Distance sampling: methods and applications

Bayesian methods, Northern bobwhite case study; Cornelia Oedekoven

Contents

We use the case study of northern bobwhite quail coveys to demonstrate the methods described in Section 8.4 *Model fitting: Bayesian Methods* of the book. We present the R code for analysing distance sampling data using Bayesian methods including a Metropolis-Hastings algorithm and an RJMCMC algorithm. For comparison, we demonstrate the methods and R code for implementing maximum likelihood methods for the full likelihood approach from Section 8.3 *Model fitting: maximum likelihood methods* of the book. For this approach we refrain from including model selection. However, we emphasize the differences in the likelihood formulations for the Bayesian and maximum likelihood approaches. Although the presented code is specific to the covey data, we use the block quote to advise the users how they may adjust the code according to their needs.

> A block quote example where advice about altering code will be provided.

The R code provided below consists of commented scripts that the user may run in their R workspace to complete the exercise. In addition, we present the R functions written for this exercise. The user can upload these at the beginning of the exercise using the *source* commands below and does not need to run these again when presented in the script.

1 The data

The covey data stem from a designed experiment where the interest was in determining whether a conservation measure - planting herbaceous buffers around agricultural fields - had the desired effect. For this purpose, pairs of survey points were set up in several states in the US, one point on a buffered treatment field, one point on an unbuffered control field. Pairs of points were surveyed at least once but up to three times per year in autumn from 2006 to 2008.

As the interest was in determining whether covey densities were higher on treatment fields, we use plot count models where the observed counts are modelled as a function of covariates. Here, the parameter of interest is the coefficient for the *Type* covariate for the level "TREAT". A coefficient that is significantly greater than zero would indicate that the buffers had the desired effect, i.e., increased covey densities. Due to the repeated counts and the spatial closeness of points within a pair, we include a random effect for each site (where 'site' refers to a pair of points) in the count model (see below for details). For simplicity, we reduced the data presented in Section 8.5.2 of the book to four states leaving 183 sites. In addition, we did not include the covariate *Year* for the detection or count models. Hence, the resulting parameter estimates are not directly comparable to those presented in Section 8.5.2.

1.1 Importing the data and functions into an R Studio workspace

For completing this case study, we need to download the zip file CoveyCaseStudy.zip and extract these into a designated folder on our personal computers. Source the functions necessary for the case study, by running the following R code.

```
source("bl.function.r")
source("covey.full.bayes.log.lik.r")
source("covey.full.bayes.log.lik.raneff.update.r")
source("covey.ml.log.lik.r")
source("covey.ml.log.lik2.r")
source("create.data.r")
source("create.glmm.data.r")
source("create.glmm.data.unconditional.r")
source("f.haz.function.ln.r")
source("f.haz.function.pt.r")
source("f.hn.function.ln.r")
source("f.hn.function.pt.r")
source("l.prior.r")
source("match.function.r")
source("var.Dblml.r")
source("which.bin.r")
```

In addition to these functions, we need read the data, provided as a `.csv` file:

```
covey<-read.csv("covey.csv")
```

1.2 Preparing the data for analyses

We begin by ensuring that the data have the correct format required for using the likelihood functions in the following sections. For this purpose, we first look at the raw data contained in the `covey` data frame.

```
kable(covey[1:3, ], digits=1)
```

gr.id	smp.id	Repetition	Year	State	Type	JDctr	det.id	distance	vis.id	size
183	365	1	2006	MO	CONTROL	0	1	401.6	2006_1	1
183	365	1	2006	MO	CONTROL	0	2	286.2	2006_1	1
183	365	1	2006	MO	CONTROL	0	3	327.9	2006_1	1

The columns represent the locations or sites (***gr.id***), point (***smp.id***), visit (***vis.id***), detection (***det.id***), distance to the detection (***distance***) and group size of the detection (***size***). These are the columns that are required for using the data formatting and likelihood functions below. Note that for the covey data group sizes were unknown and we model numbers of detections rather than number of individuals. However, the ***size*** column is required by the following functions. In addition to the required columns, we have covariates ***State***, ***Type*** and ***JDctr*** which were measured at the level of a visit to a point (as opposed to the level of individual detections). Covariate ***JDctr*** represents Julian date which was centred around its mean.

Each record represents either a detection or a visit without detection in the case that no coveys were detected during the respective visit to the point. The data are already truncated at 500m, i.e., the largest distance included in the *covey* data is 500m or less.

```
max(covey$distance,na.rm=T)
```

```
[1] 497.58
```

We use the covey case study function `create.data` to turn the raw covey data into a format that allows us to fit the detection and count models below. For the detection model, we need a data frame (which we call `dis.object`) which contains all the records of detections within the truncation distance including their distances and covariates that may be included in the detection model. For the count model, we need a data frame (which we call `glmm.data`) where each record represents a visit to a point and the total number of detections within the truncation distance during that visit to the point is

recorded. The data frame `glmm.data` contains some columns that are required such as a unique identifier for each visit to a point (`smp.vis`), `smp.id`, `gr.id` and `vis.id` from before, as well as the columns `detections` and `individuals` (which give the total number of detections and detected individuals for each visit to a point, respectively) and the covariates that might be used in the count model, ***Type***, ***JDctr*** and ***State***.

Both the `dis.object` and `glmm.data` data frames are created by the function `create.data` and combined in a list containing additional information. The additional information includes the type of sampler the data related to (points or lines), the units of distance measurements, whether we are using binned distance data, which covariates may be used in the detection and count models and which of these should be regarded as a factor covariate and what type of analysis we would like to use the data for. The argument `conditional` refers to whether we use the conditional likelihood (FALSE refers to the methods using equations 8.25 and 8.27 of the book and only applies to binned distance data).

```
# creating a data object which can be used with the functions below
covey.data<-create.data(covey,500,sampler="points","m",binned=F,
  dis.cov=c("Type","State"),count.cov=c("Type","JDctr","State"),
  dis.factor.cov = c("Type","State"), count.factor.cov = c("Type","State"),
  conditional = T)
```

```
# to have a look at what covey.data contains
str(covey.data, max.level=1)
```

```
List of 12
 $ w            : num 500
 $ sampler      : chr "points"
 $ dis.unit     : chr "m"
 $ sampler.unit : chr "m"
 $ binned       : logi FALSE
 $ dis.cov      : chr [1:2] "Type" "State"
 $ dis.cov.fac  : logi [1:2(1d)] TRUE TRUE
 $ count.cov    : chr [1:3] "Type" "JDctr" "State"
 $ count.cov.fac: logi [1:3(1d)] TRUE FALSE TRUE
 $ conditional  : logi TRUE
 $ dis.object   :'data.frame':  1023 obs. of  12 variables:
 $ glmm.data    :'data.frame':  1051 obs. of  9 variables:
```

To analyse distance data that were collected in predefined distance bins, the argument `binned` of the `create.data` function needs to be set to `TRUE` and cutpoints of the distance intervals defined using the argument `cutpoints`. The column `distance` needs to contain a distance from within the distance bin to which the detection belongs (e.g. the mid-point of the bin). The bins are automatically assigned and the columns `bin`, `distbegin` and `distend` added to the data frame `dis.object`. The column `bin` refers to the bin number while `distbegin` and `distend` refer to the cutpoints of the respective bin. The user may also use these settings to analyse exact distance data as binned.

2 Setting up a Metropolis Hastings updating algorithm

Setting up a Metropolis Hastings (MH) updating algorithm involves several steps. These include:

1. Defining the likelihood function
2. Defining the prior probabilities for the parameters
3. Proposal distributions for updating parameters in the MH algorithm
4. Initial values for the MH updating algorithm
5. Storing the parameter values for the MH updating algorithm
6. The MH updating algorithm
7. Pilot tuning the MH updating algorithm

In this section, we provide the functions and R code necessary to implement these steps. In addition, we describe how to obtain summary statistics of the posterior distributions of parameters and plot density. We begin with the theory for this algorithm.

We use a single-update random walk MH algorithm with normal proposal density, where we cycle through each parameter in the full likelihood $\mathcal{L}_{n,y|z}$ (see below for the full likelihood used for our case study). During each iteration of the MCMC algorithm, we update each of the parameters by proposing to move to a new state and accepting this move with some probability. To update, say, parameter β_0 (contained in the model with parameters β_0 and θ) at iteration t with current value β_0^t, we propose to move to a new state, β_0', with $\beta_0' \sim N\left(\beta_0^t, \sigma^2_{\beta_0}\right)$ (Hastings (1970),Davison (2003)), where $\sigma^2_{\beta_0}$ is the proposal variance for β_0. This newly proposed state is accepted as the new state with probability $\alpha(\beta_0'|\beta_0^t)$ given by:

$$\alpha(\beta_0'|\beta_0^t) = \min\left(1, \frac{\mathcal{L}_{n,y|z}(\beta_0', \theta^t)p(\beta_0')q(\beta_0^t|\beta_0')}{\mathcal{L}_{n,y|z}(\beta_0^t, \theta^t)p(\beta_0^t)q(\beta_0'|\beta_0^t)}\right) . \quad (1)$$

where $\mathcal{L}_{n,y|z}(\beta_0', \theta^t)$ is the full likelihood calculated using the newly proposed value β_0' and current values θ^t; $p(\beta_0')$ is the prior distribution for β_0 evaluated at $\beta_0 = \beta_0'$ and $q(\beta_0'|\beta_0^t)$ denotes the proposal density of the newly proposed state β_0' given that the current state is β_0^t. We note that the terms $q(\beta_0^t|\beta_0')$ and $q(\beta_0'|\beta_0^t)$ cancel in the acceptance probability as we use a symmetrical proposal distribution (normal). Equation 1 is equivalent to equation 8.39 from Section 8.4.1 of the book.

In the following we describe each of the terms required to calculate the acceptance probability, i.e., the likelihood, the prior distributions and the proposal distributions.

2.1 Defining the likelihood function

In Section 8.2.2 of the book we learned that when including covariates in the detection function, the full likelihood is $\mathcal{L}_{n,z,y} = \mathcal{L}_n \times \mathcal{L}_z \times \mathcal{L}_{y|z}$. However, in comparison to Section 8.2.2, we use a plot count model $\mathcal{L}_n$ introduced in Section 8.2.4.1 of the book and extended with a random effect in Section 8.2.5.1. As it can be difficult to specify a good model for the covariates $\mathbf{z}$, we can simplify the modelling by omitting the component $\mathcal{L}_z$ from the likelihood and using the remainder, conditional on $\mathbf{z}$:

$$\mathcal{L}_{n,y|z} = \mathcal{L}_n \times \mathcal{L}_{y|z} .$$

To avoid numerical problems, we use the log-likelihood which is given by:

$$\ell_{n,y|z} = \log_e\left(\mathcal{L}_{n,y|z}\right) = \log_e\left(\mathcal{L}_n\right) + \log_e\left(\mathcal{L}_{y|z}\right) .$$

The function `covey.full.bayes.log.lik` calculates the log-likelihood given a set of values for parameters ***p*** and random effect coefficients ***raneff*** for a full model (see below) for the covey data. It has three arguments: `p`, `raneff` and `datax` (here `datax = covey.data`). This function expects certain elements in the data; hence, it is essential that the covey data is in the format described above, i.e., a data object created with the `create.data` function.

In the following we look at each component of the likelihood $\mathcal{L}_n$ and $\mathcal{L}_{y|z}$ separately and present the R code of the `covey.full.bayes.log.lik` that corresponds to the respective components. The code in `covey.full.bayes.log.lik` can be divided into three parts, i.e. assigning the parameter values p, the likelihood component for the detection function and the likelihood component for the count model, which we discuss individually below.

However, before we look at the `covey.full.bayes.log.lik` function, we take a step back and review the components of the probability density function of observed distances used for the detection model.

