampling in the context of SCR models, see Chapter 11, where we use it to simulate inhomogeneous point processes.

Another alternative is slice sampling (Neal, 2003). In slice sampling, we sample uniformly from the area under the plot of $[\theta|y]$. Consider a single univariate parameter. Let's define an auxiliary variable, $U \sim$ ~~Unif~~$(0, [\theta|y])$. Then, θ can be sampled from the vertical slice of $[\theta|y]$ at U (Figure 17.4):

$$\theta|U \sim \cancel{\text{Unif}}(B),$$

where $B = \{\theta : [\theta|y] \geq U\}$

Slice sampling can be applied in many situations; however, implementing an efficient slice sampling procedure can be complicated. We refer the interested reader to Robert and Casella (2010, Chapter 7) for a simple example. Both rejection sampling and slice sampling can be applied on one-dimensional conditional distributions within a Gibbs sampling setup.

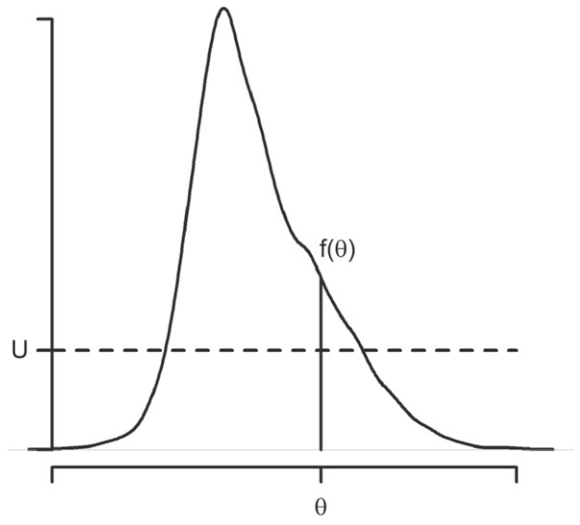

FIGURE 17.4

Slice sampling. For $U \sim$ Uniform(0,$[\theta|y]$), we can sample θ from the vertical slice of $[\theta|y]$ at U; $\theta|U \sim$ Uniform(B), where $B = \{\theta : [\theta|y] \geq U\}$.

17.4 MCMC for closed capture-recapture model M_h

By now you have seen MCMC samplers for some simple generalized (mixed) linear models. Now, to ease you into more complex models, we construct our own MCMC algorithm using a Metropolis-within-Gibbs sampler for the non-spatial model with individual heterogeneity in capture probability, model M_h, developed in Chapter 4.

To recapitulate: Under the non-spatial model, each of the n observed individuals is either detected ($y = 1$) or not ($y = 0$) during each of K sampling occasions. We estimate N using data augmentation and have a Bernoulli model for the data augmentation variables z_i

$$z_i \sim \text{Bernoulli}(\psi).$$

The binomial observation model is expressed conditional on the latent variables z_i

$$y_i \sim \text{Binomial}(K, z_i p_i).$$

Further, we prescribe a distribution for the capture probability p_i. Here we assume

$$\text{logit}(p_i) \sim \text{Normal}(\mu_p, \sigma_p^2).$$

As usual, we have to go through two general steps before we write the MCMC algorithm:

1. Identify the model with all its components (including priors).

2. Recognize and express the full conditional distributions for all parameters.

Our model components are as follows: $[y_i|p_i, z_i]$, $[p_i|\mu_p, \sigma_p]$, and $[z_i|\psi]$ for *each* $i = 1, 2, \ldots, M$, and then prior distributions $[\mu_p]$, $[\sigma_p]$, and $[\psi]$. The joint posterior distribution of all unknown quantities in the model is proportional to the joint distribution of all elements y_i, p_i, z_i, and also the prior distributions of the prior parameters:

$$\left\{\prod_{i=1}^{M}[y_i|p_i, z_i][p_i|\mu_p, \sigma_p][z_i|\psi]\right\}[\mu_p, \sigma_p, \psi].$$

For prior distributions, we assume that $\mu_p, \sigma_p,$ and ψ are mutually independent, and for μ_p and σ_p we use improper uniform priors; finally, $\psi \sim \text{Uniform}(0, 1)$. This is equivalent to Beta(1, 1), which will come in handy, as we will see in a moment. Note that the likelihood contribution for each individual, when conditioned on p_i and z_i, does not depend on ψ, μ_p, or σ_p. As such, the full conditional for the structural parameter ψ only depends on the collection of data augmentation variables z_i, and that for μ_p and σ_p will only depend on the collection of latent variables $p_i; i = 1, 2, \ldots, M$ (this is equivalent to what we saw in the Poisson regression with random intercept $\boldsymbol{\alpha}$, where hyperparameters for the distribution of $\boldsymbol{\alpha}$ did not depend on the observed data). The full conditionals for all the unknowns are as follows:

1. For p_i:

$$\begin{aligned}[p_i|y_i, \mu_p, \sigma_p, z_i] &\propto [y_i|p_i][p_i|\mu_p, \sigma_p] \text{ if } z_i = 1\\ &\quad [p_i|\mu_p, \sigma_p] \text{ if } z_i = 0.\end{aligned}$$

2. for z_i:

$$[z_i|y_i, p_i, \psi] \propto [y_i|z_i p_i]\text{Bernoulli}(z_i|\psi).$$

3. For μ_p:

$$[\mu_p|p_i, \sigma_p] \sim \left\{\prod_i [p_i|\mu_p, \sigma_p]\right\} \times \text{const.}$$

4. For σ_p:

$$[\sigma_p|p_i, \mu_p] \sim \left\{\prod_i [p_i|\mu_p, \sigma_p]\right\} \times \text{const.}$$

5. For ψ:

$$[\psi|z_i] \propto \left\{\prod_i [z_i|\psi]\right\} \text{Beta}(1, 1).$$

Remember that Beta(1,1) is equivalent to Uniform(0,1). The beta distribution is the conjugate prior to the binomial and Bernoulli distributions, and the general form of a full conditional of a beta-binomial model with sample size n, $x_i \sim \text{Bernoulli}(\theta)$ and

$\theta \sim \text{Beta}(a, b)$ is

$$[\theta|\mathbf{x}] \propto \text{Beta}(a + \sum_i x_i, b + n - \sum_i x_i).$$

In our case that means

$$[\psi|z_i] \propto \text{Beta}(1 + \sum z_i, 1 + M - \sum z_i).$$

What we've done here is identify each of the full conditional distributions in sufficient detail to toss them into our Metropolis-Hastings algorithm. The constant terms in the full conditionals for μ_p and σ_p reflect the improper prior we chose for both parameters. Because of the choice of an improper prior, prior probability densities for both parameters $\propto 1$, i.e., constant, and these constants cancel out of the MH acceptance ratio (see updating step below and following example). Below, you see the updating step for the detection parameter **p**. Note that (1) we draw candidate values on the logit scale and (2) instead of looping through $1 - M$ individuals to update all p_i, we update all elements of the vector of **p** in parallel, for computational efficiency.

```
### update the logit(p) parameters
lp.cand <- rnorm(M,lp,1) # 1 is a tuning parameter
p.cand <- plogis(lp.cand)
ll <- dbinom(ytot,K,z*p, log=T)
prior <- dnorm(lp,mu,sigma, log=T)
llcand <- dbinom(ytot,K,z*p.cand, log=T)
prior.cand <- dnorm(lp.cand,mu,sigma, log=T)

kp <- runif(M) < exp((llcand+prior.cand)-(ll+prior))
p[kp] <- p.cand[kp]
lp[kp] <- lp.cand[kp]
```

