

# Chapter 1

## MCMC Details

### 1.1 Introduction

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 GLMs and GLMMs and moving on to simple SCR models. We will also demonstrate some tricks and simple extensions to the 'spatial null model'. Finally, we will illustrate some alternative ready-to-use software packages for MCMC sampling. 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) and Robert and Casella (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.

#### 1.1.1 Why build your own MCMC algorithm?

The standard program we have used so far to run MCMC analyses is WinBUGS (Gilks et al., 1994). The wonderful thing about WinBUGS is that it will automatically use the most appropriate and efficient form 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 WinBUGS. While OpenBUGS 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 a little bit),

but you cannot use a shape file in a BUGS model. Sometimes implementing an MCMC algorithm in R may be faster 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. 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. Not to mention that, if you are an R junkie, it can actually be fun. 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.

## 1.2 MCMC and posterior distributions

As mentioned in Chapter 2, 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  $\theta$ , given a set of observed data and its prior probability distribution (the probability distribution we assign to a parameter before we observe data). The great benefit of computing the posterior distribution of  $\theta$  is that it can be used to make probability statements about  $\theta$ , such as the probability that  $\theta$  is equal to some value, or the probability that  $\theta$  falls within some range of values. As an example, suppose we conducted a Bayesian analysis to estimate detection probability of some species at a study site ( $p$ ), and we obtained a posterior distribution of  $\text{beta}(20,10)$  for the parameter  $p$ . The following R commands demonstrate how we make inferences based upon summaries of the posterior distribution. Fig 1 shows the posterior along with the summary statistics.

```
> (post.median <- qbeta(0.5, 20, 10))
[1] 0.6704151
> (post.95ci <- qbeta(c(0.025, 0.975), 20, 10))
[1] 0.4916766 0.8206164
```

Thus, we can state that there is a 95% probability that  $\theta$  lies between 0.49 and 0.82.

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 directly compute the posterior. Recall Bayes theorem:

$$p(\theta|y) = p(y|\theta) * p(\theta)/p(y), \quad (1.1)$$

where  $\theta$  is the parameter of interest,  $y$  is the observed data,  $p(\theta|y)$  is the posterior,  $p(y|\theta)$  the likelihood of the data conditional on  $\theta$ ,  $p(\theta)$  the prior probability

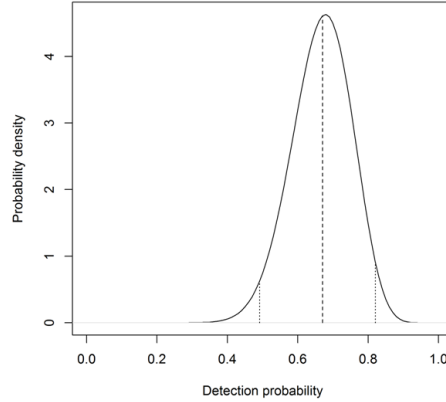


Figure 1.1: Probability density plot of a hypothetical posterior distribution of  $\text{beta}(20,10)$ ; dashed lines indicate mean and upper and lower 95% interval

69 of  $\theta$ , and, finally,  $p(y)$  is the marginal probability of the data, which can also be  
70 written as

$$p(y) = \int p(y|\theta) * p(\theta) d\theta$$

71 This marginal probability is a normalizing constant that ensures that the  
72 posterior integrates to 1. You read in Chapter 2 that this integral is often hard  
73 or impossible to evaluate, unless you are dealing with a really simple model. For  
74 example, consider that you have a Normal model, with a set of  $n$  observations,  
75  $y$  that come from a Normal distribution:

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

76 where  $\sigma$  is known and our objective is to obtain an estimate of  $\mu$  using  
77 Bayesian statistics. To fully specify the model in a Bayesian framework, we  
78 first have to define a prior distribution for  $\mu$ . Recall from Chapter 2 that for  
79 certain data models, certain priors lead to conjugacy i.e. if you choose the  
80 right prior for your parameter, your posterior distribution will be of a known  
81 parametric form. The conjugate prior for the mean of a normal model is also a  
82 Normal distribution:

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

83 If  $\mu_0$  and  $\sigma_0^2$  are fixed, the posterior for  $\mu$  has the following form (for the  
84 algebraic proof, see XXX):

$$\mu|y \sim \text{Normal}(\mu_n, \sigma_n^2) \quad (1.2)$$

85 where

$$\text{mun} = (\text{sig}^2 / \text{sig}^2 + n * \text{sig}0^2) * \text{mu}0 + (n * \text{sig}0^2 / \text{sig}^2 + n * \text{sig}0^2) * y - \text{bar}$$

86 And

$$\text{sign}^2 = \text{sig}^2 * \text{sig}0^2 / (\text{sig}^2 + n * \text{sig}0^2)$$

87 We can directly obtain estimates of interest from this Normal posterior distri-  
 88 bution, such as the mean  $\mu$ -hat and its variance; we do not need to apply  
 89 MCMC, since we can recognize the posterior as a parametric distribution, in-  
 90 cluding the normalizing constant  $p(y)$ . But generally we will be interested in  
 91 more complex models with several, say  $n$ , parameters. In this case, computing  
 92  $p(y)$  from Eq. 1.1 requires  $n$ -dimensional integration, which is can be difficult  
 93 or impossible. Thus, the posterior distribution is generally only known up to a  
 94 constant of proportionality:

$$p(\theta|y) \propto p(y|\theta) * p(\theta)$$

95 The power of MCMC is that it allows us to approximate the posterior using sim-  
 96 ulation without evaluating the high dimensional integrals and to directly sample  
 97 from the posterior, even when the posterior distribution is unknown! The price  
 98 is that MCMC is computationally expensive. Although MCMC first appeared  
 99 in the scientific literature in 1949 (Metropolis and Ulam, 1949), widespread use  
 100 did not occur until the 1980s when computational power and speed increased  
 101 (Gelfand and Smith, 1990). It is safe to say that the advent of practical MCMC  
 102 methods is the primary reason why Bayesian inference has become so popular  
 103 during the past three decades. In a nutshell, MCMC lets us generate sequential  
 104 draws of  $\theta$  (the parameter(s) of interest) from distributions approximating the  
 105 unknown posterior over  $T$  iterations. The distribution of the draw at  $t$  depends  
 106 on the value drawn at  $t-1$ ; hence, the draws from a Markov chain.<sup>1</sup> As  $T$  goes  
 107 to infinity, the Markov chain converges to the desired distribution in our case  
 108 the posterior distribution for  $\theta|y$ . Thus, once the Markov chain has reached  
 109 its stationary distribution, the generated samples can be used to characterize  
 110 the posterior distribution,  $p(\theta|y)$ , and point estimates of  $\theta$ , its standard error  
 111 and confidence bounds, can be obtained directly from this approximation of the  
 112 posterior. In practice, although we know that a Markov chain will eventually  
 113 converge, we can only generate a limited number of samples a process that  
 114 depending on the model can be quite time consuming. Assessing whether our  
 115 Markov chain has indeed converged is an important part of MCMC sampling  
 116 and we will speak about some common diagnostics in Section XX.

### 117 1.3 Types of MCMC sampling

118 There are several MCMC algorithms, the most popular being Gibbs sampling  
 119 and Metropolis-Hastings sampling. We will be dealing with these two classes in

<sup>1</sup>In case you are not familiar with Markov chains, for  $t$  random samples  $\theta(1), \dots, \theta(t)$  from a Markov chain the distribution of  $\theta(t)$  depends only on the most recent value,  $\theta(t-1)$ .

more detail and use them to construct the MCMC algorithms for SCR models. Also, we will briefly review alternative techniques that are applicable in some situations.

### 1.3.1 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<sup>2</sup>. The roots of Gibbs sampling can be traced back to work of Metropolis et al. (1953), and it is actually closely related to Metropolis sampling (see Chapter 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. In Chapter 2 you already heard about the basic principles of Gibbs sampling<sup>3</sup>. But as a refresher, let's go back to our simple 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  $\mu$ , the posterior distribution of  $\mu|y$  is also Normal. Conversely, with a fixed (known)  $\mu$ , but unknown variance, the conjugate prior for  $\sigma^2$  is an Inverse Gamma distribution with shape and scale parameters  $a$  and  $b$ :

$$\sigma^2 \sim \text{InvGamma}(a, b),$$

With fixed  $a$  and  $b$ , the posterior  $p(\sigma^2|\mu, y)$  is also an Inverse Gamma distribution, namely:

$$\sigma^2|\mu, y \sim \text{InvGamma}(an, bn), \quad (1.3)$$

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

$$p(\mu, \sigma^2|y) = \frac{p(y|\mu) * p(\mu) * p(\sigma^2)}{\int p(y|\mu) * p(\mu) * p(\sigma^2) d\mu d\sigma^2}$$

Or

$$p(\mu, \sigma^2|y) \propto p(y|\mu) * p(\mu) * p(\sigma^2)$$

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

$$p(\mu|\sigma^2, y) \propto p(y|\mu) * p(\mu)$$

The full conditional of  $\mu$  again takes the form of the Normal distribution shown in Eq. 1.2; similarly,  $p(\sigma^2|\mu, y)$  takes the form of the Inverse Gamma

<sup>2</sup>a distribution from physics we are not going to worry about, since it has no immediate connection with Gibbs sampling other than giving it its name

<sup>3</sup>maybe we should think out chapter 2 and concentrate that material here?