2.1.1 Components of the probability density function of observed distances

The probability density function of observed distances is generally composed of two other functions, the detection function $g(y)$ and the function $\pi(y)$ describing the expected distribution of animals with respect to the point (or line in the case of line transects). For $g(y)$, we focus on the hazard-rate model as in preliminary analyses (fitting detection functions to the distance data using program Distance) we found that this function provided the better fit to the covey distance data compared to the half-normal. The hazard-rate contains the shape parameter b and the scale parameter σ. If the latter is modelled as a function of covariates z the function is given by:

$$g(y_i, \mathbf{z}_i) = 1 - \exp\left[\left(-y_i/\sigma(\mathbf{z}_i)\right)^{-b}\right], \ 0 \leq y_i \leq w . \quad (2)$$

where, for our case study of point transect data, the y_i represent the radial distances from the point to the *i*th detection (for the $i = 1, 2, 3, ..., n$ detections), the $\mathbf{z_i}$ are the covariate measurements at the *i*th detection and w is the truncation distance.

For point transects, the function $\pi(y)$ describing the expected distribution of animals with respect to the point is given by:

$$\pi(y) = 2y\pi/\pi w^2 \, .$$

This function is evaluated at each y_i for $i = 1, ..., n$ to evaluate the log-likelihood of equation 3 (see below). As $\pi(y)$ is in the numerator and denominator of equation 3, we can omit the constants of $\pi(y)$ (i.e., $2\pi/\pi w^2$) from this equation as this will speed up the MH algorithm. Then, using these functions for $g(y, z)$ and $\pi(y)$, we can use the following R code to calculate $f(y_i|\mathbf{z}_i)$.

```
f.haz.function.pt<-function(dis,sigma,shape) {
  f <- dis*(1-exp(-(dis/sigma)^(-shape)))
  f
}
```

Here, the argument `dis` represents the y_i and `sigma` and `shape` represent the $\sigma(\mathbf{z}_i)$ and b from equation 2. In the case that covariates are part of the detection model, the value for argument `sigma` needs to represent the following:

$$\sigma(\mathbf{z}_i) = \exp\left(\alpha + \sum_{q=1}^{Q} \beta_q z_{iq}\right) \, ,$$

where α represents the scale intercept and the β_q the coefficients for Q covariates z_q included in the model.

In the case that the user wants to fit a half-normal detection function to point transect data instead of the hazard-rate, the following R function can be used instead of `f.haz.function.pt`:

```
f.hn.function.pt<-function(dis,sigma) {
  f <- dis*exp(-dis^2/(2*sigma^2))
  f
  }
```

In the case that the user wishes to analyse data that originated from line transect surveys, the following R code for detection models with the hazard-rate and half-normal detection function should be used instead of functions `f.haz.function.pt` and `f.hn.function.pt` for evaluating the likelihood from equation 3.

```
f.haz.function.ln<-function(dis,sigma,shape) {
  f <- 1-exp(-(dis/sigma)^(-shape))
  f
}
f.hn.function.ln<-function(dis,sigma) {
  f <- exp(-dis^2/(2*sigma^2))
  f
  }
```

2.1.2 Assigning the parameter values p for `covey.full.bayes.log.lik`

For illustrating the MH algorithm, we use the full set of covariates available for the detection and count models which includes the factor covariates *Type* and *State* for both models and, in addition for the count model, the continuous covariate *JDctr* which represents Julian date centred around its mean.

```
covey.full.bayes.log.lik<-function(p, raneff, datax){
# Part 1 of covey.full.bayes.log.lik(): setting up parameter values p for covariates
# the detection model
```

```
scale.int<-p[1]      # the scale intercept
shape<-exp(p[2])     # the shape parameter on the log-scale
sig.t<-c(0,p[3])     # coefficient for Type level "TREAT" (level "CONTROL is absorbed in
                     # the intercept)
sig.st<-c(0,p[4:6]) # state coefficients for levels "MS","NC","TN" (level "MO" is
                     # absorbed in the intercept)
# the count model
int<-p[7]            # the intercept
typ<-c(0,p[8])       # coefficient for Type level "TREAT" (level "CONTROL" is absorbed
                     # in the intercept)
day<-p[9]            # coefficient for JDctr
st<-c(0,p[10:12])    # state coefficients for levels "MS","NC","TN" (level "MO" is
                     # absorbed in the intercept)
std.ran<-exp(p[13]) # the random effects standard deviation on the log-scale
```

In the case that the user wishes to set up a corresponding likelihood function for their data, it is essential to include all parameters in argument *p* and to assign the parameters correctly in the likelihood calculations shown in the following sections.

2.1.3 The likelihood component for the detection function for `covey.full.bayes.log.lik`

The covey detection data consists of exact distances which are stored in the data frame `covey.data$dis.object` in the column `distance`. The truncation distance w is stored in `covey.data$w`. As we include covariates z in the detection function, we use equation 8.11 of the book as the likelihood component for the detection function which – after log-transforming – is given by:

$$\log_e\left(\mathcal{L}_{y|z}\right) = \sum_{i=1}^{n} \log_e f(y_i|\mathbf{z}_i) = \sum_{i=1}^{n} \log_e \left(\frac{g(y_i, \mathbf{z}_i)\pi(y_i)}{\int_0^w g(y, \mathbf{z}_i)\pi(y)\, dy} \right) . \tag{3}$$

Here, $f_{y|z}(y_i|\mathbf{z}_i)$ is the probability density function of observed distances y_i conditional on the covariates $\mathbf{z}_i$ (and on n) (see chapter 6 for details on fitting detection functions); w is the truncation distance. For our case study of point transect data, the y_i represent the radial distances from the point to the i$^{\text{th}}$ detection (for the $i = 1, 2, 3, ..., n$ detections). The $\mathbf{z_i}$ are the covariate measurements at the i$^{\text{th}}$ detection. We note, however, that when using the effective area to adjust observed counts for imperfect detection, individual level covariates cannot be used. The resolution of the covariates needs to be at the level of the visit to the point or higher (see section on count model likelihood component below).

The following R code is the part of the function `covey.full.bayes.log.lik` which calculates the log-likelihood for the detection model $\log_e\left(\mathcal{L}_{y|z}\right)$. Note that the truncation distance `coveydata$w` was set to 500 when formatting the data using the function `create.data`.

```
# Part 2 of covey.full.bayes.log.lik(): the likelihood component pertaining to the
# detection model
# calculating the f(y) for each observed distances
le<-nrow(datax$dis.object)
fe<-numeric(le)
alltype<-sort(unique(datax$dis.object$Type))
dis.type<-match(datax$dis.object$Type,alltype)
allstate<-sort(unique(datax$dis.object$State))
dis.state<-match(datax$dis.object$State,allstate)
# the sigma(z) for each detection
allscale<-exp(scale.int+sig.t[dis.type]+sig.st[dis.state])
# calculating the f(y) for each observation
# note that the truncation distance is stored in datax$w
for (e in 1:le){
fe[e]<-f.haz.function.pt(datax$dis.object$distance[e],allscale[e],shape)/
       integrate(f.haz.function.pt,0,datax$w,allscale[e],shape)$value
}
```

```
# the sum of the log(f(y))
log.e<-sum(log(fe))
```

Both covariates for the detection model were factor covariates. Hence we use indicators ***dis.type*** and ***dis.state*** for assigning the correct coefficients to the observations. Both indicators are vectors of the same length as `covey.data$dis.object$distance`.

In the case that the user wishes to analyse binned distance data, we use the multinomial likelihood given by equation 6.26 in the book. However, in practice, we use a reduced likelihood given by:

$$\mathcal{L}_{yG} = \prod_{j=1}^{u} f_j^{m_j} \ ,$$

where the $j = 1, 2, ..., u$ refer to the distance bins, f_j refers to the probability that a detection was in the jth bin and m_j is the number of detections in the jth bin. The f_j are obtained using

$$f_j = \frac{\int_{c_{j-1}}^{c_j} g(y,z)\pi(y)dy}{\int_0^w g(y,z)\pi(y)dy} \ ,$$

where the c_j are the cutpoints of the distance bins. The detection model may also include covariates in the same manner as described above, i.e. in a model for the scale parameter of the hazard-rate or half-normal detection function. When using binned distance data, the following R code replaces part 2 of the `covey.full.bayes.log.lik`

```
# For binned distance data: this replaces part 2 of covey.full.bayes.log.lik() above
# This part of the function is not loaded by the code in first chunk
# Part 2 of covey.full.bayes.log.lik(): the likelihood component pertaining to
# the detection model
# calculating the f_j for each detection
le<-nrow(datax$dis.object)
fj<-numeric(le)
alltype<-sort(unique(datax$dis.object$Type))
dis.type<-match(datax$dis.object$Type,alltype)
allstate<-sort(unique(datax$dis.object$State))
dis.state<-match(datax$dis.object$State,allstate)
# the sigma(z) for each detection
allscale<-exp(scale.int+sig.t[dis.type]+sig.st[dis.state])
# calculating the f_j for each observation (note that the truncation distance is
# stored in datax$w and the cutpoints of the interval are in the columns distbegin
# log.lik(): the likelihoodand distend of the data frame datax$dis.object)
for (e in 1:le){
fj[e]<-integrate(f.haz.function.pt,datax$dis.obect$distbegin[e],
                 datax$dis.object$distend[e],allscale[e],shape)$value/
                 integrate(f.haz.function.pt,0,datax$w,allscale[e],shape)$value
}
# the sum of the log(fj)
log.e<-sum(log(fj))
```

For binned distance data, the calculations below for obtaining the effective area remain unchanged.

2.1.4 The likelihood component for the count model for `covey.full.bayes.log.lik`

For the count model we use a Poisson likelihood where the expected value λ is modelled as a function of covariates x_q. Due to the repeated counts at the sites, we use the methods described in Section 8.2.5.1 of the book and include a random effect b_l for each site in the count model for which we assume normality with $b_l \sim N(0, \sigma_l^2)$. In the following subscript l refers to the different sites and subscript t to the repeated surveys. As each site also consisted of two points we also include a subscript for point k. The expected value is then given by:

$$\lambda_{lkt} = \exp\left(\sum_{q=1}^{Q} x_{qklt}\beta_q + b_l + \log_e(\nu_{lkt})\right) \quad (4)$$

Note that in comparison to the equations given in Section 8.2.5.1 of the book, we replaced the product of the surveyed area and the average detection probability $a_{lkt}P_{lkt}$ with the equivalent quantity, the effective area ν_{lkt}.