The parameters μ_p and σ_p are also updated using MH steps (see the code for μ_p below). In truth, we could also sample μ_p and σ_p^2 directly with certain choices of prior distributions. For example, if $\mu_p \sim \text{Normal}(0, 1000)$ then the full conditional for μ_p is also normal (see Section 17.3.1), etc.:

```
p0.cand<- rnorm(1,p0,.05)
if(p0.cand>0 & p0.cand<1){
    mu.cand<-log(p0.cand/(1-p0.cand))
    ll<-sum(dnorm(lp,mu,sigma,log=TRUE))
    llcand<-sum(dnorm(lp,mu.cand,sigma,log=TRUE))
    if(runif(1)<exp(llcand-ll)) {
     mu<-mu.cand
     p0<-p0.cand
    }
}
```

For ψ we can easily sample directly from the beta distribution:

```
psi <- rbeta(1, sum(z) + 1, M-sum(z) + 1)
```

To update the z_i we have opted for a MH updater (although they could be updated directly from their full conditional). Since z_i can only take the values of 0 or 1, we generate candidate values using `z.cand<-ifelse(z==1,0,1)`. The updating step for z_i is detailed in the next example. You can check out the full code by invoking `modelMh` from the **R** package `scrbook`.

17.5 MCMC algorithm for model SCR0

Conceptually, but also in terms of MCMC coding, it is only a small step from the non-spatial model M_h to a fully spatial capture-recapture model. Next, we will walk you through the steps of building your own MCMC sampler for the basic SCR model (i.e., without any individual-, site-, or time-specific covariates) with both a Poisson and a binomial encounter process. As usual, we will have to go through two general steps before we write the MCMC algorithm:

1. Identify the model with all its components (including priors) .
2. Recognize and express the full conditional distributions for all parameters.

It is worthwhile to go through all of Step 1 for an SCR model, but you have probably seen enough of Step 2 in our previous examples to get the essence of how to express a full conditional distribution. Therefore, we will exemplify Step 2 for some parameters and tie these examples directly to the respective **R** code snippets.

Step 1—Identify your model: Recall the components of the basic SCR model with a Poisson encounter process from Chapter 9: We assume that individuals i, or rather, their activity centers $\mathbf{s}_i$, are uniformly distributed across the state-space $\mathcal{S}$,

$$\mathbf{s}_i \sim \text{Uniform}(\mathcal{S}),$$

and that the number of times individual i encounters trap j, y_{ij}, is a Poisson variable with mean λ_{ij},

$$y_{ij} \sim \text{Poisson}(\lambda_{ij}).$$

The link between individual location, movement, and trap encounter rates is made by the assumption that λ_{ij} is a decreasing function of the distance between $\mathbf{s}_i$ and the location of j, $\mathbf{x}_j$, say

$$d_{ij} = ||\mathbf{x}_j - \mathbf{s}_i||,$$

of the Gaussian (or half-normal) form

$$\lambda_{ij} = \lambda_0 \exp(-d_{ij}^2/2\sigma^2),$$

where λ_0 is the baseline trap encounter rate at $d_{ij} = 0$ and σ is the scale parameter of the half-normal function.

As in the non-spatial example for model M_h, we estimate N, here the number of $\mathbf{s}_i$ in $\mathcal{S}$, using data augmentation (Section 4.2). We create $M - n$ all-zero encounter histories and estimate N by summing over the auxiliary data augmentation variables, z_i, which we assume are Bernoulli random variables,

$$z_i \sim \text{Bernoulli}(\psi).$$

To link the two model components, we modify our trap encounter model to

$$\lambda_{ij} = \lambda_0 \exp(-d_{ij}^2/2\sigma^2) z_i.$$

The model has the following structural parameters, for which we need to specify priors:

ψ	the Uniform(0, 1) is required as part of the data augmentation procedure and in general is a natural choice of an uninformative prior for a probability. It will also lead to conjugacy as we saw in the example of model M_h, so that we can update ψ directly from its full conditional distribution using Gibbs sampling
$\mathbf{s}_i$	since $\mathbf{s}_i$ is a pair of coordinates it is two-dimensional and we use a uniform prior limited by the extent of our state-space over both dimensions
σ	we can conceive several priors for σ but let's assume an improper prior, one that is Uniform over $(0, \infty)$. As we already saw, this choice is convenient when updating the parameter, because the constant prior probability cancels out of the MH acceptance ratio
λ_0	analogously, we will use a Uniform$(0, \infty)$ improper prior for λ_0

Step 2—Construct the full conditionals: Having completed Step 1, let's look at the full conditional distributions for some of these parameters. We saw that with improper priors, full conditionals are proportional only to the likelihood of the observations; for example, consider σ:

$$[\sigma|\mathbf{s}, \lambda_0, \mathbf{z}, \mathbf{y}] \propto \left\{ \prod_i [y_i|\mathbf{s}_i, \lambda_0, z_i, \sigma] \right\}.$$

The **R** code to update σ is shown below. Notice that we automatically reject negative candidate values, since σ cannot be <0:

```
sig.cand <- rnorm(1, sigma, 0.1) #draw candidate value
 if(sig.cand>0){ #automatically reject sig.cand that are <0
```

```
   lam.cand <- lam0*exp(-(d*d)/(2*sig.cand*sig.cand))
   ll <- sum(dpois(y, lam*z, log=TRUE))
   llcand <- sum(dpois(y, lam.cand*z, log=TRUE))
   if(runif(1) < exp(llcand - ll)){
    ll <- llcand
    lam <- lam.cand
    sigma <- sig.cand
   }
}
```

These steps are analogous for λ_0 and $\mathbf{s}_i$, and we will use MH steps for all of these parameters. Similar to the random intercepts in our Poisson GLMM, we update each $\mathbf{s}_i$ individually. Note that to be fully correct, the full conditional for $\mathbf{s}_i$ contains both the likelihood and prior component, since we did not specify an improper, but a proper uniform prior on $\mathbf{s}_i$. However, with a uniform distribution the probability density of any value is 1/(upper limit – lower limit) = constant. Thus, the prior components are identical for both the current and the candidate value so that when you calculate the ratio of posterior densities, r, the identical prior components appear both in the numerator and denominator and cancel each other out.