150 distribution shown in Eq. 1.3 both distribution we can easily sample from.  
 151 And this is precisely what we do when using Gibbs sampling we break down  
 152 high-dimensional problems into convenient one-dimensional problems by con-  
 153 structing the full conditional distributions for each model parameter separately;  
 154 and we sample from these full conditionals, which, if we choose conjugate priors,  
 155 are known parametric distributions. Let's put the concept of Gibbs sampling  
 156 into the MCMC framework of generating successive samples, using our simple  
 157 Normal model with unknown  $\mu$  and  $\sigma$  and conjugate priors as an example.  
 158 These are the steps you need to build a Gibbs sampler:

159 **Step 0:** Begin with some initial values for  $\theta$ ,  $\theta(0)$ . In our example, we have to  
 160 specify initial values for  $\mu$  and  $\sigma$ , for example by drawing a random number  
 161 from some uniform distribution, or by setting them close to what we think they  
 162 might be. (Note: This step is required in any MCMC sampling chains have to  
 163 start from somewhere. We will get back to these technical details a little later.)

164 **Step 1:** Draw  $\theta_1(1)$  from the conditional distribution  $p(\theta_1(1) | \theta_2(0), \dots, \theta_d(0))$   
 165 Here,  $\theta_1$  is  $\mu$ , which we draw from the Normal distribution in Eq. 1.2 using  
 166  $\sigma(0)$  as value for  $\sigma$ .

167 **Step 2:** Draw  $\theta_2(1)$  from the conditional distribution  $p(\theta_2(1) | \theta_1(1), \theta_3(0), \dots, \theta_d(0))$   
 168 Here,  $\theta_2$  is  $\sigma$ , which we draw from the Inverse Gamma distribution of  
 169 Eq. 1.3, using  $\mu(1)$  as value for  $\mu$ ...

170 **Step d:** Draw  $\theta_d(1)$  from the conditional distribution  $p(\theta_d(1) | \theta_1(1), \dots, \theta_{d-1}(1))$   
 171

172 In our example we have no additional parameters, so we only need step 0  
 173 through to 2. Repeat Steps 1 to d for  $K$  = a large number of samples. In terms  
 174 of R coding, this means we have to write Gibbs updaters for  $\mu$  and  $\sigma$  and  
 175 embed them into a loop over  $K$  iterations. The final code in the form of an R  
 176 function is shown in Panel 1.

177 **Panel 1:** R-code for a Gibbs sampler for a Normal model with unknown  $\mu$  and  $\sigma$  and con-  
 178 `Normal.Gibbs<-function(y=y,mu0=mu0, sig0=sig0, a=a,b=b,niter=niter) {`

```
179
180   ybar<-mean(y)
181   n<-length(y)
182   mu<-runif(1) #mean initial value
183   sig<-runif(1) #sd initial value
184   an<-n/2 + a
185
186   out<-matrix(nrow=niter, ncol=2)
187   colnames(out)<-c('mu', 'sig')
188
189   for (i in 1:niter) {
190
191     #update mu
```

```

192 mun<- (sig/(sig+n*sig0))*mu0 + (n*sig0/(sig+n* sig0))*ybar
193 sign <- (sig*sig0)/ (sig+n*sig0)
194 mu<-rnorm(1,mun, sqrt(sign))
195
196 #update sig
197 bn<- 0.5 * (sum((y-mu)^2)) +b
198 sig<-1/rgamma(1,shape=an, rate=bn)
199 out[i,]<-c(mu,sqrt(sig))
200
201 }
202 return(out)
203 }

```

204 This is it! You can use the code NormalGibbs.R in the online appendix to  
 205 simulate some data,  $y \sim \text{Normal}(5, 0.5)$  and run your first Gibbs sampler. Your  
 206 output will be a table with two columns, one per parameter, and K rows, one  
 207 per iteration. For this 2-parameter example you can visualize the joint posterior  
 208 by plotting samples of mu against samples of sig (Fig 2):

```

209 plot(out[,1], out[,2])

```

210 The marginal distribution of each parameter is approximated by just examining  
 211 the samples of this particular parameter you can visualize it by plotting a  
 212 histogram of the samples (Figure 3 a, b):

```

213 par(mfrow=c(1,2))
214 hist(out[,1]); hist (out[,2])

```

215 Finally, recall an important characteristic of Markov chains, namely, that the  
 216 chain has to have converged (reached its stationary distribution) for samples to  
 217 come from the posterior distribution. In practice, that means you have to throw  
 218 out some of the initial samples called the burn-in. We will talk about this in  
 219 more when we talk about convergence diagnostics. For now, you can use the  
 220 `plot(out[,1])` or `plot(out[,2])` command to make a time series plot of the  
 221 samples of each parameter and visually assess how many of the initial samples  
 222 you should discard. Figure 3 c and d shows plots for the estimates of mu and  
 223 sigma from our simulated data set; you see that in this simple example the  
 224 Markov chain apparently reaches its stationary distribution very quickly the  
 225 chains look 'grassy' seemingly from the start. It is hard to discern a burn-in  
 226 phase visually (but we will see examples further on where the burn-in is clearer)  
 227 and you may just discard the first 500 draws to be sure you only use samples  
 228 from the posterior distribution. The mean of the remaining samples are your  
 229 estimates of mu and sig:

```

230 > summary(mod[501:10000,])
231      mu      sig
232 Min.   : 4.936   Min.   : 0.4569
233 1st Qu.: 4.984   1st Qu.: 0.4889

```

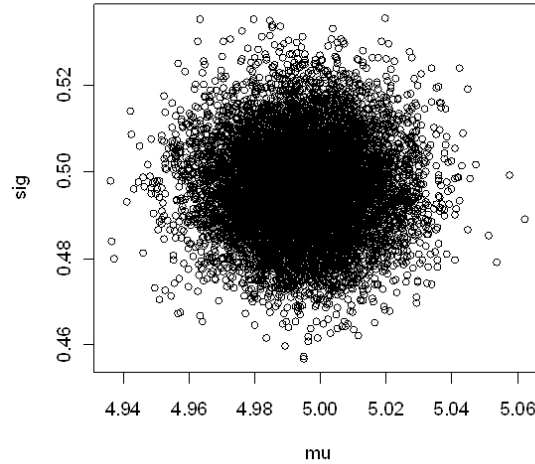


Figure 1.2: Joint posterior distribution of mu and sig from a Normal Model

```

234 Median : 4.994    Median : 0.4961
235 Mean   : 4.994    Mean   : 0.4964
236 3rd Qu.: 5.005    3rd Qu.: 0.5037
237 Max.   : 5.062    Max.   : 0.5356

```

### 1.3.2 Metropolis-Hastings sampling

Although it is applicable to a wide range of problems, the limitations of Gibbs sampling are immediately 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 2. In a nutshell, because we do not recognize the posterior  $p(\theta|y)$  as a parametric distribution, the MH algorithm generates samples from a known proposal distribution, say  $h(\theta)$ , that depends on  $\theta$  at  $t-1$ . The  $t$ -th sample is accepted or rejected based on its joint posterior probability density compared to the density of the sample at  $t-1$ . The original Metropolis algorithm requires  $h(\theta)$  to be symmetric so that  $h(\theta_t|\theta_{t-1}) = h(\theta_{t-1}|\theta_t)$ ; but a later development of the algorithm by Hastings (1970) lifted this condition. Using a symmetric proposal distribution makes life a little easier and we are going to limit our coverage of the Metropolis-Hastings sampler to this specific case. Specifically, we are going to use a Normal proposal distribution, which



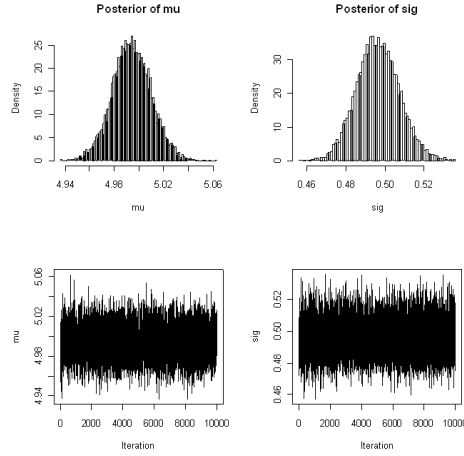


Figure 1.3: Plots of the posterior distributions of  $\mu$ (a) and  $\text{sig}$  (b) from a Normal model and time series plots of  $\mu$  (c) and  $\text{sig}$  (d).

is also referred to as 'random walk Metropolis-Hastings sampling'. It is worth knowing that there are alternative formulations of the algorithm. For example, in the independent M-H,  $\theta_t$  does not depend on  $\theta_{t-1}$ , while the Langevin algorithm (Roberts and Rosenthal, 1998) aims at avoiding the random walk by favoring moves towards regions of higher posterior probability density. The interested reader should look up these algorithms in Robert and Casella (2004) or Robert and Casella (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

$$y \sim \text{Bernoulli}(\exp(\theta)/(1 + \exp(\theta)))$$

and

$$\theta \sim \text{Normal}(\mu_0, \text{sig})$$

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  $\theta$  at  $t$  from  $h(\theta_t - \theta_{t-1})$ . We often use a Normal proposal distribution, so we draw  $\theta_1$  from  $\text{Normal}(\theta_0, \text{sig}^2)$ , where  $\text{sig}^2$  is the variance of the Normal proposal distribution, a tuning parameter that we have to set.