Our case study included multiple points per site with repeat visits to each site. Hence, we combine the likelihood functions from equations 8.31 and 8.33 of the book which include a random effect due to repeated visits to the same sampler and due to multiple samplers at the same site, respectively. Including a random effect in the count model entails that, in addition to the Poisson likelihoods given for the observed counts, we include normal densities for the random effect coefficients in the likelihood (Oedekoven et al. 2014). However, as described in Section 8.4.1 of the book, when fitting random effect models using the MH algorithm, the random effect is not integrated out analytically. Instead, we use a data augmentation scheme where the individual random effect coefficients are included as parameters (or auxiliary variables) in the model and updated at each iteration of the MCMC algorithm. As a consequence, when calculating the likelihood for a given set of values for the parameters and the random effect coefficients, we can omit the integral for the random effect from the likelihood which is now given by:

$$\mathcal{L}_n = \prod_{l=1}^{L}\left\{\prod_{k=1}^{K}\prod_{t=1}^{T_k}\frac{\lambda_{lkt}^{n_{lkt}}\exp[-\lambda_{lkt}]}{n_{lkt}!}\right\} \times \frac{1}{\sqrt{2\pi\sigma_l^2}}\exp\left[-\frac{b_l^2}{2\sigma_l^2}\right],$$

where L is the total number of sites (183 for our case study), K is the total number of points per site (2 for each site for our case study) and T_k is the number of repeat visits to the kth point (ranging between 1 and 4 for our case study). Without the integral, we are able to perform a one-to-one log-transformation. Hence, after the log-transformation, our likelihood component $\log_e(\mathcal{L}_n)$ for the count model is defined as:

$$\log_e(\mathcal{L}_n) = \sum_{l=1}^{L}\sum_{k=1}^{K}\sum_{t=1}^{T_k}\log_e\left(\frac{\lambda_{lkt}^{n_{lkt}}\exp[-\lambda_{lkt}]}{n_{lkt}!}\right) + \sum_{l=1}^{L}\log_e\left(\frac{1}{\sqrt{2\pi\sigma_l^2}}\exp\left[-\frac{b_l^2}{2\sigma_l^2}\right]\right).$$

For the count model we use the count data which are in a data frame stored in `covey.data$glmm.data`. Here, each record represents a single visit to a point and total numbers of detections within the truncation distance w are tallied for each visit in the column `covey.data$glmm.data$detections`. The part of the `covey.full.bayes.log.lik` function that calculates the count model likelihood component is given in the following R code:

```
# Part 3 of covey.full.bayes.log.lik():
# the likelihood component pertaining to the count model
# setting up indices for the factor covariates and random effect coefficients
Type0<-match(datax$glmm.data$Type,sort(unique(datax$glmm.data$Type)))
State0<-match(datax$glmm.data$State,sort(unique(datax$glmm.data$State)))
gr.id<-sort(unique(datax$glmm.data$gr.id))
Ran0<-match(datax$glmm.data$gr.id,gr.id)
# calculate the effective area for each visit to a point
glmm.sig<-exp(scale.int+sig.t[Type0]+sig.st[State0])
n.ptvis<-nrow(covey.data$glmm.data)
l.efa<-array(NA,n.ptvis)
for (j in 1:n.ptvis){
  l.efa[j]<-log(integrate(f.haz.function.pt,0,datax$w,glmm.sig[j],shape)$value*pi*2)
}
# calculate the log of the Poisson likelihood for each count
lambda<-exp(int + typ[Type0] + (day*datax$glmm.data$JDctr) + st[State0]
            + raneff[Ran0] + l.efa)
dpois.y<-dpois(datax$glmm.data$detections,lambda)
```

```
logdpois.y<-sum(log(dpois.y))
# calculate the log of the normal density for each random effect coefficient
log.dnorm.raneff<-sum(log(dnorm(raneff,0,std.ran)))
# adding up the likelihood components
log.lik<-logdpois.y + log.dnorm.raneff + log.e
# the function return
return(log.lik)
}
```

In the case that the user wishes to analyse data from line transects instead of point transects, the calculation of the effective area needs to be adjusted accordingly (see Section 6.2.2 of the book). Then, the effective strip half width μ_{lkt} is given by $\int_0^w g(y, \mathbf{z})dy$ and the effective area by $2 * \mu_{lkt} * ll_{lkt}$, where ll_{lkt} is the length of the respective line (stored in `datax$glmm.data$line.length`). If the interest lies in modelling numbers of individuals rather than detections, the user can replace the `datax$glmm.data$detections` in the code above with `datax$glmm.data$individuals`.

We note that in contrast with the detection model above, we include a continuous covariate, ***JDctr***, in the count model. Here, the observed values stored in `datax$glmm.data$JDctr` are multiplied by the coefficient for calculating the $lambda$ in the R code above: `day*datax$glmm.data$JDctr`.

2.2 Defining the prior probabilities for the parameters

In addition to defining the likelihood we need to set up the prior probabilities for our model parameters. If prior knowledge on the parameters exists, it may be incorporated via the prior distributions. If no prior knowledge on the parameters exists, uninformative priors, such as uniform, may be placed on all parameters. As we have no prior information on any of the parameters, we use uniform priors. The important part to consider here is that these need to be wide enough for the chain to move freely within the parameter space without hitting any of the boundaries. This may be investigated with pilot tuning (see below).

On the other hand, when using RJMCMC methods, care needs to be taken to not make the boundaries too wide as that might prevent the chain from moving freely across the model space (see RJMCMC section below for more details).

For the covey case study we have a total of 13 parameters in the combined detection and count model. (See Section Assigning the parameter values *p* above). We use the following R code to set up the lower and upper boundaries for the uniform distributions.

```
# lower and upper limits are given on the un-transformed scale for all parameters
# including the scale intercept, shape and RE sd
lolim<-c(50,1,rep(-5,4),-25,rep(-5,5),0.0001)
uplim<-c(1000,10,rep(5,4),3,rep(5,5),3)
```

We use the following R function to calculate the log of the uniform prior probability for each of the parameters. The argument `coef` is the input for the coefficient values and the arguments `lo` and `up` allow the input of parameter-specific lower and upper limits for the uniform distribution. In contrast to using `log(dunif(...))`, the `l.prior` function returns a very small value for the log of the prior probability if the value is outside the boundaries of the distribution. This prevents the acceptance of a newly proposed parameter value if it is outside the boundaries. In contrast, using `log(dunif(...))` returns `-Inf` if the parameter value is outside the boundaries which may cause the algorithm to crash.

```
l.prior<-function(coef,lo,up){
  l.u.int<-log(dunif(coef,lo,up))
  if(any(abs(l.u.int)==Inf))l.u.int<- -100000
  sum(l.u.int)
}
```

If the user wishes to use other prior distributions, the function `l.prior` can be adjusted accordingly.

2.3 Proposal distributions for updating parameters in the MH algorithm

In this section, we set up the proposal distributions for updating the parameters during the MH algorithm. We use normal proposal distributions where the mean is the current value of the parameter and the standard deviation is defined for each parameter. The proposal distributions have two purposes. They are used for updating the current parameter values: if, for example, at iteration t we wish to update parameter β_0 with current value β_0^t, we draw a new random sample $\beta_0' \sim N\left(\beta_0^t, \sigma^2_{\beta_0}\right)$ where σ_{β_0} is the proposal standard deviation for parameter β_0.

The proposal distributions are also used to calculate the proposal densities for equation 1 (see above). However, as normal proposal distributions are symmetric, the proposal density $q(\beta_0^t|\beta_0')$ equals $q(\beta_0'|\beta_0^t)$ and, hence, these cancel when calculating the acceptance probability for the proposal to update parameter β_0^t in equation 1 above. We use pilot-tuning to define the parameter-specific proposal standard deviations where we aim to obtain appropriate acceptance rates, i.e., allowing the chain to move freely in the parameter space Gelman et al. (1996). Increasing the variance on average results in decreasing acceptance probabilities and *vice versa*. See section on pilot-tuning below.

```
# proposal distributions
# Standard deviations for normal proposal distributions for all parameters:
# detection function (1:6) and count model (7:13)
# including random effects standard deviation
q.sd<-c(3.5,0.1,0.05,rep(0.03,3),0.02,0.03,0.005,rep(0.02,3),0.04)
```

2.4 Initial values for the MH updating algorithm

In this step, we set up the initial values for each of the 13 model parameters as well as the random effect coefficients. The initial parameter values are stored in the array `curparam`. This array (in combination with the corresponding array `newparam`) is used to update the current parameter values throughout the chain. (See the sections on implementing the MH and RJ algorithms below). The updated parameter values are added to a data frame at the end of each iteration of the chain (see below).

```
## initial values for the detection function parameters
# MCDS with covariates type and state
# scale intercept and shape (enter the model on the log-scale)
sig.0<-log(130) # the scale intercept
sha.0<-log(2.5) # shape parameter
# type coefficient
sigtr.0<-0.2    # level treat
# state coefficients
sigms.0<- 0.61  # MS
signc.0<- 0.66  # NC
sigtn.0<- 0.47  # TN
## initial values for count model parameters
int.0<- -13     # intercept
# type coefficient
typ.0<- 0.4     # level treat
# Julian Date coefficient (covariate JDctr)
day.0<- -0.01   # continuous covariate
# state coefficients
stms.0<- 0.014  # state MS
stnc.0<- -0.38  # state NC
sttn.0<- -1.36  # state TN
# random effects standard deviation and coefficients
std.ran.0<-log(1)
j=183 # number of sites in covey data (j = length(covey.data$Tj))
set.seed(1234)
raneff<-rnorm(j,0,exp(std.ran.0))
# combining the initial values in a single array
# for input into the covey.full.bayes.log.lik function
```

```
curparam<-c(sig.0,sha.0,sigtr.0,sigms.0,signc.0,sigtn.0,
            int.0,typ.0,day.0,stms.0,stnc.0,sttn.0,std.ran.0)
```

2.5 Storing the parameter values for the MH updating algorithm

In this section we set up the data frames which will store the values for the parameters and the random effect coefficients for each iteration of the chain of the MH updating algorithm. These are large objects due to the length of the chain and the size of the data frame holding the values for the random effect coefficients, given the number of random effect coefficients. As the chain will likely take a long time to complete, it is advisable to save these objects periodically while the chain is running. We deal with this when we set up the MH algorithm.

```
# number of iterations for the MC chain
nt<-1000
# param.mat holds the values for all parameters throughout the chain
param.mat<-matrix(NA,nt,length(curparam))
param.mat<-data.frame(param.mat)
names(param.mat)<-c("sig0","sha","sigtr","sigms","signc","sigtn","int","typ","day",
                    "stms","stnc","sttn","std.ran.0")
param.mat[1,]<-curparam
# raneff.mat holds the values for all random effect coefficients throughout the chain
raneff.mat<-matrix(NA,nt,j)
raneff.mat<-data.frame(raneff.mat)
raneff.mat[1,]<-raneff
```

2.6 The MH updating algorithm

This algorithm involves defining a number of iterations nt and updating each parameter in p from the full likelihood during each iteration using the acceptance probability from equation 1 above. In the above section Defining the likelihood function, we learned about the different components of the full likelihood $\mathcal{L}_{n,y|z}$ and how these are implemented in the R function `covey.full.bayes.log.lik`. We use this function for updating each of the parameters in p. The function `l.prior` is used to calculate the uniform prior probabilities for each of the parameters.

Additionally, as we have a random effect in the count model, we need to update each of the random effect coefficients b_l during each iteration. However, when updating the random effect coefficients, we can take a few shortcuts to make the algorithm more time-efficient. Firstly, we do not calculate prior probabilities for updating the coefficients (for our models, we do not place priors on the random effect coefficients). Also, as before we use symmetrical (normal) proposal distributions. Hence, when updating for example the random effect coefficient for site $l = 1$, the acceptance probability from equation 1 reduces to:

$$\alpha(b'_1|b^t_1) = \min\left(1, \frac{\mathcal{L}_{n,y|z}(b'_1, p^t, b^t_{-1})}{\mathcal{L}_{n,y|z}(b^t_1, p^t, b^t_{-1})}\right).$$