We still have to update z_i. The full conditional for z_i is

$$[z_i|y_i, \sigma, \lambda_0, \mathbf{s}_i] \propto [y_i|z_i, \sigma, \lambda_0, \mathbf{s}_i][z_i],$$

and since $z_i \sim \text{Bernoulli}(\psi)$, the term has to be taken into account when updating z_i:

```
zUps <- 0 #set counter to monitor acceptance rate
for(i in 1:M) {
#no need to update seen individuals, since their z =1
 if(seen[i])
   next
 zcand <- ifelse(z[i]==0, 1, 0)
 llz <- sum(dpois(y[i,],lam[i,]*z[i], log=TRUE))
 llcand <- sum(dpois(y[i,], lam[i,]*zcand, log=TRUE))

 prior <- dbinom(z[i], 1, psi, log=TRUE)
 prior.cand <- dbinom(zcand, 1, psi, log=TRUE)
 if(runif(1) < exp((llcand+prior.cand)-(llz+prior))){
  z[i] <- zcand
  zUps <- zUps+1
 }
}
```

The parameter ψ is a hyperparameter of the model, with an uninformative prior distribution of Uniform(0, 1) or Beta(1, 1), so that

$$[\psi|\mathbf{z}] \propto \text{Beta}(1 + \sum_i z_i, 1 + M - \sum_i z_i).$$

These are all the building blocks you need to write the MCMC algorithm for the spatial null model with a Poisson encounter process. You can find the full **R** code by calling the function `SCR0pois` in the **R** package `scrbook`:

17.5.1 SCR model with binomial encounter process

The equivalent SCR model with a binomial encounter process is very similar. Here, each individual i can only be detected once at any given trap j during a sampling occasion k. Thus,

$$y_{ij} \sim \text{Binomial}(K, p_{ij}),$$

where p_{ij} is some function of distance between $\mathbf{s}_i$ and trap location $\mathbf{x}_j$. Here we use:

$$p_{ij} = 1 - \exp(-\lambda_{ij}).$$

Recall from Chapter 3 that this is the complementary log-log (cloglog) link function, which constrains p_{ij} to fall between 0 and 1. For our MCMC algorithm that means that, instead of using a Poisson likelihood, Poisson$(y|\sigma, \lambda_0, \mathbf{s}, z)$, we use a binomial likelihood, Binomial$(y|\sigma, \lambda_0, \mathbf{s}, z; K)$, in all the conditional distributions. An exemplary updating step for λ_0 under a binomial encounter model is shown below. The full MCMC code for the binomial SCR with a cloglog link (`SCR0binom.cl`) can be found in the **R** package `scrbook`:

```
  lam0.cand <- rnorm(1, lam0, 0.1)
  #automatically reject lam0.cand that are <0
  if(lam0.cand >0){
  lam.cand <- p0.cand*exp(-(d*d)/(2*sigma*sigma))
  p.cand <- 1-exp(-lam.cand)
  ll<- sum(dbinom(y, K, pmat *z, log=TRUE))
  llcand <- sum(dbinom(y, K, p.cand *z, log=TRUE))
  if(runif(1) < exp(llcand - ll)){
      ll<-llcand
      pmat <- p.cand
      lam0 <- lam0.cand
  }
}
```

Another possibility is to model variation in the individual- and site-specific detection probability, p_{ij}, directly, without any transformation, such that

$$p_{ij} = p_0 \exp(-d_{ij}^2/(2\sigma^2))$$

and $p_0 \in [0, 1]$. This formulation is analogous to how detection probability is modeled in distance sampling under a half-normal detection function; however, in distance sampling p_0—detection of an individual on the transect line—is assumed to be 1 (Buckland et al., 2001). Under this formulation the updater for p_0 becomes:

```
 p0.cand <- rnorm(1, p0, 0.1)
 if(p0.cand > 0 & p0.cand < 1){
     #automatically rejects p0.cand that are not {0,1}
      p.cand <- p0.cand*exp(-(d*d)/(2*sigma*sigma))
      ll <- sum(dbinom(y, K, pmat *z, log=TRUE))
      llcand <- sum(dbinom(y, K, p.cand *z, log=TRUE))
      if(runif(1) < exp(llcand - ll)){
         ll <- llcand
           pmat <- p.cand
           p0 <- p0.cand
         }
    }
```

17.6 Looking at model output

Now that you have an MCMC algorithm to analyze spatial capture-recapture data with, let's run an actual analysis so we can look at the output. As an example, we will use the Fort Drum bear data set we first introduced in Chapter 1 and already analyzed in several preceding chapters. You can load the Fort Drum data (`data(beardata)`), extract the trap locations (`trapmat`) and detection data (`bearArray`), and build the augmented $M \times J$ array of individual encounter histories:

```
> M = 700
> trapmat <- beardata$trapmat
#summarizes captures across occasions
> bearmat <- apply(beardata$bearArray, 1:2, sum)
> Xaug <- matrix(0, nrow=M, ncol=dim(trapmat)[1])
> Xaug[1:dim(bearmat)[1],] <- bearmat   #create augmented data set
```

In addition to these data, we need to specify the outermost coordinates of the state-space. Since bears are wide-ranging animals we add a 20-km buffer to the maximum and minimum coordinates of the trap array:

```
> xl <- min(trapmat[,1])- 20
> yl <- min(trapmat[,2])- 20
> xu <- max(trapmat[,1])+ 20
> yu <- max(trapmat[,2])+ 20
```

Finally, use the MCMC code for the binomial encounter model with the cloglog link (`SCR0binom.cl`) and run 5000 iterations. This should take approximately 25 min (in real life we would, of course, run the algorithm a lot longer but for demonstration purposes let's stick to a number of iterations that can be run in a manageable amount of time):

```
> set.seed(13)
> mod0 <- SCR0binom.cl(y=Xaug, X=trapmat, M=M, xl=xl, xu=xu, yl=yl,
+                     yu=yu, K=8, delta=c(0.1, 0.05, 2), niter=5000)
```

Before, we used simple **R** commands to look at model results. However, there is a specific **R** package to summarize MCMC simulation output and perform some convergence diagnostics—package `coda` (Plummer et al., 2006). Download and install `coda`, then convert your model output to an mcmc object:

```
> chain <- mcmc(mod0)
```

which can be used by `coda` to produce MCMC-specific output.

17.6.1 Markov chain time series plots

Start by looking at time series plots of your Markov chains using `plot(chain)`. This command produces a time series plot and marginal posterior density plots for each monitored parameter, similar to what we did before using the `hist` and `plot` commands. Figure 17.5 shows an example of these plots for σ and λ_0. Time series plots will tell you several things: First, recall from Section 17.3.2 that the way the chains move through the parameter space gives you an idea of whether your MH steps are well tuned. If chains were constant over many iterations you would need to decrease the tuning parameter of the (normal) proposal distribution. If a chain moves along some gradient to a stationary state very slowly, you may want to increase the tuning parameter so that the parameter space is explored more efficiently.