Step2: Calculate the ratio of posterior densities for the proposed and the original value for  $\theta$ :

$$r = p(\theta(t)|y)/p(\theta(t-1)|y)$$

274 In our example,

$$r = \text{Bernoulli}(y|\theta_t) * \text{Normal}(\theta_t | \mu_0, \text{sig}_0) / \text{Bernoulli}(y|\theta_{t-1}) * \text{Normal}(\theta_{t-1} | \mu_0, \text{sig}_0)$$

275 Step 3: Set

```
276 \begin{eqnarray*}
277 \theta_t &= & \theta_t \text{ with probability } \min(r,1) // \\
278 &= & \theta_{t-1} \text{ otherwise } \\
279 \end{eqnarray*}
```

280 We can do that by drawing a random number  $u$  from a Uniform (0,1) and  
 281 accept  $\theta_t$  if  $u < r$  Repeat for  $t =$  a large number of samples. The R code for  
 282 this MH sampler is provided in Panel 2.

```
283 Panel 2: R code to run a Metropolis sampler on a simple Logit-Normal model.
284 Logreg.MH<-function(y=y, mu0=mu0, sig0=sig0, niter=niter) {
285
286   out<-c()
287
288   theta<-runif(1, -3,3) #initial value
289
290   for (iter in 1:niter){
291     theta.cand<-rnorm(1, theta, 0.2)
292
293     loglike<-sum(dbinom(y, 1, exp(theta)/(1+exp(theta)), log=TRUE))
294     logprior <- dnorm(theta,mu0 ,sig0, log=TRUE)
295     loglike.cand<-sum(dbinom(y, 1, exp(theta.cand)/(1+exp(theta.cand)), log=TRUE))
296     logprior.cand <- dnorm(theta.cand, mu0, sig0, log=TRUE)
297
298     if (runif(1)<exp((loglike.cand+logprior.cand)-(loglike+logprior))){
299       theta<-theta.cand
300     }
301     out[iter]<-theta
302   }
303
304   return(out)
305 }
```

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

312 Comparing MH sampling to Gibbs sampling, where all draws from the con-  
 313 ditional distribution are used, in the MH algorithm we discard a portion of the  
 314 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 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, i.e. your Markov chain will converge faster. The variance should be chosen so that (a) each step of drawing a new proposal value for  $\theta$  can cover a reasonable distance in the parameter space, as otherwise, the random walk moves too slowly; and (b) proposal values are not rejected too often, as otherwise the random walk will 'get stuck' at specific values for too long. As a rule of thumb, your 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 distribution). In practice, tuning will require some 'trial-and-error' and some common sense. 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 (2009) discuss this in more detail. It is important the samples drawn during the adaptive phase are discarded. You can easily check acceptance rates for the parameters you monitor (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.

### 1.3.3 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  $\theta$  becomes increasingly complex or inefficient as the number of parameters in a model increases. It is probably going to sound like MCMC sampling is too good to be true but in these cases you can simply combine MH sampling and Gibbs sampling. You can use 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.

## 1.4 GLMMs Poisson regression with a random effect

Let's assume a model that gets us closer to the problem we ultimately want to deal with a GLMM. Here, we assume we have Poisson counts,  $y$ , from  $i$  plots in  $j$  different study sites, and we believe that the counts are influenced by some plot-specific covariate,  $x$ , but that there is also a random site effect. So our model is:

$$y_{ij} \sim \text{Poisson}(lami_{ij})$$

$$lami_{ij} = \exp(a_j + b * x_i)$$

Let's use Normal priors on  $a$  and  $b$ ,

$$a_j \sim \text{Normal}(\mu_a, \sigma_a)$$

and

$$b \sim \text{Normal}(\mu_b, \sigma_b)$$

.

<sup>4</sup> Since we want to estimate the random effect in this model, we do not specify  $\mu_a$  and  $\sigma_a$ , but instead, estimate them as well, so we have to specify hyperpriors for these parameters:

```
mua \sim \text{Normal}(\mu_0, \sigma_0)
```

```
sigma \sim \text{InvGamma}(a_0, b_0)
```

With the model fully specified, we can compile the full conditionals, breaking the multi-dimensional parameter space into one-dimensional components:

```
\begin{eqnarray*}
p(a_1|a_2,a_3,a_j,b,y) & \propto & p(y_{i1}|a_1,b) * p(a_1|\mu_a, \sigma_a) \backslash \backslash \\
& \propto & \text{Poisson}(y_{i1} | \exp(a_1 + b*x[j=1])) * \text{Normal}(a_1|\mu_a, \sigma_a) \\
\end{eqnarray*}
\begin{eqnarray*}
p(a_2|a_1,a_3,a_j,b,y) & \propto & p(y_{i2}|a_2,b) * p(a_2|\mu_a, \sigma_a) \backslash \backslash \\
& \propto & \text{Poisson}(y_{i2} | \exp(a_2 + b*x[j=1])) * \text{Normal}(a_2|\mu_a, \sigma_a) \\
\end{eqnarray*}
\text{and so on for all elements of } a.
\begin{eqnarray*}
p(b|a,y) & \propto & p(y|a,b) * p(b) \backslash \backslash \\
& \propto & \text{Poisson}(y | \exp(a + b*x)) * \text{Normal}(b|\mu_b, \sigma_b) \\
\end{eqnarray*}
```

Finally, we need to update the hyperparameters for  $a$ :

$$p(\mu_a|a) \propto p(a|\mu_a, \sigma_a) * p(\mu_a)$$

---

<sup>4</sup>Why is  $b$  a hyperparameter?

$$p(siga|a) \propto p(a|mua, siga) * p(siga)$$

Since we assumed  $a$  to come from a Normal distribution, the choice of priors for  $mua$  Normal and  $siga$  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  $a$  and  $b$  we use Normal proposal distributions with standard deviations  $sigha$  and  $sighb$ .

First, we set the initial values  $a(0)$  and  $b(0)$ . Then, starting with  $a1$ , we draw  $a1(1)$  from Normal( $a1(0)$ ,  $sigha$ ), calculate the conditional posterior density of  $a1(0)$  and  $a1(1)$  and compare their ratios,

$$r = \text{Poisson}(y(j=1)|\exp(a1(1)+b*x)) * \text{Normal}(a1(1)|mua, siga) / \text{Poisson}(y(j=1)|\exp(a1(0)+b*x)) * \text{Normal}(a1(0)|mua, siga)$$

and accept  $a1(1)$  with probability  $\min(r, 1)$ . We repeat this for all  $a$ 's.

For  $b$ , we draw  $b(1)$  from Normal( $b(0)$ ,  $sighb$ ), compare the posterior densities of  $b(0)$  and  $b(1)$ ,

$$r = \text{Poisson}(y|\exp(a+b(1)*x)) * \text{Normal}(b(1)|mub, sigb) / \text{Poisson}(y|\exp(a+b(0)*x)) * \text{Normal}(b(0)|mub, sigb),$$

and accept  $b(1)$  with probability  $\min(r, 1)$ .

For  $mua$  and  $siga$ , we sample directly from the full conditional distributions (Eq XX and Eq XX):

$$mua(1) \sim \text{Normal}(mun, sign)$$

where  $mun = (siga(0)/siga(0) + na * sig0) * mu0 + (na * sig0/siga(0) + na * sig0) * abar(1)$  and  $sign = siga(0) * sig0 / (siga(0) + n * sig0)$ . Here,  $abar$  is the current mean of the vector  $a$ , which we updated before, and  $na$  is the length of  $a$ . For  $siga$  we use  $siga(1) \sim \text{InvGamma}(an, bn)$ , where  $an = na/2 + a0$ , and  $bn = 1/2 \sum (a(1) - mua(1))^2 + b0$ .

We repeat these steps over  $K$  iterations of the MCMC algorithm. In this example we may not want to save each value for  $a$ , 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  $a$  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 full code for the MCMC algorithm of our Poisson GLMM in Panel 3 shows one way how to monitor acceptance of individual  $a$ 's.