Furthermore, we no longer need to calculate the full likelihood for updating each random effect coefficient. We now may limit the calculations to those parts that make a difference when subtracting the log-likelihood with the current value of the random effect coefficient $log_e(\mathcal{L}_{n,y|z}(b^t_1, p^t, b^t_{-1}))$ from the log-likelihood with the updated value of the random effect coefficient $log_e(\mathcal{L}_{n,y|z}(b'_1, p^t, b^t_{-1}))$. Here, we no longer need the likelihood contribution for the detection model (the component ***log.e*** from the overall log-likelihood `log.lik` in `covey.full.bayes.log.lik` above) as this component will remain unaffected when updating any of the b_l. For updating a single random effect coefficient, we need to include the Poisson likelihoods for each count at the respective site as well as the normal density for the coefficient. Hence, following the example for updating the random effect coefficient b_1 for site $l = 1$, the following part for the overall likelihood is included in calculating the acceptance probability: $\sum_{k=1}^{K}\sum_{t=1}^{T_k} \log_e\left(\frac{\lambda_{1kt}^{n_{1kt}}\exp[-\lambda_{1kt}]}{n_{1kt}!}\right) + \log_e\left(\frac{1}{\sqrt{2\pi\sigma_l^2}}\exp\left[-\frac{b_1^2}{2\sigma_l^2}\right]\right)$. This is achieved using the function `covey.full.bayes.log.lik.raneff.update` which updates each of the random effect coefficients and returns the updated random effect coefficients.

```
############ Metropolis Hastings updating algorithm ############
# nt is the number of iterations and is set above
# row 1 in param.mat and raneff.mat contains the initial values, so we start with i = 2
# We use object t1 to measure how long each iteration takes -- see end of i loop below.
# The t1 objects and command lines with Sys.time can be omitted from the algorithm
t1<-unclass(Sys.time())
for (i in 2:nt){
  print(i)
  # updating the 13 model parameters
  newparam<-curparam
  for (b in c(1:13)){
    num<-NA
    den<-NA
    # proposing to update the bth parameter
    u<-rnorm(1,0,q.sd[b])
    # the scale intercept, shape and RE sd are on the log-scale in curparam and
    # newparam while the boundaries for the priors are not on the log-scale
    if(!is.na(match(b,c(1,2,13)))){
      newparam[b]<-log(exp(curparam[b])+u)
      new.l.prior<-l.prior(exp(newparam[b]),lolim[b],uplim[b])
      cur.l.prior<-l.prior(exp(curparam[b]),lolim[b],uplim[b])
    }
    else{
      newparam[b]<-curparam[b]+u
      new.l.prior<-l.prior(newparam[b],lolim[b],uplim[b])
      cur.l.prior<-l.prior(curparam[b],lolim[b],uplim[b])
    }
    num<-covey.full.bayes.log.lik(newparam,raneff,covey.data) + new.l.prior
    den<-covey.full.bayes.log.lik(curparam,raneff,covey.data) + cur.l.prior
    A<-min(1,exp(num-den))
    V<-runif(1)
    ifelse(V<=A,curparam[b]<-newparam[b],newparam[b]<-curparam[b])
  }
  # storing the updated parameter values
  param.mat[i,]<-curparam
  # updating the random effect coefficients
  raneff<-covey.full.bayes.log.lik.raneff.update(curparam, raneff, covey.data)
  # storing the updated random effect coefficients
  raneff.mat[i,]<-raneff
  # saving the parameter matrices every 5000 iterations
  if(!is.na(match(i,seq(0,nt,5000))==T)){
    save(param.mat,file='param.mat.RData')
    save(raneff.mat,file='raneff.mat.RData')
  }
  # The next 2 lines allow you to measure the time each iteration takes to complete.
  # They can be omitted from the algorithm
  print(paste("this iteration took ",round(unclass(Sys.time())-t1,2)," seconds",sep=""))
  t1<-unclass(Sys.time())
  # this next if statement allows us to periodically check the trace plots
  # without stopping the algorithm (which is handy for pilot tuning)
  if(!is.na(match(i,seq(0,nt,100))==T)){
    par(mfrow=c(2,2))
    for (k in 1:13){
      plot(param.mat[1:i,k],xlab=k,t='l',main=i)
    }
  }
}# end of i loop
```

2.7 Pilot tuning the MH updating algorithm

For an MH updating algorithm, pilot tuning involves ensuring that the parameter space is explored freely for each parameter. This can be done using trace plots which may be produced at any iteration i with the following R code.

```
# The following code will only work if the MH algorithm has run for several iterations.
# We recommend running at least 100 iterations.
# Producing trace plots for all 13 parameters for iterations 1:i
par(mfrow=c(3,2))
for (b in 1:13){
plot(param.mat[1:i,b],t='l',xlab="Iteration",main=paste("Parameter ",b,sep=""),ylab="")}
```

For our case study, we use normal proposal distributions which use the current value of the respective parameter as the mean and have parameter-specific proposal variances (defined in the section on proposal distributions above). Pilot tuning may involve adjusting the standard deviations of the proposal distributions (defined in the array `q.sd` above). We show two sets of trace plots for the same selection of parameters in *p* - after pilot tuning (Figure 1) presenting the desired pattern of quick up-and-down movements and before pilot tuning (Figure 2) presenting the undesired pattern of a 'skyscraper landscape'. For Figure 1, the proposal standard deviations were set to:

```
# The user does not need to run this part for the exercise
# proposal standard deviations after pilot tuning
q.sd<-c(3.5,0.1,0.05,rep(0.03,3),0.02,0.03,0.005,rep(0.02,3),0.04)
```

For Figure 2, the proposal standard deviations were set to:

```
# The user does not need to run this part for the exercise
# proposal standard deviations before pilot tuning
q.sd<-c(12,rep(0.3,7),0.03,rep(0.3,4))
```

Figure 1. Trace plots for iterations 1:1000 for parameters 7:12 after pilot tuning. Parameters 7:12 correspond to the count model parameters (except the random effects standard deviation).

Figure 2. Trace plots for iterations 1:554 for parameters 7:12 before pilot tuning with larger proposal standard deviations compared to Figure 1. Parameters 7:12 correspond to the count model parameters (except the random effects standard deviation).

In addition, we need to ensure that the potential values that each parameter in `p` can take are not limited by the boundaries of the uniform prior distributions. This can also be checked using the trace plots. Figure 3 shows the trace plot of parameter 1 when the upper boundary of its uniform prior was set to 250 before pilot tuning ($log_e(250) \approx 5.52$). It is clear that the upper boundary was too low for this parameter and that the chain was not moving freely. We note that artificially constraining one parameter may affect other parameters as well.

Figure 3. Trace plot for parameter 1, the intercept of the scale parameter for the detection model before pilot tuning when the upper boundary of the uniform prior probability distribution for this parameter was set too low.

2.7.1 Inference from an MH updating algorithm

2.7.1.1 Inference on parameters from an MH updating algorithm We draw inference on parameters by obtaining summary statistics of the marginal posterior distributions. For this purpose, we use the parameter values stored in `param.ma` after completing the MH algorithm above for all $nt = 100000$ iterations. For the purpose of the exercise, rather than waiting for the 10000 iterations to complete, the results can also be uploaded into the workspace using:

```
load("param.mat.rdata")
```

Using *param.mat*, we calculate the mean, standard deviation and central 95% credible intervals. We use the first 9999 iterations for the burn-in phase and omit these. These summary statistics may be obtained using the following R code:

```
# means
kable(t(round(apply(param.mat[10000:nt,1:12],2,mean),4)))
```

sig0	sha	sigtr	sigms	signc	sigtn	int	typ	day	stms	stnc	sttn
5.8481	1.2752	-0.2457	-0.0243	0.0119	-0.1237	-13.1646	0.6761	-0.0228	-0.3818	-1.5033	-1.1605

```
# standard deviations
kable(t(round(apply(param.mat[10000:nt,1:12],2,sd),4)))
```

sig0	sha	sigtr	sigms	signc	sigtn	int	typ	day	stms	stnc	sttn
0.0548	0.1003	0.0628	0.0613	0.0811	0.0904	0.1327	0.1026	0.0038	0.2002	0.1973	0.2512

```
# 95% credible intervals
kable(round(apply(param.mat[10000:nt,1:6],2,quantile,probs=c(0.025,0.975)),4))
```

	sig0	sha	sigtr	sigms	signc	sigtn
2.5%	5.7435	1.0765	-0.3805	-0.1416	-0.1337	-0.2878
97.5%	5.9532	1.4713	-0.1241	0.0986	0.1784	0.0753

```
kable(round(apply(param.mat[10000:nt,7:12],2,quantile,probs=c(0.025,0.975)),4))
```

	int	typ	day	stms	stnc	sttn
2.5%	-13.4671	0.4737	-0.0300	-0.7444	-1.8819	-1.7523
97.5%	-12.9332	0.8727	-0.0154	-0.0042	-1.1377	-0.6604

```
## for the random effects standard deviation we convert the values back from the log-scale
# mean
kable(t(round(mean(exp(param.mat[10000:nt,13])),4)))
```

0.733

```
# standard deviation
kable(t(round(sd(exp(param.mat[10000:nt,13])),4)))
```