Second, you will be able to see if your chains converged and how many initial simulations you have to discard as burn-in. In the case of the chains shown in Figure 17.5, we would probably consider the first 750–1,000 iterations as burn-in, as afterwards the chains seem to be fairly stationary.

17.6.2 Posterior density plots

The `plot` command also produces posterior density plots and it is worthwhile to look at those carefully. For parameters with priors that have bounds (e.g., uniform over some interval), you will be able to see if your choice of the prior is truncating the posterior distribution. In the context of SCR models, this will mostly involve our choice of M, the size of the augmented data set. If the posterior of N has a lot of mass concentrated close to M (or equivalently, the posterior of ψ has a lot of mass concentrated close to 1), as in the example in Figure 17.6, we have to re-run the analysis with a larger M. A diffuse posterior plot suggests that the parameter may not be well identified. There may not be enough information in your data to estimate model parameters and you may have to consider a simpler model, or your model may contain parameters that are confounded and therefore not identifiable. Finally, posterior density plots will show you if the posterior distribution is symmetrical or skewed—if the distribution has a heavy tail, using the mean as a point estimate of your parameter of interest may be biased and you may want to opt for the median or mode instead.

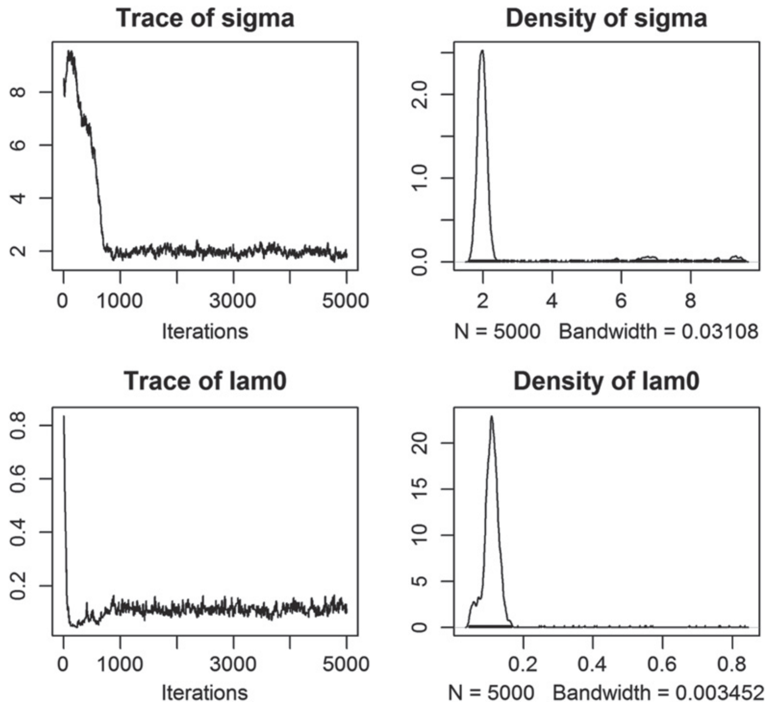

FIGURE 17.5

Time series and posterior density plots for σ and λ_0 for the Fort Drum black bear data.

17.6.3 Serial autocorrelation and effective sample size

Checking the degree of autocorrelation in the Markov chains and estimating the effective sample size the chain has generated should be part of evaluating your model output. If you use **WinBUGS** through the `R2WinBUGS` package, the `print` command will automatically return the effective sample size for all monitored parameters. In the `coda` package there are several functions you can use to do so. The function `effectiveSize` will directly give you an estimate of the effective sample size for the parameters:

```
> effectiveSize(window(chain, start=1001))
    sigma       lam0       psi          N
 93.89807  163.72311  51.96443   46.45394
```

Alternatively, you can use the `autocorr.diag` function, which will show you the degree of autocorrelation for different lag values (which you can specify within the function call, we use the defaults below):

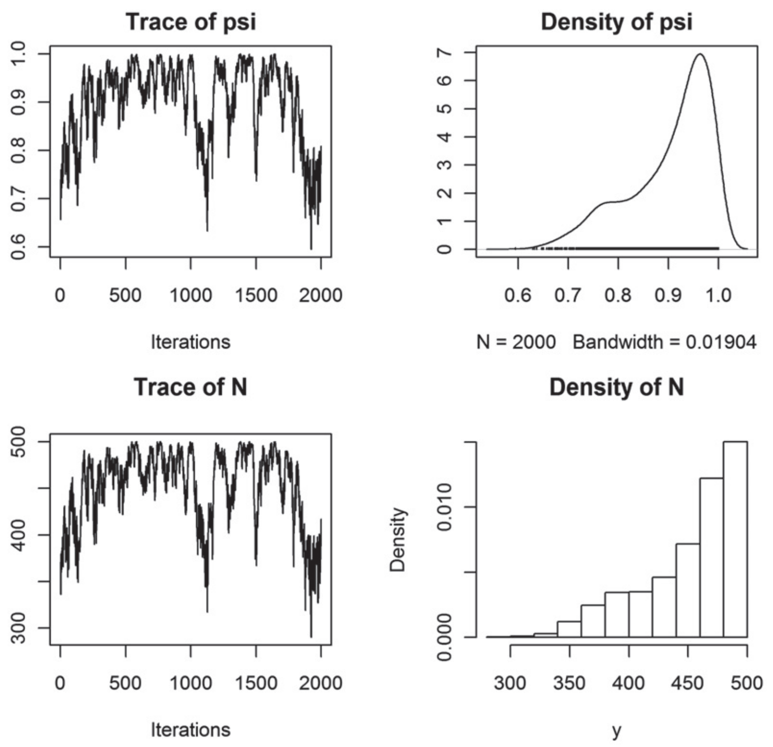

FIGURE 17.6

Time series and posterior density plots of ψ and *N* for the Fort Drum black bear data truncated by the upper limit of *M* (500).

```
> autocorr.diag(window(chain, start=1001))
             sigma        lam0        psi          N
 Lag 0   1.0000000  1.00000000  1.0000000  1.0000000
 Lag 1   0.9316928  0.91464875  0.9745833  0.9663320
 Lag 5   0.7603332  0.67445407  0.8525272  0.8500215
 Lag 10  0.6065374  0.48724122  0.7514657  0.7530124
 Lag 50  0.1122331  0.06564406  0.3811939  0.3823236
```

In the present case we see that autocorrelation is especially high for the parameter ψ and effective sample size for this parameter is only 52! This means we would have to run the model for much longer to obtain a reasonable effective sample size. For now, let's continue using this small number of samples to look at the output.