Panel 3: R code for the Metropolis-within-Gibbs sampler for a Poisson regression with random intercepts

```
Pois.reg<-function(y=y,site=site,mu0=mu0,sig0=sig0,a0=a0,b0=b0, mub=mub, sigb=sigb, niter=niter){
  lev<-length(unique(site))      #number of sites
```

```

424 a<-runif(lev,-5,5) #initial values a
425 b<-runif(1,0,5) #initial value b
426 mua<-mean(a)
427 siga<-sd(a)
428
429 out<-matrix(nrow=niter, ncol=3)
430 colnames(out)<-c('mua','siga','b')
431
432 for (iter in 1:niter) {
433
434   #update a
435   aUps<-0 #initiate counter for acceptance rate of a
436   for (j in 1:lev) { #loop over sites
437     a.cand<-rnorm(1, a[j], 0.1) #update intercepts a one at a time
438     loglike<- sum(dpois (y[site==j], exp(a[j] + b*x[site==j]), log=TRUE))
439     logprior<- dnorm(a[j], mua,siga, log=TRUE)
440     loglike.cand<- sum(dpois (y[site==j], exp(a.cand + b *x[site==j]), log=TRUE))
441     logprior.cand<- dnorm(a.cand, mua,siga, log=TRUE)
442     if (runif(1)< exp((loglike.cand+logprior.cand) (loglike+logprior))) {
443       a[j]<-a.cand
444       aUps<-aUps+1
445     }
446   }
447
448   if(iter %% 100 == 0) { #this lets you check the acceptance rate of a at every 100th i
449     cat("      Acceptance rates\n")
450     cat("      a =", aUps/lev, "\n")
451   }
452
453   #update b
454   b.cand<-rnorm(1, b, 0.1)
455   avec<-rep(a, times=c(rep(10,10)))
456   loglike<- sum(dpois (y, exp(avec + b*x), log=TRUE))
457   logprior<- dnorm(b, mub,sigb, log=TRUE)
458   loglike.cand<- sum(dpois (y, exp(avec + b.cand *x), log=TRUE))
459   logprior.cand<- dunif(b.cand, mub,sigb, log=TRUE)
460   if (runif(1)< exp((loglike.cand+logprior.cand) (loglike+logprior) )) {
461     b<-b.cand
462   }
463
464   #update mua using Gibbs sampling
465   abar<-mean(a)
466   mun<- (siga/(siga+lev*sig0))*mu0 + (lev*sig0/(siga+lev* sig0))*abar
467   sign <- (siga*sig0)/ (siga+lev*sig0)
468   mua<-rnorm(1,mun, sqrt(sign))
469

```

```

470 #update siga using Gibbs sampling
471 a0n<-lev/2 + a0
472 b0n<- 0.5 * (sum((a-mua)^2)) +b0
473 siga<-1/rgamma(1,shape=a0n, rate=b0n)
474
475 out[iter,]<-c(mua, sqrt(siga), b)
476
477 }
478
479 return(out)
480
481 }
482

```

#### 1.4.1 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 is more out there to approximate posterior distributions than 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  $p(\theta|y)$  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  $p(\theta|y) < M * g(\theta)$ . We then sample a candidate value for  $\theta$  from  $g(\theta)$ , calculate  $r = p(\theta|y)/M * g(\theta)$  and keep the sample with the probability  $r$ .  $M$  is a constant that has to be picked so that  $r \in [0,1]$ , for example by evaluating both  $p(\theta|y)$  and  $g(\theta)$  at  $n$  points and looking at their ratios. Rejection sampling only works well if  $g(\theta)$  is similar to  $p(\theta|y)$ , 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 9. Another alternative is slice sampling (Neal, 2003). In slice sampling, we sample uniformly from the area under the plot of  $p(\theta|y)$ . Considering a single univariate theta. Let's define an auxiliary variable,  $U \sim Uniform(0, p(\theta|y))$ . Then,  $\theta$  can be sampled from the vertical slice of  $p(\theta|y)$  at  $U$  (Figure 4):

```

508 theta|U ~simUnif(B),

```

where  $B = \theta : U < p(\theta|y)$

5

---

<sup>5</sup>there are supposed to be equations in the caption of figure 4 but it kept causing errors

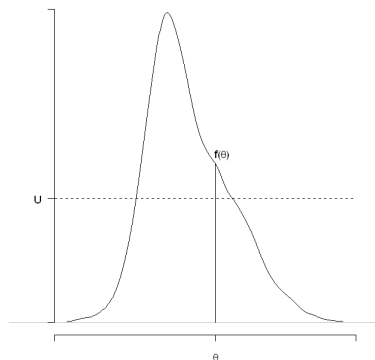


Figure 1.4: Slice sampling. For...

511 Slice sampling can be applied in many situations; however, implementing an  
 512 efficient slice sampling procedure can be complicated. We refer the interested  
 513 reader to chapter 7 of ? for a simple example. Both rejection sampling and slice  
 514 sampling can be applied on one-dimensional conditional distributions within a  
 515 Gibbs sampling setup.

## 516 1.5 MCMC for closed capture-recapture Model 517 Mh

518 6

## 519 1.6 MCMC algorithm for the basic spatial capture- 520 recapture model

521 By now you have seen how to build MCMC algorithms for some basic generalized  
 522 linear models. Now, we'll walk you through the steps of building your own  
 523 MCMC sampler for the basic SCR model (i.e. without any individual, site or  
 524 time specific covariates) with both a Poisson and a binomial encounter process.  
 525 As usual, we will have to go through two general steps before we write the  
 526 MCMC algorithm:

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

529 It is worthwhile to go through all of step 1 for an SCR model, but you have  
 530 probably seen enough of step 2 in our previous examples to get the essence of  
 531 how to express a full conditional distribution. Therefore, we will exemplify step  
 532 2 for some parameters and tie these examples directly to the respective R code.

---

<sup>6</sup>Andy could move material from chapter 3 to here.



533 **Step 1 Identify your model**

534 Recall the components of the basic SCR model with a Poisson encounter  
 535 process from Chapter 3: We assume that individuals  $i$ , or rather, their activity  
 536 centers  $s_i$ , are uniformly distributed across our state space  $S$ ,

$$s_i \sim U(S)$$

537 and that the number of times individual  $i$  encounters trap  $j$ ,  $y_{ij}$ , is a random  
 538 Poisson variable with mean  $\lambda_{mij}$ ,

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

539 The tie between individual location, movement and trap encounter rates is made  
 540 by the assumption that  $\lambda_{mij}$ , is a decreasing function of the distance between  
 541  $s_i$  and  $j$ ,  $D_{ij}$ , of the half-normal form

$$\lambda_{mij} = \lambda_{m0} * \exp(-D_{ij}^2/2 * \sigma^2),$$

542 where  $\lambda_{m0}$  is the baseline trap encounter rate at  $D_{ij} = 0$  and  $\sigma$  controls the  
 543 shape of the half-normal function.

544 In order to estimate the number of  $s_i$  in  $S$ ,  $N$ , we use data augmentation  
 545 (sect. 3.XYZ) and create  $M-n$  all-0 encounter histories, where  $n$  is the number  
 546 of individuals we observed and  $M$  is an arbitrary number that is larger than  $N$ .  
 547 We estimate  $N$  by summing over the auxiliary data augmentation variables,  $z_i$ ,  
 548 which is 1 if the individual is part of the population and 0 if not, and assume  
 549 that  $z_i$  is a random Bernoulli variable,

```
550 \[
551 zi \sim \text{bernoulli}(\psi) .
552 \]
```

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

$$\lambda_{mij} = \lambda_{m0} * \exp(-D_{ij}^2/2 * \sigma^2) * z_i.$$

554 The model has the following structural parameters, for which we need to spec-  
 555 ify priors  $\psi$  the Uniform (0,1) is required as part of the data augmentation  
 556 procedure and in general is a natural choice of an uninformative prior for a  
 557 probability; note that this is equivalent to a Beta(1,1) prior, which will come in  
 558 handy later.  $s_i$  since  $s_i$  is a pair of coordinates it is two-dimensional and we use  
 559 a uniform prior limited by the extent of our state-space over both dimensions.  
 560  $\sigma$  we can conceive several priors for sigma but let's assume an improper prior  
 561 one that is Uniform over  $(-\infty, \infty)$ . We will see why this is convenient when we  
 562 construct the full conditionals for sigma.  $\lambda_0$  analogous, we will use a Uniform  
 563  $(-\infty, \infty)$  improper prior for sigma. The parameter that is the objective of our  
 564 modeling,  $N$ , is a derived parameter that we can simply obtain by summing all  
 565  $z_i$ 's:

$$N = \text{sum}(z)$$

566 **Step 2 - Construct the full conditionals** Having completed step 1,  
 567 let's look at the full conditional distributions for some of these parameters.  
 568 We find that with improper priors, full conditionals are proportional only to  
 569 the likelihood of the observations; for example, take the movement parameter  
 570 sigma:

$$Sig|s, lam0, z, ypropto[y|s, lam0, z, sig] * [sig]$$

571 Since the improper prior implies that  $[sig]$  propto 1, we can reduce this further  
 572 to

$$Sig|s, lam0, z, ypropto[y|s, lam0, z, sig]$$

573 The R code to update sigma is shown in Panel 4.<sup>7</sup>

574 Panel 4: R code to update sigma within an MCMC algorithm for an SCR model when using a  
 575 `sig.cand <- rnorm(1, sigma, 0.1)` #draw candidate value  
 576 `if(sig.cand>0){` #automatically reject sig.cand that are <0  
 577 `lam.cand <- lam0*exp(-(D*D)/(2*sig.cand*sig.cand))`  
 578 `ll<- sum(dpois(y, lam*z, log=TRUE))`  
 579 `llcand <- sum(dpois(y, lam.cand*z, log=TRUE))`  
 580 `if(runif(1) < exp( llcand - ll )){`  
 581 `ll<-llcand`  
 582 `lam<-lam.cand`  
 583 `sigma<-sig.cand`  
 584 `}`  
 585 `}`  
 586