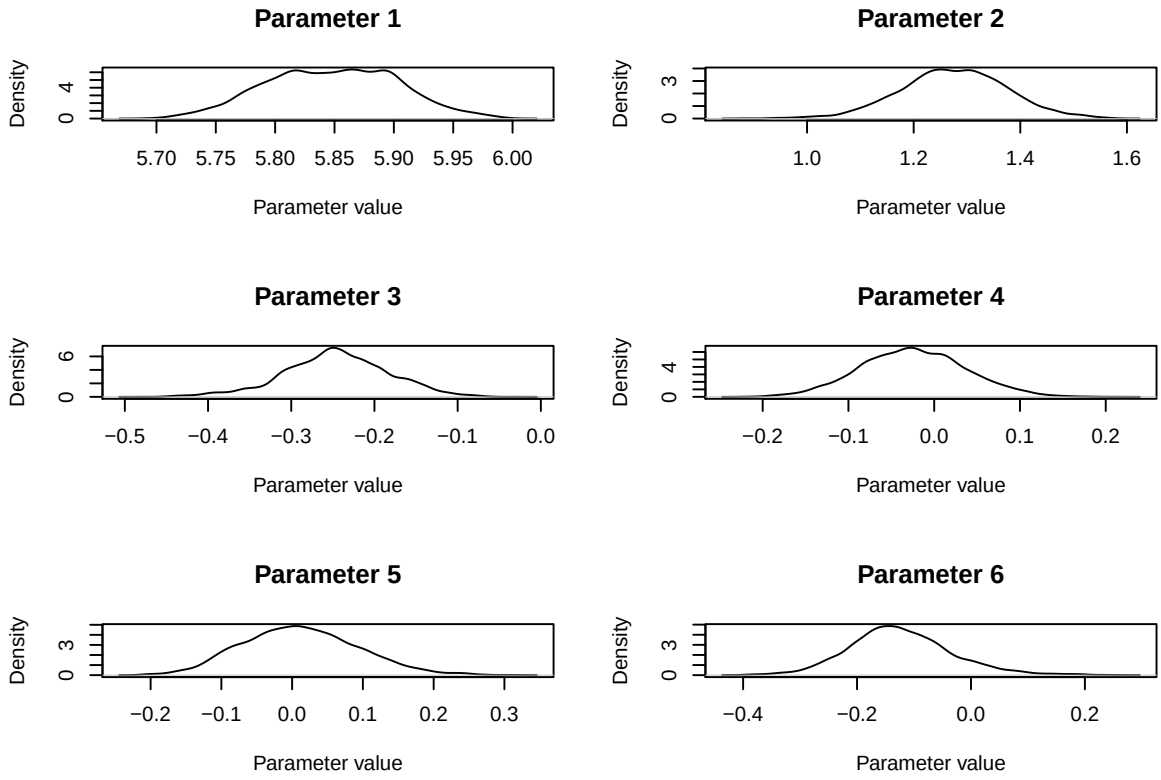
0.0656

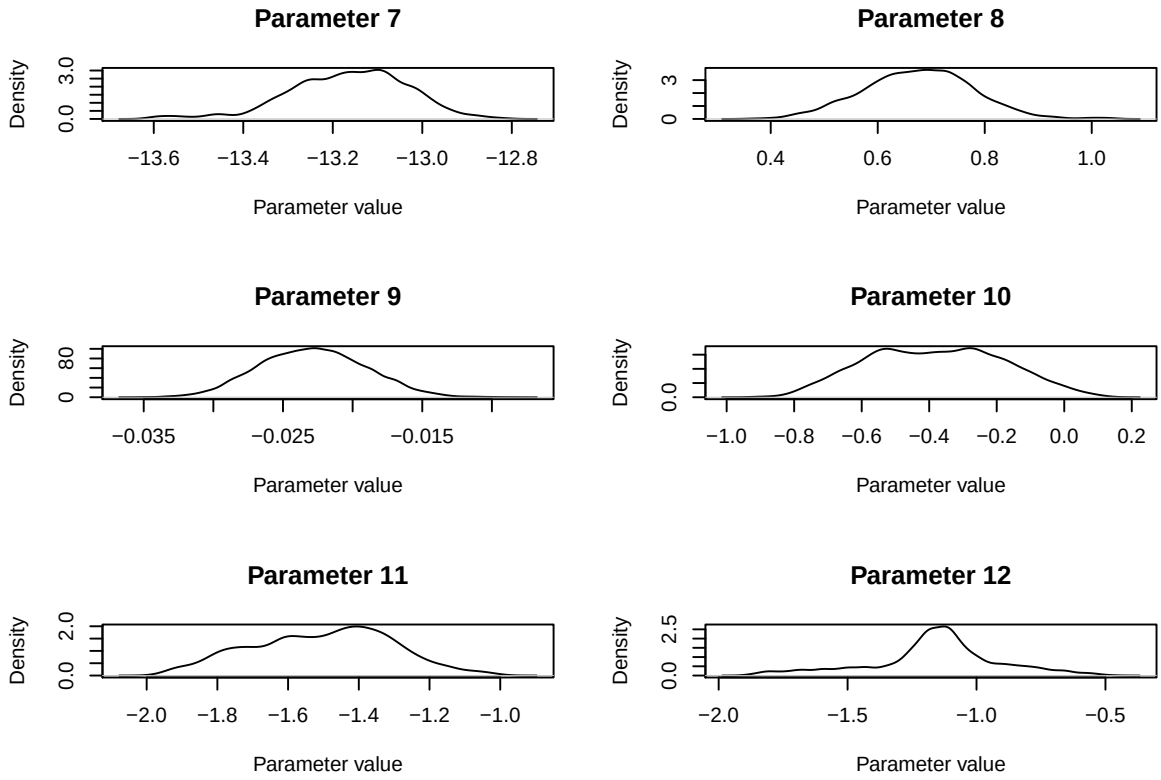
```
# 95% credible intervals
kable(t(round(quantile(exp(param.mat[10000:nt,13]),probs=c(0.025,0.975)),4)))
```

2.5%	97.5%
0.6062	0.8635

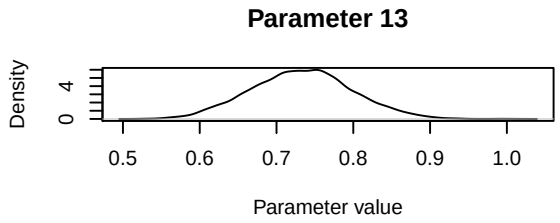
We can plot the distribution for each of the parameters using the following R code:

```
par(mfrow=c(3,2))
for (b in 1:12){
  plot(density(param.mat[10000:nt,b]),t='l',
       main=paste("Parameter ",b,sep=""),
       ylab="Density",xlab="Parameter value")
}
```






```
# for the random effects standard deviation we convert the values back from the log-scale
# before plotting
plot(density(exp(param.mat[10000:nt,13])),t='l',
     main=paste("Parameter ",13,sep=""),
     ylab="Density",xlab="Parameter value")
```



The parameter of interest for this study was the coefficient for level "TREAT" for the ***Type*** covariate in the count model. The mean estimate for this parameter was 0.68 (SD = 0.103, 95% central credible intervals = {0.47, 0.87}). Remembering that this parameter entered the model for the expected value λ_{lkt} on the log-scale (equation 4 from above: $\lambda_{lkt} = \exp\left(\sum_{q=1}^{Q} x_{qklt}\beta_q + b_l + \log_e(\nu_{lkt})\right)$) we conclude that covey densities were 97.66 % higher on treated fields compared to control fields ($100 \times [exp(0.67) - 1]$).

2.7.2 Inference on plot density from an MH updating algorithm

If we wish to draw inference on plot density, we can treat plot density as a parameter and obtain similar summary statistics as for the parameters p. For this purpose, we use the parameter values from the count model stored in `param.mat` and calculate the expected density for each iteration (excluding the burn-in phase). We note that while λ_{lkt} (equation 4) models the expected counts, $\exp\left(\sum_{q=1}^{Q} x_{qklt}\beta_q + b_l\right)$ from the λ_{lkt} model (i.e., without the effective area $\log_e(\nu_{lkt})$) models the densities (i.e., number of coveys per m^2).

We can obtain posterior distributions of plot density for any given combination of covariates (e.g., ***control*** vs ***treatment*** plots). We may wish, for example, to draw inference on a baseline plot density where each covariate value is set to their baseline level (level for ***Type*** = "CONTROL", ***JDctr*** = 0, level for ***State*** = "MO"). We further need to add a contribution from the random effect: $0.5 \times \sigma_l^2$. Hence, the expected baseline density D_{bl} can be expressed as: $D_{bl} = exp(\beta_1 + (0.5 \times \sigma_l^2)))$, where β_1 is the intercept for the count model and σ_l is the random effects standard deviation. We can calculate the D_{bl} for each iteration either during the MH-update by including the appropriate code or calculate these after the MH algorithm has completed. Regardless, we need to consider that the unit for distance measurements were metres (`covey.data$dis.unit` = m). As the effective area also enters the model for λ in m^2, the units for the density estimates from the count model are m^2. We can convert the density estimates from D_{bl}/m^2 to D_{bl}/km^2 by including the appropriate code:

```
Dbl<-array(NA,nt)
# calculate the baseline density
# omit the burn-in
for (i in 10000:nt){
  Dbl[i]<-exp(param.mat[i,7]+(0.5*exp(param.mat[i,13])^2))
}
Dbl<-Dbl*1000*1000
```

Summary statistics for the baseline density can now be obtained using:

```
# the posterior mean, standard deviation and central credible intervals for
# baseline density after omitting the burn-in
# (rounded to 2 decimal points)
round(mean(Dbl[10000:nt]),2)
```

```
[1] 2.54
```

```
round(sd(Dbl[10000:nt]),2)
```

```
[1] 0.33
```

```
round(quantile(Dbl[10000:nt],prob=c(0.025,0.975)),2)
```

```
 2.5% 97.5%
 1.87  3.18
```

We conclude that the baseline density of northern bobwhite coveys was 2.54 coveys per km^2 (SD = 0.33, 95% central credible intervals = {1.87, 3.183}).

3 Setting up a reversible jump MCMC algorithm

In this section, we use the methods described in Section 8.4.2 of the book. We build upon the previous example and include model selection as part of the inference. Now, instead of conditioning on the full model being the correct model as in the MH algorithm above, we include a reversible jump (RJ) step at each iteration where we propose to move to a different model. Now each iteration consists of a proposal to update the model (the RJ step) and a proposal to update the parameters of the current model (the MH step). For the MH step we use an algorithm similar to the above except that only parameters contained in the current model are updated. The RJ step is described in more detail in the following section.

However, setting up an RJMCMC updating algorithm involves a few additional steps compared to the MH algorithm above. The steps for the RJMCMC algorithm include:

1. Defining the likelihood function
2. Defining the prior probabilities for the parameters
3. Proposal distributions for the MH step and the RJ step
4. Initial model and parameter values for the RJMCMC algorithm
5. Storing the parameter values for the RJMCMC algorithm
6. The RJMCMC algorithm
7. Pilot tuning the RJMCMC algorithm

In this section, we provide the functions and R code necessary to implement these steps. In addition, we describe how to obtain summary statistics of the posterior model probabilities and posterior distributions of parameters and plot density. We begin with the details for the RJ step.

3.1 The RJ step

The RJ step at each iteration involves proposing to add or delete each of the available covariates for the detection and count models depending on whether they are in the current model or not. If, for example, at iteration t covariate ***State*** is not contained in the detection model, we propose to add it. We use m to denote the current model with parameters p (not including ***State*** in the detection model) and m' to denote the newly proposed model (including ***State*** in the detection model). This involves drawing a random sample for each of the coefficients of the three levels of ***State***, say β_2', β_3', β_4', (***State*** has four levels; however, the first level "MO" is absorbed in the intercept). We use the respective proposal distributions for drawing

the samples (e.g., $\beta_2' \sim N(\mu_{\beta_2}, \sigma^2_{\beta_2})$ where μ_{β_2} and $\sigma^2_{\beta_2}$ are the parameter-specific proposal mean and variance). We use the identity function as the bijective function (similar to equation 8.41 of the book) where all other parameters p of the current model m are set equal to the corresponding parameters in p' and the additional parameters for model m' are:

$$\begin{aligned} u_2 &= \beta_2' \\ u_3 &= \beta_3' \\ u_4 &= \beta_4' \,; \end{aligned}$$

for this particular model move. We then calculate the acceptance probability $A(m'|m)$ for this proposed move using equation 5 (equivalent to equation 8.42 from Section 8.4.2 of the book). However, for our proposed move, this equation can be simplified. As we use the identity function as the bijective function, the Jacobian $|J|$ equals 1. Furthermore, as we consider all model probabilities equally likely, $P(m|m')$ and $P(m'|m)$ as well as $p(m)$ and $p(m')$ cancel in equation 5. This leaves calculating the likelihoods (for the newly proposed model $\mathcal{L}_{n,y|z}(p', m')$ as well as the current model $\mathcal{L}_{n,y|z}(p, m)$), the proposal densities for the new parameters ($q\left(u_2\right) q\left(u_3\right) q\left(u_4\right)$) and the prior probabilities for the new parameters ($p(\beta_2')p(\beta_3')p(\beta_4')$):

$$A(m'|m) = \frac{\mathcal{L}_{n,y|z}(p', m')p(\beta_2')p(\beta_3')p(\beta_4')}{\mathcal{L}_{n,y|z}(p, m)q\left(u_2\right) q\left(u_3\right) q\left(u_4\right)} \,. \tag{5}$$

We show the R code for calculating the acceptance probabilities in the section The RJMCMC algorithm below.

3.2 Defining the likelihood function for the RJMCMC algorithm

For the RJMCMC algorithm, we use the same likelihood function `covey.full.bayes.log.lik` (including the `f.haz.function.pt`) as for the MH algorithm described above (see Section Defining the likelihood function above) despite the fact that now the current model may contain fewer parameters than the full model. Therefore, it is important that the values for the respective model coefficients listed in the array ***p*** are in the same position as they would be in the full model. If, for example, the detection model contains the covariate ***State*** but not ***Type***, the third position in ***curdetparam*** must be 0 while positions 1,2,4,5,6 contain the current values for the scale intercept, the shape parameter and the state coefficients. In comparison to the MH algorithm, we store the current parameter values of the detection and count models in separate arrays, `curdetparam` and `curcountparam` – mainly as this makes it easier to keep track of which detection model and which count model we are currently in (see below for details). However, as the likelihood function `covey.full.bayes.log.lik` expects one array of parameter values containing the detection and count model parameters, we combine these using, e.g., `c(curdetparam,curcountparam)`.

> If the user wishes to adjust the likelihood function for their RJMCMC algorithm, we recommend that they follow this scheme of defining the function for the full model and setting parameter values for covariates not contained in the current model to zero – provided that all possible models are nested within the full model. In the case that moves between non-nested models should be included in the algorithm, e.g. between the half-normal and the hazard-rate detection model, the likelihood function may require an additional argument that allows switching between detection models.
>
> In the case that the user wants to analyse binned distance data, the same modifications apply as described above. We note, however, that moves between models where distances are analysed as exact and models where distances are analysed as binned are not allowed in an RJMCMC algorithm. This is because the data change between the two approaches.

3.3 Prior probability distributions

We consider all models equally likely. Hence, the probability $P(m'|m)$ of proposing to move to model m' given that the chain is in model m and the probability $P(m|m')$ for the reverse step of proposing to move to model m given that the chain is in model m' are equal and cancel in equation 8.42 of the book.