17.6.4 Summary results

Now that we checked that our chains apparently have converged and pretending that we have generated enough samples from the posterior distribution, we can look at the actual parameter estimates. The `summary` function will return two sets of results: the mean parameter estimates, with their standard deviation, the naive standard error—i.e., your regular standard error calculated for T (= number of iterations) samples without accounting for serial autocorrelation—and the time-series SE (in **WinBUGS** and earlier in this book referred to as MC error), which accounts for autocorrelation. Remember that this error decreases with increasing chain length and should be 1% or less of the parameter estimate. In **WinBUGS** the MC error can be extracted from a model object created with the `bugs` call, say `mod`, by using `mod$summary`. You should adjust the `summary` call by removing the burn-in from calculating parameter summary statistics. To do so, use the `window` command, which lets you specify at which iteration to start "counting." In contrast to **WinBUGS**, which requires you to set the burn-in length before you run the model, this command gives us full flexibility to make decisions about the burn-in after we have seen the trajectories of our Markov chains. For our example, `summary(window(chain, start=1001))` returns the following output:

```
Iterations = 1001:5000
Thinning interval = 1
Number of chains = 1
Sample size per chain = 4000

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:
              Mean        SD   Naive SE  Time-series SE
  sigma     1.9697   0.12534  0.0019818        0.012792
  lam0      0.1124   0.01521  0.0002405        0.001311
  psi       0.7295   0.11794  0.0018648        0.015278
  N       510.9190  81.99868  1.2965130       10.580567
2. Quantiles for each variable:
              2.5%       25%       50%       75%     97.5%
  sigma     1.7288    1.8831    1.9666    2.0517    2.2240
  lam0      0.0863    0.1008    0.1112    0.1217    0.1449
  psi       0.5100    0.6423    0.7261    0.8170    0.9549
  N       359.0000  451.0000  508.0000  572.0000  668.0000
```

Looking at the MC errors (the column labeled `Time-series SE`), we see that in spite of the high autocorrelation, the MC error for σ is below the 1% threshold, whereas for all other parameters, MC errors are still above, another indication that for a thorough analysis we should run a longer chain.

Our algorithm gives us a posterior distribution of N, but we are usually interested in the density, D. Density itself is not a parameter of our model, but we can derive a posterior distribution for D by dividing each value of N (N at each iteration) by the area of the state-space (here 3032.719 km^2) and we can use summary statistics of the resulting distribution to characterize D:

```
> summary(window(chain[,4]/ 3032.719, start=1001))

Iterations = 1001:5000
Thinning interval = 1
Number of chains = 1
Sample size per chain = 4000

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

         Mean          SD   Naive SE  Time-series SE
    0.1684690   0.0270380  0.0004275       0.0034888

2. Quantiles for each variable:

      2.5%      25%      50%      75%    97.5%
    0.1184   0.1487   0.1675   0.1886   0.2203
```

We see that the mean density of $0.17/\text{km}^2$ is very similar to the estimate of $0.18/\text{km}^2$ obtained under the non-spatial model M_0 in Chapter 4.

17.6.5 Other useful commands

While inspecting the time series plot gives you a first idea of how well you tuned your MH algorithm, use `rejectionRate` to obtain the rejection rates (1—acceptance rates) of the parameters that are written to your output:

```
> rejectionRate(chain)
      sigma        lam0         psi           N
 0.42988598  0.78775755  0.00000000  0.03160632
```

Rejection rates should lie between 0.2 and 0.8 (Section 17.3.2), so our tuning seems to have been appropriate here. Draws of the parameter ψ are never rejected since we update it with Gibbs sampling, where all candidate values are kept. And since N is the sum of all z_i, all it takes for N to change from one iteration to the next are small changes in the z-vector, so the rejection rate of N is always low. If you have run several parallel chains, you can combine them into a single mcmc object using the `mcmc.list` command on the individual chains (note that each chain has to be converted to an mcmc object before combining them with `mcmc.list`). You can then easily obtain the Gelman-Rubin diagnostic (Gelman et al., 2004), in **WinBUGS** called Rhat, using `gelman.diag`, which will indicate if all chains have converged to the same stationary distribution (see Section 17.8.1 for an example). For details on these and other functions, see the `coda` manual, which can be found (together with the package) on CRAN.

17.7 Manipulating the state-space

So far, we have constrained the location of the activity centers to fall within the outermost coordinates of our rectangular state-space by posing upper and lower bounds for X and Y. But what if $\mathcal{S}$ has an irregular shape—maybe there is a large water body we would like to remove from $\mathcal{S}$, because we know our terrestrial study species does not occur there. Or the study takes place in a clearly defined area such as an island.

As mentioned before, this situation is difficult to handle in **BUGS** engines. In some simple cases we can adjust the state-space by setting one of the coordinates of $\mathbf{s}_i$ to be some function of the other and reject candidate $\mathbf{s}_i$ that do not fall within this modified state-space. In this manner, we can cut off corners of the rectangle to approximate the actual state-space.[3] To illustrate this approach, plot the following rectangle, representing your state-space polygon, and line, representing, for example, the approximation of a shore line:

```
> xlim <- c(-5,5)
> ylim <- c(-7,7)
> plot(xlim, ylim, type='n')
> abline(a=4, b=0.4)
```

The Y coordinates limiting your state-space to the habitat that is suitable to the species you study can now be expressed as a linear function of the X coordinates, in this case, $Y = 4 + 0.4 \times X$. To include this new limit in a **BUGS** model, we need to change the following:

```
#draw SX and SY as before
SX[i] ~ dunif(xlim[1],xlim[2])
SY[i] ~ dunif(ylim[1],ylim[2])
#calculate upper limit for Y given X
ymax[i] <- 4+0.4*SX[i]
# use step function to see if location [SX, SY]
# is below the Y limit (Pin = 1) or not (Pin = 0)
Pin[i] <- step(ymax[i] - SY[i])
In[i] ~ dbern(Pin[i])
```

The object `In` is a vector of M 1s, passed as data to the model. If Pin = 0, the likelihood will be 0 and the candidate [SX, SY] pair will be rejected. If Pin = 1, this bit of the likelihood is equal to 1, and whether or not the candidate pair of coordinates is accepted depends only on capture history of i. This approach can be very useful in some situations but is clearly restricted by the functional form of the relationship between SX and SY that it requires.

[3] This idea was pitched to us by Mike Meredith, Biodiversity Conservation Society Sarawak and WCS Malaysia.

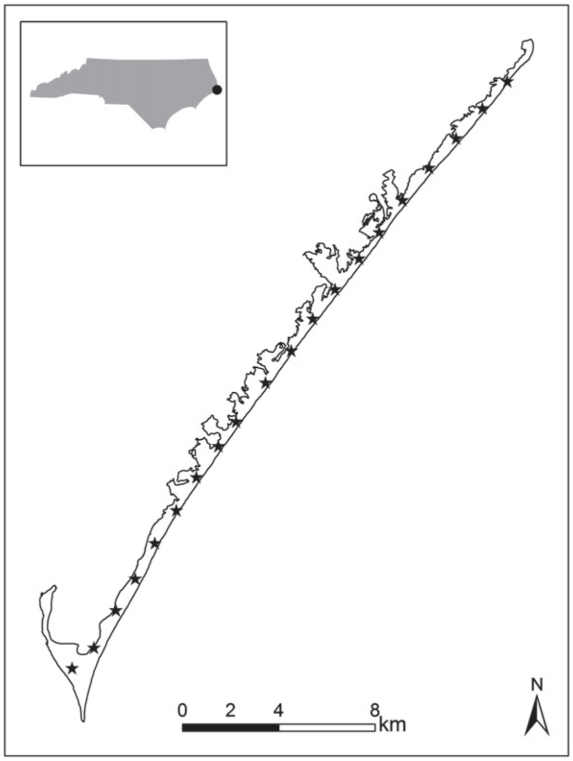

FIGURE 17.7

Camera traps (stars) set up on South Core Banks, a barrier island within Cape Lookout National Seashore, North Carolina (inset map) to estimate the raccoon population (see Chapter 19 for details).