587 These steps are analogous for lam0 and si and we will use MH steps for all of  
 588 these parameters. Similar to the random intercepts in our Poisson GLMM, we  
 589 update each si individually. Note that to be fully correct, the full conditional  
 590 for si contains both the likelihood and prior component, since we did not specify  
 591 an improper, but a Uniform prior on si. However, with a Uniform distribution  
 592 the probability density of any value is  $1/(\text{upper limit} - \text{lower limit}) = \text{constant}$ .  
 593 Thus, the prior components are identical for both the current and the candidate  
 594 value and can be ignored (formally, when you calculate the ratio of posterior  
 595 densities, r, the identical prior component appears both in the numerator and  
 596 denominator, so that they cancel each other out).

597 We still have to update zi. The full conditional for zi is

$$zi|y, sigma, lam0, spropto[y|z, sigma, lam0, s] * [zi]$$

598 and since  $zi \sim \text{Bernoulli}(\psi_i)$ , the term has to be taken into account when  
 599 updating zi. The R code for updating zi is shown in Panel 5.

<sup>7</sup> Somewhere in chapter 2 i added a comment about rejecting parameters outside of the parameter space as being an ok thing to do. Richard said he read something in Robert and Casellas book on that. Hopefully he can remember where and we can cite it back in Ch 2 and again here. It could be mentioned in a sentence or two up in the MCMC section.

```

600 Panel 5: R code to update z
601     zUps <- 0 #set counter to monitor acceptance rate
602     for(i in 1:M) {
603         if(seen[i]) #no need to update seen individuals, since their z =1
604             next
605         zcand <- ifelse(z[i]==0, 1, 0)
606         llz <- sum(dpois(y[i,], lam[i,]*z[i], log=TRUE))
607         llcand <- sum(dpois(y[i,], lam[i,]*zcand, log=TRUE))
608
609         prior <- dbinom(z[i], 1, psi, log=TRUE)
610         prior.cand <- dbinom(zcand, 1, psi, log=TRUE)
611         if(runif(1) < exp( (llcand+prior.cand) - (llz+prior) )) {
612             z[i] <- zcand
613             zUps <- zUps+1
614         }
615     }

```

616 Psi itself is a hyperparameter of the model, with an uninformative prior  
617 distribution of Unif(0,1) or Beta(1,1), so that

$$Psi|z \propto [z|psi] * Beta(1,1)$$

618 The Beta distribution is the conjugate prior to the Binomial and Bernoulli  
619 distributions (remember that  $z \sim Bernoulli(psi)$ ). The general form of a full  
620 conditional of a Beta-Binomial model with  $y_i \sim Bernoulli(p)$  and  $p \sim Beta(a, b)$   
621 is

$$p(p|y) \propto Beta(a + \text{sum}(y_i), b + n - \text{sum}(y_i))$$

622 In our case, this means we update psi as follows:

```

623 si<-rbeta(1, 1+sum(z), 1 + M-sum(z))

```

624 These are all the building blocks you need to write the MCMC algorithm  
625 for the spatial null model with a Poisson encounter process. You can find the  
626 full R code (SCR0pois.R) in the online supplementary material.

### 627 1.6.1 SCR model with binomial encounter process

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

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

631 Where  $p_{ij}$  is some function of distance between  $s_i$  and trap location  $x_j$ . Here  
632 we use:

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

Recall from Chapter 2 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,  $Poisson(y|\sigma, \lambda_0, s, z)$ , we use a Binomial likelihood,  $Binomial(y, K|\sigma, \lambda_0, s, z)$ , in all the conditional distributions. As an example, Panel 6 shows the updating step for  $\lambda_0$  under a binomial encounter model. The full MCMC code for the binomial SCR can be found in the online supplements.

```

640 Panel 6: MCMC updater for  $\lambda_0$  in a SCR model with Binomial encounter process and cloglog
641      $\lambda_0.cand \leftarrow rnorm(1, \lambda_0, 0.1)$ 
642     if( $\lambda_0.cand > 0$ ){ #automatically reject  $\lambda_0.cand$  that are  $< 0$ 
643          $\lambda.cand \leftarrow \lambda_0.cand * \exp(-(D*D)/(2*\sigma*\sigma))$ 
644          $p.cand \leftarrow 1 - \exp(-\lambda.cand)$ 
645          $ll \leftarrow \text{sum}(\text{dbinom}(y, K, pmat * z, \text{log}=\text{TRUE}))$ 
646          $llcand \leftarrow \text{sum}(\text{dbinom}(y, K, p.cand * z, \text{log}=\text{TRUE}))$ 
647         if( $\text{runif}(1) < \exp(llcand - ll)$  ){
648              $ll \leftarrow llcand$ 
649              $pmat \leftarrow p.cand$ 
650              $\lambda_0 \leftarrow \lambda_0.cand$ 
651         }
652     }

```

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

```

655  $p_{ij} \leftarrow p_0 * \exp(-D_{ij}^2/(2*\sigma^2))$ 

```

and  $p_0 = \{0, 1\}$ . 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, 2001). Under this formulation the updater for  $\lambda_0$  (equivalent to  $p_0$  in Eq XX) becomes:

```

661      $\lambda_0.cand \leftarrow rnorm(1, \lambda_0, 0.1)$ 
662     if( $\lambda_0.cand > 0 \ \& \ \lambda_0.cand < 1$  ){ #automatically reject  $\lambda_0.cand$  that are
663          $\lambda.cand \leftarrow \lambda_0.cand * \exp(-(D*D)/(2*\sigma*\sigma))$ 
664          $ll \leftarrow \text{sum}(\text{dbinom}(y, K, \lambda * z, \text{log}=\text{TRUE}))$  #no transformation needed
665          $llcand \leftarrow \text{sum}(\text{dbinom}(y, K, \lambda.cand * z, \text{log}=\text{TRUE}))$ 
666         if( $\text{runif}(1) < \exp(llcand - ll)$  ){
667              $ll \leftarrow llcand$ 
668              $\lambda \leftarrow \lambda.cand$ 
669              $\lambda_0 \leftarrow \lambda_0.cand$ 
670         }
671     }