As for the MH algorithm above, we place uniform priors on all model parameters and use the same function `l.prior` to calculate the log of the prior probabilities. The lower and upper boundaries for the uniform prior distributions for each of the parameters are the same as for the MH algorithm; however, we define the boundaries separately for the detection function and count models to be consistent with the format of `curdetparam` and `curcountparam` (i.e., arrays of length 6 and 7 for the full detection and count models, respectively).

```
# setting the lower and upper limits for the uniform prior probability distributions
detlolim<-c(50,1,rep(-5,4))
detuplim<-c(1000,10,rep(5,4))
countlolim<-c(-25,rep(-5,5),0.0001)
countuplim<-c(3,rep(5,5),3)
```

3.4 Proposal distributions for the RJ step

As for the MH algorithm, we use normal proposal distributions. However, for the RJMCMC algorithm, we need two sets of proposal distributions: one set for the RJ step and another set for the MH step. The proposal distributions for the RJ step are used to draw random samples for new model coefficients. They are also used to calculate proposal densities for calculating the acceptance probabilities using equation 5. We note that certain parameters are always in the model and do not need a proposal distribution for the RJ step. However, for consistency and simplicity (see below), we define these as well. These parameters include the scale intercept and shape parameter of the hazard-rate detection model and the intercept and random effects standard deviation of the count model.

The proposal distributions for the MH step are the same as for the MH algorithm above with zero means and parameter-specific standard deviations. However, we use the same format for storing the standard deviations as for `curdetparam` and `curcountparam`.

```
# proposal distributions for parameters in the detection and count model
# for RJ step: proposals of new parameters
detprop.mean<-c(356.60, 3.74, 0.23, -0.06, -0.03, -0.18)
countprop.mean<-c(-13.20, 0.33, -0.023, -0.43, -1.46, -1.30, 0.70)
detprop.sd<-c(1.41, 0.84, rep(0.05,4))
countprop.sd<-c(0.3, 0.1, 0.01, 0.2, 0.2, 0.2, 0.3)
# for MH step: proposals to update current parameters
detq.sd<-c(3.5,0.1,0.05,rep(0.03,3))
countq.sd<-c(0.02,0.03,0.005,rep(0.02,3),0.04)
```

3.5 Storing the parameter values and model choices for each iteration

This is more complex than for the MH algorithm, as we need not only to store the values for the parameters and random effect coefficients during each iteration, but also to keep track of which model was chosen for each iteration and which parameters in p are switched on and off. For this purpose, we set up model indicator matrices, `det.list` and `count.list`, which indicate what parameters are switched on for each possible covariate combination. These indicator matrices contain combinations of 0 and 1 in each row where the row number refers to the model number and 1 indicates that the respective parameter (indicated by column names) is switched on for the respective model. These will help us store the model choices for the detection and count models for each iteration as integer numbers in the vectors `det.model` and `count.model`, respectively.

```
# array that will keep track of the model choice (stored as an integer number
# to the model number (row number) from det.list and count.list below)
det.model<-array(NA,nt)
count.model<-array(NA,nt)
# we need to know which model contains which parameters
# row number indicates the model number
# column names indicate which parameters are switched on or off for the respective model
# 0: parameter is switched off
# 1: parameter is switched on
det.list<-matrix(0,4,6)     # 2 covariates: 4 possible models
count.list<-matrix(0,8,7)   # 3 covariates: 8 possible models
# columns: scale intercept, shape, type coef level TREAT, state coef levels MS, NC, TN
colnames(det.list)<-c("scale.int","shape","sig.tr","sig.ms","sig.nc","sig.tn")
# columns: intercept, type coef level TREAT, JDctr coef, state coef levels MS, NC, TN
colnames(count.list)<-c("int","type.tr","day","st.ms","st.nc","st.tn","sd.ran")
```

```
# all detection models contain the scale intercept and shape parameter (hazard rate)
det.list[1,]<-c(1,1,rep(0,4))              # global model with no covariates
det.list[2,]<-c(1,1,1,rep(0,3))            # mcds with type
det.list[3,]<-c(1,1,0,rep(1,3))            # mcds with state
det.list[4,]<-c(rep(1,6))                  # mcds with type and state
# all count models contain the intercept and RE sd
count.list[1,]<-c(1,rep(0,5),1)            # no covariates
count.list[2,]<-c(1,1,rep(0,4),1)          # with type
count.list[3,]<-c(1,0,1,rep(0,3),1)        # with JDctr
count.list[4,]<-c(1,0,0,rep(1,3),1)        # with state
count.list[5,]<-c(1,1,1,rep(0,3),1)        # with type and JDctr
count.list[6,]<-c(1,1,0,rep(1,3),1)        # with type and state
count.list[7,]<-c(1,0,rep(1,4),1)          # with JDctr and state
count.list[8,]<-c(rep(1,7))                # with type, JDctr and state
# data frames for storing the parameter values
detparam.mat<-matrix(NA,nt,6)
detparam.mat<-data.frame(detparam.mat)
countparam.mat<-matrix(NA,nt,7)
countparam.mat<-data.frame(countparam.mat)
# these obtain the same column names as det.list and count.list
colnames(detparam.mat)<-c("scale.int","shape","sig.tr","sig.ms","sig.nc","sig.tn")
colnames(countparam.mat)<-c("int","type.tr","day","st.ms","st.nc","st.tn","sd.ran")
# data frame for storing the random effect coefficients
raneff.mat<-matrix(NA,nt,183)       # 183 sites
raneff.mat<-data.frame(raneff.mat)
```

3.6 Setting initial models and parameter values for the RJMCMC algorithm

In contrast to the MH algorithm above where we conditioned on the full model, we will begin our chain with the most parsimonious model options: the global hazard-rate detection function without any covariates (detection model 1) and a count model containing no covariates (count model 1). Hence, we need set up starting values for the model parameters including the scale intercept and shape parameter of the hazard-rate detection model and the intercept and random effects standard deviation of the count model. Other options for starting the chain include beginning the chain with the full models (models 4 and 8 for the detection and count models) or drawing the respective model numbers by chance.

The current model choices for the detection and count models are stored in `curdetmodel` and `curcountmodel` which are updated at each iteration. The current parameter values are stored in `curdetparam` and `curcountparam` which are updated at each iteration in combination with `newdetparam` and `newcountparam`. As for the MH algorithm, we store the current values for the random effect coefficients in `raneff`.

```
## Setting the initial models
# global detection function and intercept + RE sd count model
curdetmodel<-1
curcountmodel<-1
det.model[1]<-curdetmodel
count.model[1]<-curcountmodel
# curdetlist and curcountlist will tell you which parameters are
# switched on for the current model
curdetlist<-det.list[curdetmodel,]
curcountlist<-count.list[curdetmodel,]
## Setting up initial values
# model 1 for the detection function only contains two parameters:
# scale intercept and shape (which are both on the log-scale)
sig.0<-log(130)
sha.0<-log(2.5)
# initial values for count model parameters
int.0<- -13        # intercept
```

```
std.ran.0<-log(1)  # random effects standard deviation
j=183              # number of sites in the covey data
set.seed(1234)
b0<-rnorm(j,0,exp(std.ran.0))
# curdetparam always needs to be of length 6 and curcountparam of length 7
# as the likelihood function covey.full.bayes.log.lik expects an array
# of length 13 for argument p
curdetparam<-array(0,6)
curdetparam[1:2]<-c(sig.0,sha.0)
curcountparam<-array(0,7)
curcountparam[1]<-c(int.0)
curcountparam[7]<-c(std.ran.0)
raneff<-b0
detparam.mat[1,]<-curdetparam
countparam.mat[1,]<-curcountparam
raneff.mat[1,]<-raneff
```

3.7 The RJMCMC algorithm

As for the MH algorithm, we use the function `covey.full.bayes.log.lik.raneff.update` to update the random effect coefficients. (See Section The MH updating algorithm above for more details about this function). In the following algorithm, we begin each iteration with the RJ step where we propose to update the current model. Here, we cycle through each covariate in the detection and count models and propose to add or delete it depending on whether it is in the current model or not. The potential covariates are ***Type*** and ***State*** for the detection model and ***Type***, ***JDctr*** (centred Julian date) and ***State*** for the count model. Cycling through each of these covariates completes the RJ step. Note that ***Type*** and ***State*** are factor covariates with two and four levels, respectively, and ***JDctr*** is a continuous covariate.