In **R**, we are much more flexible, as we can use the actual state-space polygon to constrain $\mathbf{s}_i$. To illustrate that, let's look at a camera trapping study of raccoons (*Procyon lotor*) conducted on South Core Banks, a barrier island within Cape Lookout National Seashore, North Carolina (details of the study can be found in Sollmann et al. (2013a) and in Chapter 19 where we present the analysis of this data set with spatial mark-resight models). Since camera traps were spread across the entire length of the island, we set the state-space to be delineated by the shore line of the island (Figure 17.7), which clearly cannot easily be approximated as a rectangle. Instead, within **R** we can use an actual shapefile of the island.

In other circumstances you may still want to create the state-space as before, by adding some buffer to your trapping grid, but you may find that the resulting rectangle includes water bodies, paved parking lots, or any other kind of habitat you know is never used by the species you study. In order to precisely describe the state-space, these features need to be removed. You can create a precise state-space polygon in

ArcGIS and read it into **R**, or create the polygon directly within **R**, by intersecting two shapefiles—one of the rectangle defining the outer limits of your state-space and one of the landscape features you want to remove. While you will most likely have to obtain the shapefile describing the landscape of and around your trapping grid (coastlines, water bodies, etc.) from some external source, the polygon shapefile buffering your outermost trapping grid coordinates can easily be written in **R**.

If `xmin`, `xmax`, `ymin`, and `ymax` mark the most extreme X and Y coordinates of your trapping grid and b is the distance you want to buffer with, load the package `shapefiles` (Stabler, 2006) and issue the following **R** commands:

```
> xl = xmin-b
> xu = xmax+b
> yl = ymin-b
> yu = ymax+b

             #create data frame with coordinate pairs
> dd <- data.frame(Id=c(1,1,1,1,1),X=c(xl,xu,xu,xl,xl),
+  Y=c(yl,yl,yu,yu,yl))
> ddTable <- data.frame(Id=c(1),Name=c("Item1"))
            #convert to shapefile, type polygon
> ddShapefile <- convert.to.shapefile(dd, ddTable, "Id", 5)
            # name and save to location of choice
> write.shapefile(ddShapefile, 'c:/Test', arcgis=T)
```

You can read shapefiles into by **R** loading the package `maptools` (Lewin-Koh et al., 2011) and using the function `readShapeSpatial`. Make sure you read in shapefiles in UTM format, so that units of the trap array, the movement parameter σ, and the state-space are all identical. Intersection of polygons can be done in **R** also, using the package `rgeos` (Bivand and Rundel, 2011) and the function `gIntersect`. The area of your (single) polygon can be extracted directly from the state-space object `SSp`:

```
> area <- SSp@polygons[[1]]@Polygons[[1]]@area /1000000
```

Note that dividing by 1,000,000 will return the area in km^2 if your coordinates describing the polygon are in UTM. If your state-space consists of several disjoint polygons, you will have to sum the areas of all polygons to obtain the size of the state-space. To include this polygon into our MCMC sampler we need one last spatial **R** package, `sp` (Pebesma and Bivand, 2011), which has a function, `over`, which allows us to check if a pair of coordinates falls within a polygon or not.[4] All we have to do is embed this new check into the updating steps for the $\mathbf{s}_i$:

[4]Remember from Section 6.4.2 that the `over` function takes as its second argument (among others) an object of the class "SpatialPolygons" or "SpatialPolygonsDataFrame." The former produces a vector while the latter produces a data frame (e.g., in the example above), which is important for how you index the output.

```
     #draw candidate value
Scand <- as.matrix(cbind(rnorm(M, S[,1], 2), rnorm(M, S[,2], 2)))
      #convert to spatial points on UTM (m) scale
Scoord <- SpatialPoints(Scand*1000)
      # check if scand is within the polygon
SinPoly <- over(Scoord,SSp)

for(i in 1:M) {
    #if scand falls within polygon, continue update
   if(is.na(SinPoly[i]) == FALSE) {
... [rest of the updating step remains the same]
```

Note that it is much more time efficient to draw all M candidate values for $\mathbf{s}$ and check once if they fall within the state-space, rather than running the `over` command for every individual pair of coordinates. To make sure that our initial values for $\mathbf{s}$ also fall within the polygon of $\mathcal{S}$, we use the function `runifpoint` from the package `spatstat` (Baddeley and Turner, 2005), which generates random uniform points within a specified polygon. You'll find this modified MCMC algorithm (`SCR0poisSSp`) in the **R** package `scrbook`.

Finally, observe that we are converting candidate coordinates of $\mathcal{S}$ back to meters to match the UTM polygon. In all previous examples, for both the trap locations and the activity centers we have used UTM coordinates divided by 1000 to estimate σ on a km scale. This is adequate for wide-ranging species like bears. In other cases you may center all coordinates on 0. No matter what kind of transformation you use on your coordinates, make sure to always convert candidate values for $\mathcal{S}$ back to the original scale (UTM) before running the `over` command.

17.8 Increasing computational speed

Using custom written MCMC algorithms in **R** is not only more flexible but can also be faster than using programs such as **JAGS** and especially **WinBUGS**. Also, **R** tends to use much less memory than **JAGS**, which can be crucial if you are running a large model but only have limited memory available. **WinBUGS** is limited in the amount of memory it can access and thus will likely not max out your memory, but as a trade-off, it will take a long time to run such models. In this chapter we have provided you with the guidelines to write your own MCMC sampler. But beyond the material that we have covered there are a number of ways you can make your sampler more efficient, through parallel computing or by accessing an alternative computer language such as **C++**. Exploring these options exhaustively is beyond the scope of this book; instead, in this section we will give you some pointers to get started with these more advanced computational issues.

17.8.1 Parallel computing

If you are using a computer with several cores, you can make use of parallel computing to speed up overall computation. In parallel computing we execute commands simultaneously on different cores of the computer, instead of running them serially on one single core. For example, imagine you have 4 cores available and you want to implement a for-loop in **R**; instead of going through the loop iteration by iteration, you can prompt **R** to execute iterations 1–4 simultaneously on the four different cores. The core that finishes first will then continue with iteration 5, and so on. There are several packages in **R** that allow you to induce parallel computing, such as `snow` (Tierney et al., 2011) and `snowfall` (Knaus, 2010), and the more current versions of **R** (from 2.14.0 upwards) come with a pre-installed set of functions grouped under the name `parallel`.