```

## 1.6.2 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 ex-

ample, we will use the bear data ... <sup>8</sup> You can use the same script provided back in Chapter XX to read in the data and build the augmented encounter history array; then source the MCMC code for the binomial encounter model algorithm with the cloglog link and run 5000 iterations. This should take approximately 25 minutes.

```
> source('SCR0binom.txt')
> mod0<-SCR.0(y=bigTrap, X=trapmat, M=M, xl=xl, xu=xu, yl=yl, yu=yu, K=8, 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.

### 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 (Fig. 5). Time series plots will tell you several things: First, 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 probably 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 5, we would probably consider the first 750 - 1000 iterations as burn-in, as afterwards the chains seem to be fairly stationary.

### A word of caution about chain convergence

Since we do not know what the stationary posterior distribution of our Markov chain should look like (this is the whole point of doing an MCMC approximation), we effectively have no means to assess whether it has converged to this desired distribution or not. As mentioned before, the only certainty is that a Markov chain will *eventually* converge to its stationary distribution, but no-one can tell us how long this will take. Also, you only now the part of your posterior distribution that the Markov chain has explored so far for all you know the chain could be stuck in a local maximum, while other maxima remain completely undiscovered. Acknowledging that there is truly nothing we can do to

---

<sup>8</sup>Does this data set come up before Ch6? If not, introduce data here. Or, Andy, would you rather use simulated data?

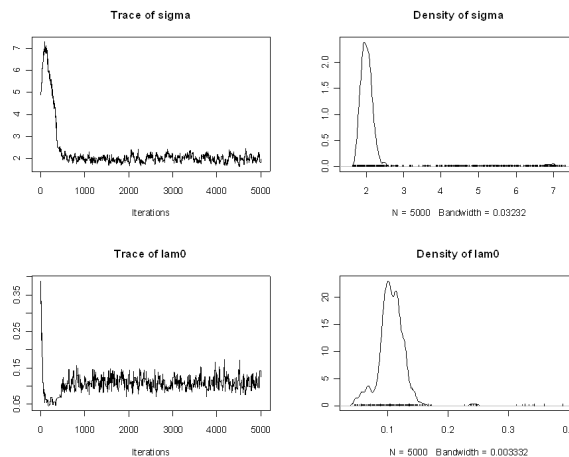


Figure 1.5: Time series and posterior density plots for sigma and lam0.

ever proof convergence of our MCMC chains, there are several things we can do  
 to increase the degree of confidence we have about the convergence of our chains.  
 One option, and that advocated by what we will loosely call the WinBUGS com-  
 munity, is to run several Markov chains and to start them off at different initial  
 values that are overdispersed relative to the posterior distribution. Such initial  
 values help to explore different areas of the parameter space simultaneously;  
 if after a while all chains oscillate around the same average value, chances are  
 good that they indeed converged to the posterior distribution. Gelman and Ru-  
 bin came up with a diagnostic statistic that essentially compares within-chain  
 and between-chain variance to check for convergence of multiple chains (Gel-  
 man et al., 2004). Of course, running several parallel chains is computationally  
 expensive. Extra computational demands are not the only and by no means  
 the major concern some people voice when it comes to running several parallel  
 MCMC chains to assess convergence. Again, consider the fact that we do not  
 know anything about the true form of the posterior distribution we are trying to  
 approximate. How do we, then, know how to pick overdispersed initial values?  
 We don't all we can do is pick overdispersed values relative to our expectations  
 of what the posterior should look like. To use a quote from the home page  
 of Charlie Geyer, a Bayesian statistician from the University of Minnesota, "If  
 you don't know any good starting points [...], then restarting the sampler at  
 many bad starting points is [...] part of the problem, not part of the solution."  
 (<http://users.stat.umn.edu/~charlie/mcmc/diag.html>). His suggestion is that  
 your only chance to discover a potential problem with your MCMC sampler is  
 to run it for a very long time. But again, there is no way of knowing how long  
 is long enough. It is up to you to decide, which school of thoughts appeals more  
 to you one long versus several parallel Markov chains. Irrespectively, part of

developing an MCMC sampler should be to make sure (within reasonable limits) that you are not missing regions of high posterior density because of the way you specify your starting values. Once you have explored the behavior of your chain under a reasonable range of starting values, you may feel comfortable enough to run only one long chain. The fact that convergence cannot be proven does not mean that you should not look for potential problems in your MCMC sampler. Some problems are easily detected using simple plots, such as the time series plots we discussed above. If the overall trajectory of your chain at the end of your simulations is still upward or downward, your chain clearly has not converged and you need to run your model much longer. If you run several parallel chains and their stationary distributions look different, you may be looking at a multi-modal posterior or a problem with your sampler. With these words of caution, let's get back to looking at our model output.

### 1.6.3 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  $\psi$  has a lot of mass concentrated close to 1), as in the example in Figure 6, we have to re-run the analysis with a larger  $M$ . A flat posterior plot shows you that the parameter essentially cannot be identified there may not be enough information in your data to estimate model parameters and you may have to consider a simpler model. 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.

### 1.6.4 Serial autocorrelation and effective sample size

Even when we can be relatively confident that our chains have converged, the subsequent samples generated from a Markov chain are not iid samples from the posterior distribution, due to the correlation amongst samples introduced by the Markov process. As a consequence, the variance of the mean cannot simply be derived with the standard variance estimator, which takes into account the sample size (here, number of iterations). Rather, the sample size has to be adjusted to account for the autocorrelation in subsequent samples (see Chapter 8 in ? for more details). This adjusted sample size is referred to as the effective sample size. Checking the degree of autocorrelation in your Markov chains and estimating the effective sample size your 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

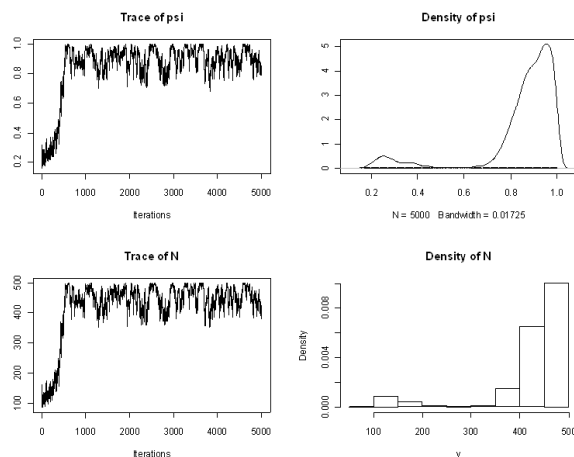


Figure 1.6: Time series and posterior density plots of  $\psi$  and  $N$  for the beaver data set truncated by the upper limit of  $M$  (500).

you can use to do so. `effectiveSize()` will directly give you an estimate of the effective sample size for your parameters:

```
> effectiveSize(chain)
      sigma      lam0      psi      N
3.930303 78.259159 30.436348 32.047392
```

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):

```
> autocorr.diag(mcmc(mod))
      sigma      lam0      psi      N
Lag 0  1.0000000 1.0000000 1.0000000 1.0000000
Lag 1  0.9979948 0.9494134 0.9847503 0.9774201
Lag 5  0.9915567 0.8038168 0.9111951 0.9113525
Lag 10 0.9836016 0.6714021 0.8462108 0.8509803
Lag 50 0.8985337 0.1983780 0.6138516 0.6233994
```

Whichever function you use, if you find that your supposedly long Markov chain has not generated enough pseudo-iid samples, you should consider a longer run. In the present case we see that autocorrelation is especially high for the parameter `sigma` and our effective sample size for this parameter is 4! <sup>9</sup> This means we would have to run the model for much longer to obtain a reasonable effective sample size. Unfortunately, with many SCR models we observe high

<sup>9</sup>Anyone have any idea how the autocorrelation in `sigma` could be reduced?



degrees of serial autocorrelation, which means we have to run long chains to obtain enough samples that can be considered iid, in order to obtain reasonable estimates of our parameters and their variances. What exactly constitutes a reasonable effective sample size is hard to say, but as a rule of thumb you should probably aim at several hundreds of these pseudo-iid samples. A more meaningful measure of whether you've run your chain for enough iterations is the time-series or Monte Carlo error - the 'noise' introduced into your samples by the stochastic MCMC process - which we introduced in Chapter 2. The MC error decreases with increasing sample size and its magnitude can thus be controlled by adjusting the length of the Markov chain. As a rule of thumb, the MC error should be 1% or less of the parameter estimate. Once you have reached this level, the estimates of the mean, standard error and 95% quantiles should no longer change significantly with additional iterations. For highly correlated samples, it will take more iterations to reduce the MC error. In coda, the MC error is given as part of the summary results (see below). Another option to deal with the serial autocorrelation of samples is to 'thin' Markov chains by some rate  $r$  and save only every  $r$ -th iteration. But as discussed in Chapter 2, this is not efficient and should only be applied if needed for practical reasons (e.g. a large number of parameters and iterations may force you to thin your samples so you object storing the model output does not become unmanageably large). For now, let's continue using this small set of samples to continue looking at the output.

### 1.6.5 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  $K$  (= number of iterations) samples without accounting for serial autocorrelation - and the corrected MC error (Time-series SE), which accounts for autocorrelation. In WinBUGS, this latter value is referred to as MC error and is only given in the log output within BUGS itself. 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
```

```

845
846 1. Empirical mean and standard deviation for each variable,
847    plus standard error of the mean:
848

```

	Mean	SD	Naive SE	Time-series SE
sigma	1.9986	0.13805	0.0021827	0.016091
lam0	0.1096	0.01523	0.0002407	0.001401
psi	0.6113	0.09148	0.0014465	0.010734
N	489.8535	71.79695	1.1352094	8.431119

```

854
855 2. Quantiles for each variable:
856

```

	2.5%	25%	50%	75%	97.5%
sigma	1.75780	1.89847	1.9900	2.0944	2.2772
lam0	0.08357	0.09824	0.1087	0.1192	0.1427
psi	0.45110	0.54838	0.6052	0.6639	0.8192
N	366.00000	440.00000	485.0000	530.0000	654.0000

```

862
863 Looking at the MC errors, we see that in spite of the high autocorrelation, the
864 MC error for sigma is below the 1Our algorithm gives us a posterior distribution
865 of N, but we are usually interested in the density, D. Density itself is not a
866 parameter of our model, but we can derive a posterior distribution for D by
867 dividing each value of N (N at each iteration) by the area of the state-space
868 (here 3032.719 km2) and we can use summary statistics of this distribution to
characterize D:

```

```

869 > summary(window(chain[,4]/ 3032.719, start=1001))
870 Iterations = 1001:5000
871 Thinning interval = 1
872 Number of chains = 1
873 Sample size per chain = 4000
874

```

```

875 1. Empirical mean and standard deviation for each variable,
876    plus standard error of the mean:
877

```

	Mean	SD	Naive SE	Time-series SE
	0.1615229	0.0236741	0.0003743	0.0027801

```

880
881 2. Quantiles for each variable:
882

```

	2.5%	25%	50%	75%	97.5%
	0.1207	0.1451	0.1599	0.1748	0.2156

```

885 If we compare our mean density of 0.16/km2 (and other parameters) with results
886 from the same model run in secr and WinBUGS in Chapter XX, we see that
887 estimates are almost identical (Table 1).