Following the RJ step is the MH step where we propose to update current model parameters. Here we cycle through each of the current parameters and propose to update it. At the completion of the RJ step, we store current model choices (detection and count model) in the respective data frames (`det.model` and `count.model`). At the completion of the MH step, we store the current parameter values and random effect coefficients in the respective data frames (`detparam.mat`, `countparam.mat` and `raneff.mat`). We note that in the case a parameter is not in the current model, the parameter value will be stored as 0. Using the information stored in `det.list` or `count.list` in combination with the model choice stored in `det.model` or `count.model` allows us to distinguish these 0s from true zeros for a parameter value.

```
# nt is the number of iterations and is set above
# row 1 in detparam.mat, coutparam.mat and raneff.mat contains the initial values, so we
# start with i = 2.
# We use object t1 to measure how long each iteration takes -- see end of i loop below.
# The t1 objects and command lines with Sys.time can be omitted from the algorithm
t1<-unclass(Sys.time())
for (i in 2:nt){
  print(i)
########################## RJ step ##########################
# method: propose to add or delete each covariate depending on
# whether it is in the current model
  newdetlist<-curdetlist
  newdetparam<-curdetparam
  newcountlist<-curcountlist
  newcountparam<-curcountparam
  ########################## detection function parameters
  # the scale intercept and shape parameters are always in the model
  # covariate type
  # check if it is in the current model:
  # propose to add it if it is not in the current model
  if(curdetlist[3]==0){
    newdetlist[3]<-1
```

```
  newdetparam[3]<-rnorm(1,detprop.mean[3],detprop.sd[3])
  num<-NA
  den<-NA
  # the prior probabilities
  new.l.prior<-l.prior(newdetparam[3],detlolim[3],detuplim[3])
  # likelihood
  new.lik<-covey.full.bayes.log.lik(c(newdetparam,newcountparam),raneff,covey.data)
  cur.lik<-covey.full.bayes.log.lik(c(curdetparam,curcountparam),raneff,covey.data)
  prop.dens<-log(dnorm(newdetparam[3],detprop.mean[3],detprop.sd[3]))
  # numerator and denominator in equation 5
  num<-new.lik+new.l.prior
  den<-cur.lik+prop.dens
}
# propose to delete it if it is in the current model
else{
  newdetlist[3]<-0
  newdetparam[3]<-0
  num<-NA
  den<-NA
  # the prior probabilities
  cur.l.prior<-l.prior(curdetparam[3],detlolim[3],detuplim[3])
  # likelihood
  new.lik<-covey.full.bayes.log.lik(c(newdetparam,newcountparam),raneff,covey.data)
  cur.lik<-covey.full.bayes.log.lik(c(curdetparam,curcountparam),raneff,covey.data)
  # proposal density
  prop.dens<-log(dnorm(curdetparam[3],detprop.mean[3],detprop.sd[3]))
  # numerator and denominator in equation 5
  num<-new.lik+prop.dens
  den<-cur.lik+cur.l.prior
}
A<-min(1,exp(num-den))
V<-runif(1)
if(V<=A){
  curdetparam[3]<-newdetparam[3]
  curdetlist[3]<-newdetlist[3]
}
else{
  newdetparam[3]<-curdetparam[3]
  newdetlist[3]<-curdetlist[3]
}
# covariate state
# check if it is in the current model:
# propose to add it if it is not in the current model
if(curdetlist[4]==0){
  newdetlist[4:6]<-1
  newdetparam[4:6]<-rnorm(3,detprop.mean[4:6],detprop.sd[4:6])
  num<-NA
  den<-NA
  # the prior probabilities
  new.l.prior<-sum(l.prior(newdetparam[4:6],detlolim[4:6],detuplim[4:6]))
  # likelihood
  new.lik<-covey.full.bayes.log.lik(c(newdetparam,newcountparam),raneff,covey.data)
  cur.lik<-covey.full.bayes.log.lik(c(curdetparam,curcountparam),raneff,covey.data)
  # proposal density
  prop.dens<-sum(log(dnorm(newdetparam[4:6],detprop.mean[4:6],detprop.sd[4:6])))
  # numerator and denominator in equation 5
  num<-new.lik+new.l.prior
```

```
  den<-cur.lik+prop.dens
}
# propose to delete it if it is in the current model
else{
  newdetlist[4:6]<-0
  newdetparam[4:6]<-0
  num<-NA
  den<-NA
  # the prior probabilities
  cur.l.prior<-sum(l.prior(curdetparam[4:6],detlolim[4:6],detuplim[4:6]))
  # likelihood
  new.lik<-covey.full.bayes.log.lik(c(newdetparam,newcountparam),raneff,covey.data)
  cur.lik<-covey.full.bayes.log.lik(c(curdetparam,curcountparam),raneff,covey.data)
  # proposal density
  prop.dens<-sum(log(dnorm(curdetparam[4:6],detprop.mean[4:6],detprop.sd[4:6])))
  # numerator and denominator in equation 5
  num<-new.lik+prop.dens
  den<-cur.lik+cur.l.prior
}
A<-min(1,exp(num-den))
V<-runif(1)
if(V<=A){
  curdetparam[4:6]<-newdetparam[4:6]
  curdetlist[4:6]<-newdetlist[4:6]
}
else{
  newdetparam[4:6]<-curdetparam[4:6]
  newdetlist[4:6]<-curdetlist[4:6]
}
########################################## count model parameters
# the intercept and RE sd are always in the model
# covariate type
# check if it is in the current model:
# propose to add it if it is not in the current model
if(curcountlist[2]==0){
  newcountlist[2]<-1
  newcountparam[2]<-rnorm(1,countprop.mean[2],countprop.sd[2])
  num<-NA
  den<-NA
  # the prior probabilities
  new.l.prior<-l.prior(newcountparam[2],countlolim[2],countuplim[2])
  # likelihood
  new.lik<-covey.full.bayes.log.lik(c(newdetparam,newcountparam),raneff,covey.data)
  cur.lik<-covey.full.bayes.log.lik(c(curdetparam,curcountparam),raneff,covey.data)
  # proposal density
  prop.dens<-log(dnorm(newcountparam[2],countprop.mean[2],countprop.sd[2]))
  # numerator and denominator in equation 5
  num<-new.lik+new.l.prior
  den<-cur.lik+prop.dens
}
# propose to delete it if it is in the current model
else{
  newcountlist[2]<-0
  newcountparam[2]<-0
  num<-NA
  den<-NA
  # the prior probabilities
  cur.l.prior<-l.prior(curcountparam[2],countlolim[2],countuplim[2])
```

```
  # likelihood
  new.lik<-covey.full.bayes.log.lik(c(newdetparam,newcountparam),raneff,covey.data)
  cur.lik<-covey.full.bayes.log.lik(c(curdetparam,curcountparam),raneff,covey.data)
  # proposal density
  prop.dens<-log(dnorm(curcountparam[2],countprop.mean[2],countprop.sd[2]))
  # numerator and denominator in equation 5
  num<-new.lik+prop.dens
  den<-cur.lik+cur.l.prior
}
A<-min(1,exp(num-den))
V<-runif(1)
if(V<=A){
  curcountparam[2]<-newcountparam[2]
  curcountlist[2]<-newcountlist[2]
}
else{
  newcountparam[2]<-curcountparam[2]
  newcountlist[2]<-curcountlist[2]
}
# covariate JDctr (Julian date)
# check if it is in the current model:
# propose to add it if it is not in the current model
if(curcountlist[3]==0){
  newcountlist[3]<-1
  newcountparam[3]<-rnorm(1,countprop.mean[3],countprop.sd[3])
  num<-NA
  den<-NA
  # the prior probabilities
  new.l.prior<-l.prior(newcountparam[3],countlolim[3],countuplim[3])
  # likelihood
  new.lik<-covey.full.bayes.log.lik(c(newdetparam,newcountparam),raneff,covey.data)
  cur.lik<-covey.full.bayes.log.lik(c(curdetparam,curcountparam),raneff,covey.data)
  # proposal density
  prop.dens<-log(dnorm(newcountparam[3],countprop.mean[3],countprop.sd[3]))
  # numerator and denominator in equation 5
  num<-new.lik+new.l.prior
  den<-cur.lik+prop.dens
}
# propose to delete it if it is in the current model
else{
  newcountlist[3]<-0
  newcountparam[3]<-0
  num<-NA
  den<-NA
  # the prior probabilities
  cur.l.prior<-l.prior(curcountparam[3],countlolim[3],countuplim[3])
  # likelihood
  new.lik<-covey.full.bayes.log.lik(c(newdetparam,newcountparam),raneff,covey.data)
  cur.lik<-covey.full.bayes.log.lik(c(curdetparam,curcountparam),raneff,covey.data)
  # proposal density
  prop.dens<-log(dnorm(curcountparam[3],countprop.mean[3],countprop.sd[3]))
  # numerator and denominator in equation 5
  num<-new.lik+prop.dens
  den<-cur.lik+cur.l.prior
}
A<-min(1,exp(num-den))
V<-runif(1)
if(V<=A){
```

```
  curcountparam[3]<-newcountparam[3]
  curcountlist[3]<-newcountlist[3]
}
else{
  newcountparam[3]<-curcountparam[3]
  newcountlist[3]<-curcountlist[3]
}
# covariate state
# check if it is in the current model:
# propose to add it if it is not in the current model
if(curcountlist[4]==0){
  newcountlist[4:6]<-1
  newcountparam[4:6]<-rnorm(3,countprop.mean[4:6],countprop.sd[4:6])
  num<-NA
  den<-NA
  # the prior probabilities
  new.l.prior<-sum(l.prior(newcountparam[4:6],countlolim[4:6],countuplim[4:6]))
  # likelihood
  new.lik<-covey.full.bayes.log.lik(c(newdetparam,newcountparam),raneff,covey.data)
  cur.lik<-covey.full.bayes.log.lik(c(curdetparam,curcountparam),raneff,covey.data)
  # proposal density
  prop.dens<-sum(log(dnorm(newcountparam[4:6],countprop.mean[4:6],countprop.sd[4:6])))
  # numerator and denominator in equation 5
  num<-new.lik+new.l.prior
  den<-cur.lik+prop.dens
}
# propose to delete it if it is in the current model
else{
  newcountlist[4:6]<-0
  newcountparam[4:6]<-0
  num<-NA
  den<-NA
  # the prior probabilities
  cur.l.prior<-sum(l.prior(curcountparam[4:6],countlolim[4:6],countuplim[4:6]))
  # likelihood
  new.lik<-covey.full.bayes.log.lik(c(newdetparam,newcountparam),raneff,covey.data)
  cur.lik<-covey.full.bayes.log.lik(c(curdetparam,curcountparam),raneff,covey.data)
  # proposal density
  prop.dens<-sum(log(dnorm(curcountparam[4:6],countprop.mean[4:6],countprop.sd[4:6])))
  # numerator and denominator in equation 5
  num<-new.lik+prop.dens
  den<-cur.lik+cur.l.prior
}
A<-min(1,exp(num-den))
V<-runif(1)
if(V<=A){
  curcountparam[4:6]<-newcountparam[4:6]
  curcountlist[4:6]<-newcountlist[4:6]
}
else{
  newcountparam[4:6]<-curcountparam[4:6]
  newcountlist[4:6]<-curcountlist[4:6]
}
# determining the current models and storing them
curdetmodel<-match.function(curdetlist,det.list)
curcountmodel<-match.function(curcountlist,count.list)
det.model[i]<-curdetmodel
```

```
  count.model[i]<-curcountmodel
################################ MH step ################################
  newdetparam<-curdetparam
  newcountparam<-curcountparam
  # updating current detection function parameters
  for (b in which(curdetlist==1)){
    num<-NA
    den<-NA
    # proposing to update the bth parameter
    u<-rnorm(1,0,detq.sd[b])
    # remembering that scale intercept and shape are on the log-scale
    if(!is.na(match(b,c(1,2)))){
      newdetparam[b]<-log(exp(curdetparam[b])+u)
      new.l.prior<-l.prior(exp(newdetparam[b]),detlolim[b],detuplim[b])
      cur.l.prior<-l.prior(exp(curdetparam[b]),detlolim[b],detuplim[b])
    }
    else{
      newdetparam[b]<-curdetparam[b]+u
      new.l.prior<-l.prior(newdetparam[b],detlolim[b],detuplim[b])
      cur.l.prior<-l.prior(curdetparam[b],detlolim[b],detuplim[b])
    }
    num<-covey.full.bayes.log.lik(c(newdetparam,newcountparam),raneff,covey.data)
            + new.l.prior
    den<-covey.full.bayes.log.lik(c(curdetparam,curcountparam),raneff,covey.data)
            + cur.l.prior
    A<-min(1,exp(num-den))
    V<-runif(1)
    ifelse(V<=A,curdetparam[b]<-newdetparam[b],newdetparam[b]<-curdetparam[b])
  }
  # updating current count model parameters
  for (b in which(curcountlist==1)){
    num<-NA
    den<-NA
    # proposing to update the bth parameter
    u<-rnorm(1,0,countq.sd[b])
    if(b==length(curcountparam)){newcountparam[b]<-log(exp(curcountparam[b])+u)
      new.l.prior<-l.prior(exp(newcountparam[b]),countlolim[b],countuplim[b])
      cur.l.prior<-l.prior(exp(curcountparam[b]),countlolim[b],countuplim[b])
    }
    else{
      newcountparam[b]<-curcountparam[b]+u
      new.l.prior<-l.prior(newcountparam[b],countlolim[b],countuplim[b])
      cur.l.prior<-l.prior(curcountparam[b],countlolim[b],countuplim[b])
    }
    num<-covey.full.bayes.log.lik(c(newdetparam,newcountparam),raneff,covey.data)
          + new.l.prior
    den<-covey.full.bayes.log.lik(c(curdetparam,curcountparam),raneff,covey.data)
          + cur.l.prior
    A<-min(1,exp(num-den))
    V<-runif(1)
    ifelse(V<=A,curcountparam[b]<-newcountparam[b],newcountparam[b]<-curcountparam[b])
  }
  detparam.mat[i,]<-curdetparam
  countparam.mat[i,]<-curcountparam
  # updating the random effect coefficients
  raneff<-covey.full.bayes.log.lik.raneff.update(c(curdetparam,curcountparam),
                                                 raneff, covey.data)
```

```
  raneff.mat[i,]<-raneff
  # saving the parameter matrices every 5000 iterations
  if(!is.na(match(i,seq(0,nt,5000))==T)){
    save(detparam.mat,file='detparam.mat.RData')
    save(countparam.mat,file='countparam.mat.RData')
    save(raneff.mat,file='raneff.mat.RData')
    save(det.model,file="det.model.RData")
    save(count.model,file="count.model.RData")
  }
  print(paste("this iteration took ",round(unclass(Sys.time())-t1,2)," seconds",sep=""))
  t1<-unclass(Sys.time())
# The following lines allow you to check on if the chain is moving freely across the
# parameter space every 500 iterations without stopping the algorithm
#    if(!is.na(match(i,seq(0,nt,500))==T)){
#      par(mfrow=c(2,2))
#      for (k in 1:13){
#        plot(param.mat[1:i,k],xlab=k,t='l',main=i)
#      }
#   }

# The following lines allow you to check on model mixing every 500 iterations without
# stopping the algorithm
#   if(!is.na(match(i,seq(0,nt,500))==T)){
#     plot(det.model[1:i],t='l')
#     plot(count.model[1:i],t='l')
#     print(table(det.model))
#     print(table(count.model))
#   }
}# end of i loop
```

3.8 Pilot tuning the RJMCMC algorithm

For pilot tuning the within-model moves (the MH step), i.e. adjusting prior and proposal distributions, refer to the section Pilot tuning the MH updating algorithm above.