The MCMC algorithms developed here and in other parts of this book come with plenty of opportunities to parallelize computation. In various instances within the algorithm, we have for-loops across our augmented data set of size M, or we may have for-loops across sampling occasions. We also have for-loops across iterations of the algorithm, but since one iteration of the Markov chain depends on the preceding iteration these should always be run serially, not in parallel. There is another dimension we can think of, and that is running multiple chains of an algorithm to assess convergence. This is a comparatively easy implementation of parallel computing and thus provides a good starting point to understand how it works in **R**.

Let's go back to the ~~Ft.~~ Drum black bear data we analyzed above with the cloglog version of the binomial SCR model (Section 17.6) and run three parallel chains using `snowfall`. All we need to do is wrap our function `SCR0binom.cl` within another function that can then be executed in parallel, returning a list with one output matrix for each chain (install `snowfall` before executing the code below; we assume the data objects are already in your workspace from the previous analysis):

```
> library(snowfall)
## create wrapper function
> wrapper <- function(a){
+ out <- SCR0binom.cl(y=Xaug, X=trapmat, M=M, xl=xl, xu=xu, yl=yl,
+                    yu=yu, K=8, delta=c(0.1, 0.05, 2), niter=5000)
+ return(out)
+ }
```

After creating the wrapper function we need to initialize the cluster of cores, defining that we want computation to be implemented in parallel and how many cores we want it to be run on. Here, we assume we have (at least) 3 cores, but if your computer only has 2, make sure to adjust the code accordingly (i.e., set `cpus=2`). In that case, 2 of the 3 chains will be run in parallel and whichever core finishes first will then pick up the third chain. Further, we have to export all **R** libraries and data to all the cores, and set up a random number generator, so that we do not get identical results from the different cores:

```
> sfInit( parallel=TRUE, cpus=3) #initialize cluster
> sfLibrary(scrbook) #export library scrbook
> sfExportAll() #export all data in current workspace
> sfClusterSetupRNG() #set up random number generator
> outL = sfLapply(1:3,wrapper) # execute 'wrapper' 3 times
```

The object `outL` is a list of length 3, with one `out` matrix from the function `SCR0binom.cl` for each chain. After computation is complete, terminate the cluster using the command `sfStop`. Note that the intermediate output of current values and acceptance rates in the **R** console is suppressed when using parallel computing. We can now look at the output as described previously using the package `coda`, by first defining `outL` to be a list of `mcmc` objects:

```
> library(coda)
#turn output into MCMC list
> res <- mcmc.list(as.mcmc(outL[[1]]),as.mcmc(outL[[2]]),as.mcmc(outL[[3]]))
> summary(window(res, start=1001)) #remove first 1000 iterations as burn-in


[... some output removed ...]

            Mean         SD    Naive SE   Time-series SE
sigma     1.9723    0.13093   0.0011952        0.0087055
lam0      0.1115    0.01535   0.0001401        0.0009003
psi       0.7130    0.10787   0.0009847        0.0077910
N       499.6166   74.74934   0.6823650        5.4232653

2. Quantiles for each variable:

             2.5%        25%        50%        75%      97.5%
sigma     1.74339     1.8811     1.9637     2.0530     2.2618
lam0      0.08443     0.1007     0.1105     0.1211     0.1438
psi       0.52046     0.6350     0.7093     0.7814     0.9627
N       366.00000   446.0000   497.0000   547.0000   674.0000
```

Now that we have parallel chains we can also use the function `gelman.diag` to evaluate if chains have converged:

```
> gelman.diag(window(res, start=1001)) #assess chain convergence

Potential scale reduction factors:

       Point est. Upper C.I.
sigma        1.01       1.04
lam0         1.01       1.02
psi          1.07       1.21
N            1.07       1.21

Multivariate psrf

1.05
```

We can see that estimates are similar to what we observed when running a single chain (see Section 17.6) and that all 3 chains appear to have converged, based on their point estimates of the $\hat{R}$ statistic, but, as already noted before, for a real analysis we might want to run this model for quite a bit longer, to bring down the upper confidence interval limits on $\hat{R}$ for ψ and N. If you have 3 cores then running these 3 parallel chains should not have taken longer than running a single chain. Yet if you look at the effective sample size now using `effectiveSize`, you can see that it has roughly tripled, as we would expect:

```
> effectiveSize(window(res, start=1001))
   sigma       lam0        psi          N
272.6935  411.8384  167.4192  168.3355
```

17.8.2 Using C++

Parallel computing is a great tool to speed up computations, but its usefulness is limited by how many cores you have available. Even with a decent number of cores, large models may still take a long time to run. A major reason for this is that for-loops in **R** are time consuming, whereas they are handled much more time-efficiently in other computer languages such as **C++**. As we saw above, MCMC algorithms consist of for-loops within for-loops, so that it stands to reason that implementing them in a language like **C++** should make those algorithms run much faster. Being avid **R** users, we cannot claim to be fluent in **C++** or to be aware of all the opportunities this language brings for faster computing. It is also beyond the scope of this book to go into the nuts and bolts of how **C++** works or provide a tutorial, and we refer you to the vast amounts of online and print material designed to give the interested user an introduction to **C++**. Just google “introduction C++” and you are sure to come across sites such as `http://www.cplusplus.com` that provide step-by-step instructions to get you started. Here, we only want to point out one approach to linking **R** with **C++**: the packages `inline` (Sklyar et al., 2010) and `RcppArmadillo` (François et al., 2011). These two packages provide a very convenient interface between the two languages, but there are other ways of calling **C++** functions from within **R**, such as the `.Call` command. If you are interested, we suggest you refer to the package manuals and vignettes, as well as the online document “Writing R extensions” (at `http://cran.r-project.org/doc/manuals/R-exts.html`) for a much more thorough treatment of this topic.

In order to use **C++** you need a compiler such as `g++` that (together with other compilers, for example for **C** and **FORTRAN**) comes with **Rtools**, which you can easily download from the web (at `http://cran.r-project.org/bin/windows/Rtools/`). All of these compilers are part of the GNU compiler collection (`http://gcc.gnu.org/`). Make sure the version of **Rtools** matches your version of **R** or you may run into compilation errors later on. To give you a taste of **C++** we will show you how to write a function that calculates the squared distances of individual activity centers to all traps, as is implemented in the `scrbook` package in the function `e2dist` (to be exact, `e2dist` calculates the distance, not the squared distance), and compare performance between **R** and **C++**. We will refer to these functions as “distance functions.” First, let us set up dummy data—a matrix holding the coordinates of the trap array, outer limits of the state-space, and uniformly distributed activity centers for $M = 700$ individuals:

```
> gx <- seq(1,10,1)
> gy <- seq(1,10,1)
> X <- as.matrix(expand.grid(gx, gy))
> M <- 700
> J <- dim(X)[1]
> b <- 3
> xl <- min(gx)-b
> xu <- max(gx)+b
> yl <- min(gy)-b
> yu <- max(gy)+b
> S <- cbind(runif(M, xl, xu), runif(M, yl,yu))
```