```

### 1.6.6 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
0.44108822 0.77675535 0.00000000 0.01940388
```

Recall that rejection rates should lie between 0.2 and 0.8, so our tuning seems to have been appropriate here. Psi is never rejected since we update it with Gibbs sampling, where all candidate values are kept. And since N is the sum of all z, 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 R-hat, using `gelman.diag()`, which will indicate if all chains have converged to the same stationary distribution. For details on these and other functions, see the coda manual, which can be found together with the package on the CRAN mirror.

## 1.7 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 S has an irregular shape maybe there is a large water body we would like to remove from 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 WinBUGS. In some simple cases we can adjust the state space by setting `SXi` to be some function of `SYi` or vice versa. In this manner, we can cut off corners of the rectangle to approximate the actual state space. In R, we are much more flexible, as we can use the actual state-space polygon to constrain out si. <sup>10</sup>To illustrate that, let's look at a camera trapping study of Florida panthers (*Puma concolor coryi*) conducted in the Picayune Strand Restoration Project (PSRP) area, southwest Florida (Fig. 7), by XXX, and financed by XXX. In the 1960ies the PSRP area was slated for housing development, but then bought back by the State of Florida and is currently being restored to its original hydrology and vegetation. In an effort to estimate the density of the local Florida panther population, 98 camera traps were operated in the area for 21 months between 2005 and 2007. Florida panthers are wide-ranging animals and in order to account for their wide movements, the state-space was defined as

<sup>10</sup> Have to check if we can use panther stuff for the book; otherwise, use raccoon example.

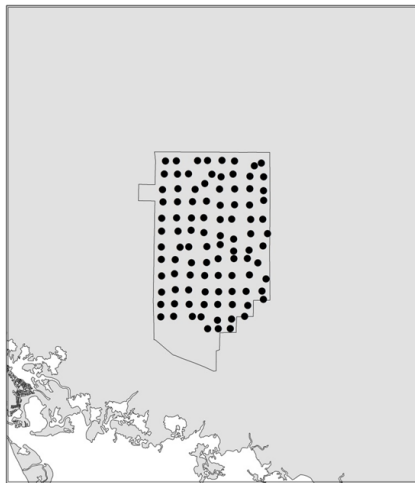


Figure 1.7: Rectangular state-space for a Florida panther camera trapping study in the PSRP area (grey outline, red block inset map of Florida) contain some ocean (white) that needs to be removed from the state-space.

the trapping grid buffered by 15 km around its outermost coordinates. However, the resulting rectangle contained some ocean in its southwestern corner (Fig. 7). In order to precisely describe the state-space, the ocean has 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. In the present case we intersected two shape files one of the state of Florida and one of the rectangle defined by a strip of 15 km around the camera-trapping grid. 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, a polygon shapefile buffering your outermost trapping grid coordinates can easily be written in R.

If `xmin`, `xmax`, `ymin` and `ymax`, mark the outermost `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 use:

```

941 xl= xmin-b
942 xu= xmax+b
943 yl= ymin-b
944 yu= ymax+b
945
946 dd <- data.frame(Id=c(1,1,1,1,1),X=c(xl,xu,xu,xl,xl),Y=c(yl,yl,yu,yu,yl)) #create data
947 ddTable <- data.frame(Id=c(1),Name=c("Item1"))
948 ddShapefile <- convert.to.shapefile(dd, ddTable, "Id", 5) #convert #to shapefile, type
949 write.shapefile(ddShapefile, 'c:/, arcgis=T) # save to location of #choice

```

You can read shapefiles into 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 `sigma` 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 1000000 will return the area in km<sup>2</sup> if your coordinates describing the polygon are in UTM. If your state-space consists of several disjunct 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. All we have to do is embed this new check into the updating steps for `s`:

```
Scand <- as.matrix(cbind(rnorm(M, S[,1], 2),
                          rnorm(M, S[,2], 2)))      #draw candidate value
Scoord<-SpatialPoints(Scand*1000)      #convert to spatial points on UTM (m) scale
SinPoly<-over(Scoord,SSp) # check if scand is within the polygon
for(i in 1:M) {
  if(is.na(SinPoly[i])==FALSE) { #if scand falls within polygon, continue update
    [rest of the updating step remains the same]
```

Note that it is much more time-efficient to draw all `M` candidate values for `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 `s` also fall within the polygon of `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 in the online supplementary material (SCR0poisSSp). Finally, observe that we are converting candidate coordinates of `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 `sigma` on a km scale. This is adequate for wide ranging individuals 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 `S` back to the original scale (UTM) before running the `over()` command.

## 990 1.8 MCMC software packages

991 Throughout most of this book we will use WinBUGS and, occasionally, JAGS  
 992 to run MCMC analyses. Here, we will briefly discuss the main pros and cons of  
 993 these two programs as well as WinBUGS successor OpenBUGS. You can find  
 994 scripts to simulate data and run the basic SCR model in all three programs in  
 995 the online supplementary material (simSCR0poisBUGS).

### 996 1.8.1 WinBUGS

997 In a nutshell, WinBUGS (and the other programs) do everything that we just  
 998 went through in this chapter (and quite a bit more). Looking through your  
 999 model, WinBUGS determines which parameters it can use standard Gibbs sam-  
 1000 pling for (i.e. for conjugate full conditional distributions). Then, it determines,  
 1001 in the following hierarchy, whether to use adaptive rejection sampling, slice  
 1002 sampling or in the 'worst' case Metropolis-Hastings sampling for the other full  
 1003 conditionals (Spiegelhalter et al., 2003). If it uses MH sampling, it will auto-  
 1004 matically tune the updater so that it works efficiently. While WinBUGS is a  
 1005 convenient piece of software that is still widely used, its major drawback is that  
 1006 it is no longer being developed, i.e. no new functions or distributions are added  
 1007 and no bugs are fixed.

### 1008 1.8.2 OpenBUGS

1009 OpenBUGS is essentially the successor of WinBUGS. While the latter is no  
 1010 longer worked on, OpenBUGS is constantly developed further. The name  
 1011 'OpenBUGS' refers to the software being open source, so users do not need  
 1012 to download a license key, like they have to for WinBUGS (although the license  
 1013 key for WinBUGS is free and valid for life).

1014 Compared to WinBUGS, OpenBUGS has a lot more built-in functions. The  
 1015 method of how to determine the right updater for each model parameter has  
 1016 changed and the user can manually control the MCMC algorithm used to update  
 1017 model parameters. Several other changes have been implemented in OpenBUGS  
 1018 and a detailed list of differences between the two BUGS versions, can be found  
 1019 at <http://www.openbugs.info/w/OpenVsWin>

1020 While OpenBUGS is a useful program for a lot of MCMC sampling appli-  
 1021 cations, for reasons we do not understand, simple SCR models do not converge  
 1022 in OpenBUGS. It is therefore advisable that you check any OpenBUGS SCR  
 1023 model results against result from WinBUGS. Also, currently, the R package  
 1024 BRugs (Thomas et al., 2006) necessary for running OpenBUGS through R  
 1025 has problems with 64-bit machines, so you may have to use the 32-bit version  
 1026 of R and OpenBUGS in order to make it work. The BUGS project site at  
 1027 <http://www.openbugs.info> provides a lot of information on and download links  
 1028 for OpenBUGS.

1029 There is an extensive help archive for both WinBUGS and OpenBUGS and  
 1030 you can subscribe to a mailing list, where people pose and answer questions of

1031 how to use these programs at <http://www.mrc-bsu.cam.ac.uk/bugs/overview/list.shtml>

### 1032 1.8.3 JAGS Just Another Gibbs Sampler

1033 JAGS, currently at Version 3.1.0, is another free program for analysis of Bayesian  
 1034 hierarchical models using MCMC simulation. Originally, JAGS was the only  
 1035 program using the BUGS language that would run on operating systems other  
 1036 than the 32 bit Windows platforms. By now, there are OpenBUGS versions for  
 1037 Linux or Macintosh machines. JAGS 'only' generates samples from the posterior  
 1038 distribution; analysis of the output is done in R either by running JAGS through  
 1039 R using either the packages `rjags` (Plummer, 2011) or `R2jags` (Su and Yajima,  
 1040 2011), or by using `coda` on your JAGS output. The program, manuals and `rjags`  
 1041 can be downloaded at <http://sourceforge.net/projects/mcmc-jags/files/> When  
 1042 run from within R using the package `rjags` or `R2jags`, writing a JAGS model is  
 1043 virtually identical to writing a WinBUGS model. However, some functions may  
 1044 have slightly different names and you can look up available functions and their  
 1045 use in the JAGS manual. One potential downside is that JAGS can be very  
 1046 particular when it comes to initial values. These may have to be set as close  
 1047 to truth as possible for the model to start. Although JAGS lets you run several  
 1048 parallel Markov chains, this characteristic interferes with the idea of using  
 1049 overdispersed initial values for the different chains. Also, we have occasionally  
 1050 experienced JAGS to crash and take the R GUI with it. Only re-installing both  
 1051 JAGS and `rjags` seemed to solve this problem. On the plus side, JAGS usually  
 1052 runs a little faster than WinBUGS, sometimes considerably faster (see section  
 1053 4.XYZ), is constantly being developed and improved and it has a variety of  
 1054 functions that are not available in WinBUGS. For example, JAGS allows you  
 1055 to supply observed data for some deterministic functions of unobserved vari-  
 1056 ables. In BUGS we cannot supply data to logical nodes. Another useful feature  
 1057 is that the adaptive phase of the model (the burn-in) is run separately from  
 1058 the sampling from the stationary Markov chains. This allows you to easily add  
 1059 more iterations to the adaptive phase if necessary without the need to start  
 1060 from 0. There are other, more subtle differences and there is an entire manual  
 1061 section on differences between JAGS and OpenBUGS. For questions and prob-  
 1062 lems there is a JAGS forum online at [http://sourceforge.net/projects/mcmc-](http://sourceforge.net/projects/mcmc-jags/forums/forum/610037)  
 1063 [jags/forums/forum/610037](http://sourceforge.net/projects/mcmc-jags/forums/forum/610037).<sup>11</sup>

## 1064 1.9 Summary and Outlook

1065 While there are a number of flexible and extremely useful software packages  
 1066 to perform MCMC simulations, it sometimes is more efficient to develop your  
 1067 own MCMC algorithm. Building an MCMC code follows three basic steps:  
 1068 Identify your model including priors and express full conditional distributions  
 1069 for each model parameter. If full conditionals are parametric distributions, use

---

<sup>11</sup>As we make progress on the book, let's be sure to add linkages to places where we use JAGS in examples.

1070 Gibbs sampling to draw candidate parameter values from this distributions;  
 1071 otherwise use Metropolis-Hastings sampling to draw candidate values from a  
 1072 proposal distribution and accept or reject them based on their posterior proba-  
 1073 bility densities. These custom-made MCMC algorithms give you more modeling  
 1074 flexibility than existing software packages, especially when it comes to handling  
 1075 the state-space: In BUGS (and JAGS for that matter) we define a continuous  
 1076 rectangular state-space using the corner coordinates to constrain the Uniform  
 1077 priors on the activity centers  $s$ . But what if a continuous rectangle isn't an ad-  
 1078 equate description of the state-space? In this chapter we saw that in R it only  
 1079 takes a few lines of code to use any arbitrary polygon shapefile as the state-  
 1080 space, which is especially useful when you are dealing with coastlines or large  
 1081 bodies of water that need removing from the state-space. Another example is  
 1082 the SCR R package SPACECAP (Gopalaswamy et al., 2011) that was developed  
 1083 because implementation of an SCR model with a discrete state-space was inef-  
 1084 ficient in WinBUGS. Another situations in which using BUGS/JAGS becomes  
 1085 increasingly complicated or inefficient is when using point processes other than  
 1086 the Uniform Poisson point process which underlies the basic SCR model (see  
 1087 Chapter X). In the Chapters 9 and XX you will see examples of different point  
 1088 processes, implemented using custom-made MCMC algorithms.<sup>12</sup> Finally,  
 1089 the Chapters XX and XX deal with unmarked or partially marked populations  
 1090 using hand-made MCMC algorithms to handle the (partially) latent individual  
 1091 encounter histories. While some of these models can be written in BUGS/JAGS,  
 1092 <sup>13</sup>, they are painstakingly slow; others cannot be implemented in BUGS/JAGS  
 1093 at all. In conclusion, while you can certainly get by using BUGS/JAGS for  
 1094 standard SCR models, knowing how to write your own MCMC sampler allows  
 1095 you to tailor these models to your specific needs.

---

<sup>12</sup>Richard, Beth expand on that?

<sup>13</sup>the Poisson one for partially marked we wrote in BUGS and it should work with a known number of marked; the Bernoulli in JAGS with the `dsum()` function should work for the fully unknown; maybe some others? I dont remember. We may have to try writing the others before saying that they dont work in BUGS/JAGS; they are certainly much faster in R, though.



# Bibliography

- Baddeley, A. and Turner, R. (2005), “Spatstat: an R package for analyzing spatial point patterns,” *Journal of Statistical Software*, 12, 1–42, ISSN 1548-7660.
- Bivand, R. and Rundel, C. (2011), *rgeos: Interface to Geometry Engine - Open Source (GEOS)*, r package version 0.1-8.
- Buckland, S. T. (2001), *Introduction to distance sampling: estimating abundance of biological populations*, Oxford, UK: Oxford University Press.
- Casella, G. and George, E. I. (1992), “Explaining the Gibbs sampler,” *American Statistician*, 46, 167–174.
- Gelfand, A. and Smith, A. (1990), “Sampling-based approaches to calculating marginal densities,” *Journal of the American statistical association*, 85, 398–409.
- Gelman, A., Carlin, J. B., Stern, H. S., and Rubin, D. B. (2004), *Bayesian data analysis, second edition.*, Boca Raton, Florida, USA: CRC/Chapman & Hall.
- Geman, S. and Geman, D. (1984), “Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6, 721–741.
- Gilks, W. and Wild, P. (1992), “Adaptive rejection sampling for Gibbs sampling,” *Applied Statistics*, 41, 337–348.
- Gilks, W. R., Thomas, A., and Spiegelhalter, D. J. (1994), “A Language and Program for Complex Bayesian Modelling,” *Journal of the Royal Statistical Society. Series D (The Statistician)*, 43, 169–177, ArticleType: primary\_article / Issue Title: Special Issue: Conference on Practical Bayesian Statistics, 1992 (3) / Full publication date: 1994 / Copyright 1994 Royal Statistical Society.
- Gopalaswamy, A. M., Royle, A. J., Hines, J., Singh, P., Jathanna, D., Kumar, N. S., and Karanth, K. U. (2011), *A Program to Estimate Animal Abundance and Density using Spatially-Explicit Capture-Recapture*, r package version 1.0.4.

- 1127 Hastings, W. (1970), “Monte Carlo sampling methods using Markov chains and  
1128 their applications,” *Biometrika*, 57, 97–109.
- 1129 Lewin-Koh, N. J., Bivand, R., contributions by Edzer J. Pebesma, Archer, E.,  
1130 Baddeley, A., Bibiko, H.-J., Dray, S., Forrest, D., Friendly, M., Giraudoux, P.,  
1131 Golicher, D., Rubio, V. G., Hausmann, P., Hufthammer, K. O., Jagger, T.,  
1132 Luque, S. P., MacQueen, D., Niccolai, A., Short, T., Stabler, B., and Turner,  
1133 R. (2011), *maptools: Tools for reading and handling spatial objects*, r package  
1134 version 0.8-10.
- 1135 Link, W. A. and Barker, R. J. (2009), *Bayesian Inference: With Ecological*  
1136 *Applications*, London, UK: Academic Press.
- 1137 Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., Teller, E., et al.  
1138 (1953), “Equation of state calculations by fast computing machines,” *The*  
1139 *journal of chemical physics*, 21, 1087–1092.
- 1140 Metropolis, N. and Ulam, S. (1949), “The Monte Carlo method,” *Journal of the*  
1141 *American Statistical Association*, 44, 335–341.
- 1142 Neal, R. (2003), “Slice sampling,” *Annals of Statistics*, 31, 705–741.
- 1143 Pebesma, E. and Bivand, R. (2011), *Package ‘sp’*, r package version 0.9-91.
- 1144 Plummer, M. (2011), *rjags: Bayesian graphical models using MCMC*, r package  
1145 version 3-5.
- 1146 Plummer, M., Best, N., Cowles, K., and Vines, K. (2006), “CODA: Convergence  
1147 Diagnosis and Output Analysis for MCMC,” *R News*, 6, 7–11.
- 1148 Robert, C. P. and Casella, G. (2004), *Monte Carlo statistical methods*, New  
1149 York, USA: Springer.
- 1150 — (2010), *Introducing Monte Carlo Methods with R*, New York, USA: Springer.
- 1151 Roberts, G. O. and Rosenthal, J. S. (1998), “Optimal scaling of discrete ap-  
1152 proximations to Langevin diffusions,” *Journal of the Royal Statistical Society:*  
1153 *Series B (Statistical Methodology)*, 60, 255–268.
- 1154 Spiegelhalter, D., Thomas, A., Best, N., and Lunn, D. (2003), *WinBUGS User*  
1155 *Manual Version 1.4*.
- 1156 Stabler, B. (2006), *shapefiles: Read and Write ESRI Shapefiles*, r package version  
1157 0.6.
- 1158 Su, Y.-S. and Yajima, M. (2011), *R2jags: A Package for Running jags from R*,  
1159 r package version 0.02-17.
- 1160 Thomas, A., O’Hara, B., Ligges, U., and Sturtz, S. (2006), “Making BUGS  
1161 Open,” *R News*, 6, 12–17.