Next, we can write a "pedestrian" version of `e2dist` and check how long it takes to calculate the squared distance matrix:

```
> Dfun <- function(M, J, S, X){
+ D2 <- matrix(0, nrow=M, ncol=J)
+ for(i in 1:M){
     + for(j in 1:J){
     + D2[i,j] <- (S[i,1]-X[j,1])^2 + (S[i,2]-X[j,2])^2
     + }}
+ return(D2)
+}

> system.time(
+ (D2R <- Dfun(M, J, S, X))
+)

user  system  elapsed
0.81    0.01     0.82
```

The code to implement the same function in **C++** using the `inline` and `RcppArmadillo` packages is shown in Panel 17.3. These packages allow you to use a range of data formats such as lists and matrices, and they take care of compiling the code in **C++** and loading the resulting function into **R**. This is also referred to compiling **C++** code "on the fly." You will see that the way the code is set up is reasonably similar to **R**. One difference that is worthy to point out is that in **C++** indices for vectors range from 0 to $n-1$, *not* from 1 to n, as in **R**. Note that with `inline` we only need to write the core of the code and define the type of the variables we want to pass to the function, while the `cxxfunction` call takes care of the rest. Once your function is compiled and loaded you should check out the full **C++** code by calling `DfunArma@code`.

Executing this code shows that it is faster than the **R** version of the distance function or `e2dist`; in fact, it is too fast for the time resolution of the `system.time` function to even give us a time estimate:

```
### calculate squared distances using RcppArmadillo
library(inline)
library(RcppArmadillo)

#write core of function code
code<-'
/*define input, assign correct class (matrix, vector etc)*/
arma::mat Sn=Rcpp::as<arma::mat>(S);
arma::mat Xn=Rcpp::as<arma::mat>(X);
int Ntot=Rcpp::as<int>(M);
int ntraps=Rcpp::as<int>(J);
/*create matrix to hold squared distances*/
arma::mat D2(Ntot, ntraps);

/*loop over M and J to calculate distances*/
for (int i=0; i<Ntot; i++){
for(int j=0; j<ntraps; j++){
D2(i,j)= pow(Sn(i,0)-Xn(j,0), 2) + pow(Sn(i,1)-Xn(j,1), 2);
}
}
/*return D2 in R format*/
return Rcpp::wrap(D2);
'

# compile and load
DfunArma<-cxxfunction(signature(M="integer", J="integer", S="numeric",
X="numeric"), plugin="RcppArmadillo", body=code)
```

PANEL 17.3

Code to compute squared distance between individual activity centers and traps in **C++** from within **R** using `inline` and `RcppArmadillo`.

```
> system.time(
+ (out <- DfunArma(M,J,S,X)))

user  system  elapsed
   0       0        0
```

While speed differences of less than 1 s may seem negligible, remember that each command has to be executed at each iteration of the Markov chain. Especially with time-consuming models such as those for open populations (Chapter 16) or multi-session models (Chapter 14) we believe that **C++** holds large potential to make implementation of such models more feasible.

17.9 Summary and outlook

Programs like **JAGS** and **WinBUGS** do all the MCMC-related things that we went through in this chapter (and quite a bit more). Looking through your model, they determine which parameters they can use standard Gibbs sampling for (i.e., for conjugate full conditional distributions). Then, they determine whether to use adaptive rejection sampling, slice sampling, or—in the "worst" case—Metropolis-Hastings sampling for the other full conditionals (how the sampler is chosen differs among softwares). For MH sampling, they will automatically tune the updater so that it works efficiently.

Although these programs are flexible and extremely useful for performing MCMC simulations, it sometimes is more efficient to develop your own MCMC algorithm. Building an MCMC code follows three basic steps: Identify your model including priors and express full conditional distributions for each model parameter. If full conditionals are parametric distributions, use Gibbs sampling to draw candidate parameter values from those distributions; otherwise use Metropolis-Hastings sampling to draw candidate values from a proposal distribution and accept or reject them based on their posterior probability densities.

These custom-made MCMC algorithms give you more modeling flexibility than existing software packages, especially when it comes to handling the state-space: In **WinBUGS** and **JAGS** we define a continuous rectangular state-space using the corner coordinates to constrain the uniform priors on the activity centers **s**. But what if a continuous rectangle is an inadequate description of the state-space? In this chapter we saw that in **R** it only takes a few lines of code to use any arbitrary polygon shapefile as the state-space, which is especially useful when you are dealing with coastlines or large bodies of water that need removing from the state-space. Another example is the SCR **R** package `SPACECAP` (Gopalaswamy et al., 2012a) that was developed because implementation of an SCR model with a discrete state-space was inefficient in **WinBUGS**.

Another situation in which using a **BUGS** engine becomes increasingly complicated or inefficient is when using point processes other than the homogeneous binomial point process ("uniformity of density") which underlies the basic SCR model (see Section 5.10~~ in Chapter 5~~). In Chapter 11 you already saw an example of an inhomogeneous point process model and we briefly introduce a different point process, implemented using a custom-made MCMC algorithm, in Chapter 20. Finally, Chapter 19 deals with partially marked populations using custom-made MCMC algorithms to handle the (partially) latent individual encounter histories. While some of these models can be written in the **BUGS** language, they are painstakingly slow; others (for example, the classes of models considered in Chapter 12) cannot be implemented in **WinBUGS/JAGS** at all and we have to either use likelihood-based inference or develop our own MCMC algorithms. In conclusion, while you can certainly get by using **BUGS/JAGS** for standard SCR models, knowing how to write your own MCMC sampler gives you more flexibility to tailor these models to your specific needs.

This page is intentionally left blank

Non Print Items

Abstract: The aim of this chapter is to provide you with working knowledge of how to construct Markov chain Monte Carlo (MCMC) algorithms. This comes in handy when **BUGS** engines fail or are inefficient, or when additional flexibility in model specification or algorithm tuning is needed. We describe technical background and concepts related to MCMC sampling, summarization of posterior output, and specific methods for constructing MCMC algorithms such as Gibbs sampling, Metropolis-Hastings, and other methods. We provide specific examples of developing MCMC samplers in **R** using GLMs and GLMMs, and then extend these ideas to capture-recapture (model M_h) and SCR models. We provide an SCR example where the state-space is an irregular polygon defined by an ESRI shapefile, which cannot be used in any of the **BUGS** engines but can easily be added to an **R** based MCMC sampler. Finally, we discuss ideas for increasing computational speed by using multiple cores simultaneously using the **R** package `snowfall`, or integrating **C++** code with **R** using the packages `RcppArmadillo` and `inline`.

Keywords: C++, Cluster computing, Convergence, Gibbs sampling, Markov chain Monte Carlo, Metropolis-Hastings, Metropolis-within-Gibbs, Parallel computing, Posterior simulation, Rejection sampling, Shape file, Slice sampling