Pilot tuning the between-model move (the RJ step), may involve adjusting the proposal distributions for the RJ step and the prior distributions. Decreasing the standard deviation of the normal proposal distribution for a parameter and/or widening the boundaries of a uniform prior distribution may decrease the acceptance probability of a proposal to include the respective parameter in the current model.

Potential problems with respect to model mixing for random effect models may occur when the acceptance of a covariate into a model is prevented because the random effect coefficients have absorbed the effect of the covariate. This may be diagnosed by plotting individual means of random effect coefficient values from the chain stored in *raneff.mat*. One potential method for preventing this is to use hierarchical centering (Oedekoven et al. submitted)).

> One strategy to define the mean and standard deviations of the normal proposal distributions for the RJ step is to use the point estimates and standard errors from the maximum likelihood two-stage approach (see chapter 7 of the book for details).

3.9 Inference from an RJMCMC algorithm

3.9.1 Inference on model probabilities from an RJMCMC algorithm

After the algorithm has completed, we can use the data frames that store the model choices and parameter values to draw inference. For the purpose of the exercise, these data frames may also be uploaded into the workspace using:

```
# if you do not wish to complete the algorithm, you may
# load the data frames from the completed algorithm
# for completing the exercise
load("det.model.rdata")
load("detparam.mat.rdata")
load("count.model.rdata")
load("countparam.mat.rdata")
load("Dbl.RJ.2.rdata")
```

For the RJMCMC algorithm, we may draw inference on both the model probabilities and the model parameters by obtaining summary statistics of the respective marginal posterior distributions. As for the MH algorithm, we consider the first 9999 iterations of the chain as the burn-in and omit these from obtaining summary statistics. We may use the following code to obtain model probabilities:

```
# to calculate model probabilities (omiting the burn-in)
kable(t(round(table(det.model[10000:nt])/length(det.model[10000:nt]),6)))
```

1	2
0.999	0.001

```
round(table(count.model[10000:nt])/length(count.model[10000:nt]),6)
```

```
8
1
```

We conclude that the preferred detection model was model 2 which included the *Type* covariate. After the burn-in, this model was selected in 0.1% of the iterations. The second most preferred detection model was the global model without any covariates (model 1) which was selected in 99.9% of the iterations. The full model (model 4) including covariates *Type* and *State* was selected in $< 0.01\%$ of the iterations. For the count model, the full model (model 8) with all covariates including *Type*, *JDctr* and *State* was selected in 100% of the iterations.

3.9.2 Inference on parameters from an RJMCMC algorithm

For inference on the parameters, we use the parameter values stored in `detparam.mat` and `countparam.mat` and calculate the mean, standard deviation and central 95% credible intervals. We consider the first 9999 iterations as the burn-in phase and omit these. Furthermore, we now have several choices about which iterations we include for presenting summary statistics for each parameter. We can use all iterations after the burn-in and consider the value taken by a parameter that was not in the model to be 0. We could also present summary statistics for parameters conditional on the corresponding covariates being in the model (hence excluding those iterations during which the covariates were not in the model). We could also condition on the preferred model and present summary statistics for parameters conditional on the model choice. However, we have to consider that when covariates are correlated, estimates of the coefficient for one covariate with another covariate absent may not be comparable with estimates when the other covariate is present.

The parameter of interest for our case study was the coefficient for level "TREAT" for the *Type* covariate in the count model. From the results of the MH algorithm we concluded that the mean estimate for this parameter was 0.68 (SD = 0.103, 95% central credible intervals = {0.47, 0.87}). We will now investigate whether the inference on this parameter changes when including model selection in the RJMCMC algorithm. For this purpose, we obtain summary statistics for this parameter using the parameter values stored in `countparam.mat`. As *Type* was also included in most but not all of the detection models, inference on this parameter might change depending on which iterations we include in the model inference. The *Type* covariate was included in all count models. However, three different detection models were selected during the RJMCMC algorithm; one of these did not include the *Type* covariate in the detection model while the other two did. We use the following R code to produce summary statistics for three types of scenarios: 1. using all iterations after the burn-in, 2. conditioning on *Type* being part of the detection model and 3. conditioning on the preferred detection model.

```
[1] 0.33

[1] 0.06

 2.5% 97.5%
 0.21  0.46

[1] 0.78

[1] 0.03

 2.5% 97.5%
 0.72  0.82

[1] 0.78

[1] 0.03

 2.5% 97.5%
 0.72  0.82
```

```
x24<-which(det.model[10000:nt]%in%c(2,4))+9999
x2<-which(det.model[10000:nt]==2)+9999
infmat<-matrix(c(round(mean(countparam.mat[10000:nt,2]),2),
round(sd(countparam.mat[10000:nt,2]),2),
round(quantile(countparam.mat[10000:nt,2],probs=c(0.025)),2),
round(quantile(countparam.mat[10000:nt,2],probs=c(0.975)),2),
round(mean(countparam.mat[x24,2]),2),
round(sd(countparam.mat[x24,2]),2),
round(quantile(countparam.mat[x24,2],probs=c(0.025)),2),
round(quantile(countparam.mat[x24,2],probs=c(0.975)),2),
round(mean(countparam.mat[x2,2]),2),
round(sd(countparam.mat[x2,2]),2),
round(quantile(countparam.mat[x2,2],probs=c(0.025)),2),
round(quantile(countparam.mat[x2,2],probs=c(0.975)),2)),3,4,byrow=TRUE)
colnames(infmat)<-c("Mean", "SD", "LoLim","UpLim")
rownames(infmat)<-c("All","M2&4","M2" )
```

Table 10: Mean, standard deviation (SD), lower (LoLim) and upper (UpLim) limit of 95% central credible intervals for the three inference scenarios (all iterations, conditioning on detection models 2 and 4, conditioning on detection model 2).

	Mean	SD	LoLim	UpLim
All	0.33	0.06	0.21	0.46
M2&4	0.78	0.03	0.72	0.82
M2	0.78	0.03	0.72	0.82

In this case, inference on the parameter for level "TREAT" of the *Type* covariate in the count model is unaffected by whether we include iterations from only the preferred detection model 2 or from both detection models that include *Type* (2 and 4). When including all iterations (and hence the 5% of iterations for which *Type* was not part of the detection model), the mean estimate for the parameter is slightly smaller (0.67 compared to 0.69 for the other two scenarios). Also, the standard deviation increases and 95% credible intervals widen for scenario 1.

3.9.3 Inference on plot density from an RJMCMC algorithm

Similar to inference on the parameters for an RJMCMC algorithm, we can include all iterations from the chain or condition on the preferred models for obtaining summary statistics for plot density. For further details, see sections Inference on parameters from an RJMCMC algorithm and Inference on plot density from an MH updating algorithm. The summary statistics for baseline plot density conditional on the preferred models can be calculated using the `Dbl.RJ.2` object provided with the exercise.

4 Maximum likelihood methods for a full likelihood approach

In this section, we use the preferred model from the RJMCMC algorithm for fitting a full likelihood model using maximum likelihood methods. This model contained the covariate ***Type*** in the detection model and the covariates ***Type***, ***JDctr*** and ***State*** in the count model. In contrast to the Bayesian approach above, we do not need to define prior and proposal distributions or set up data frames which store parameter values during the algorithm. We only need to define a likelihood function which returns the negative log-likelihood – negative as we use the R function `nlm` for optimizing which returns the parameter values which minimize the return of the function. In short, the steps required for this approach can be summarized as:

1. Defining the likelihood function
2. Obtaining maximum likelihood estimates
3. Extracting maximum likelihood estimates, standard errors and AIC
4. Inference on plot density

We note, however, that when using the `nlm` function, the user is required to provide starting values for the parameters. Model selection can be done by fitting a range of plausible models and comparing AIC values. Forward step-wise or backward step-wise methods can be used – see Oedekoven et al. (2013).

4.1 Defining the likelihood function for the maximum likelihood approach

As for the Bayesian approach above, we use the full likelihood which combines the detection and count model $\mathcal{L}_{n,y|z} = \mathcal{L}_n \times \mathcal{L}_{y|z}$. As above we use the log-likelihood $\ell_{n,y|z} = \log_e\left(\mathcal{L}_{n,y|z}\right) = \log_e\left(\mathcal{L}_n\right) + \log_e\left(\mathcal{L}_{y|z}\right)$.

In contrast to the Bayesian likelihood functions above, the function `covey.ml.log.lik` is model-specific as only parameters that are switched on in the model that is fitted to the data should be part of parameter string `p`. The function `covey.ml.log.lik` calculates the negative log-likelihood given a set of values for parameters `p` of the defined model for the covey data. It has three arguments: `p`, `datax` (here `datax = covey.data`) and `lim` (limits for integrating out the random effect). In contrast to the Bayesian approach, we do not need to include the random effect coefficients as an argument for the function. We integrate the random effect out instead. The argument `lim` makes it easier to define limits for the integration – values other than infinity may be desirable as infinity may lead to numerical problems. For small random effects standard deviations (i.e. less than $\sigma_l < 1$), using `lim = 5` will return approximately the same results as using `lim = $Inf$`. The `covey.ml.log.lik` function expects certain elements in the data; hence, it is essential that the covey data are in the format described above, i.e., a data object created with the `create.data` function.

In the following we look at each component of the likelihood $\mathcal{L}_n$ and $\mathcal{L}_{y|z}$ separately and present the R code of the `covey.ml.log.lik` that corresponds to the respective components. The code in `covey.ml.log.lik` can be divided into 3 parts which we discuss individually below.

4.1.1 Assigning the parameter values `p`

For illustrating the maximum likelihood approach, we use the preferred model from the RJMCMC algorithm with the factor covariate ***Type*** in the hazard-rate detection model and the full set of covariates available for the count model which includes the factor covariates ***Type*** and ***State*** and the continuous covariate ***JDctr*** which represents Julian date centred around its mean.

The following three code chunks subdivide the function `covey.ml.log.lik` into sections that can be readily explained. The sections themselves are incomplete (i.e. the chunk immediately following does not contain the entire function).

```
covey.ml.log.lik<-function(p,datax,lim){
## Part 1 of covey.ml.log.lik(): setting up the parameter values p for covariates
# the detection model
scale.int<-p[1]      # the scale intercept
shape<-exp(p[2])     # the shape parameter on the log-scale
sig.t<-c(0,p[3])     # coefficient for Type level "TREAT" (level ""CONTROL" is absorbed
                     # in the intercept)
# the count model
int<-p[4]            # the intercept
typ<-c(0,p[5])       # coefficient for Type level "TREAT" (level "CONTROL" is absorbed
                     # in the intercept)
day<-p[6]            # coefficient for JDctr
st<-c(0,p[7:9])      # state coefficients for levels "MS","NC","TN" (level "MO" is
                     # absorbed in the intercept)
std.ran<-exp(p[10]) # the random effects standard deviation on the log-scale
```

In the case that the user wishes to alter the covariates included in the model using the maximum likelihood approach, it is essential to only include parameters in argument `p` that will be estimated. This is in contrast to the Bayesian approach where the likelihood could be defined for the full model and parameters set to zero which are not included in the current model.

4.1.2 The likelihood component for the detection function - maximum likelihood methods

As for the Bayesian approach, we use the detection data which are stored in the data frame `covey.data$dis.object` in the column `distance`. The truncation distance w is stored in `covey.data$w`. With covariates z in the detection function, we use equation 8.11 of the book as the likelihood component for the detection function which – after log-transforming – is given by equation 3 above. For details see Section The likelihood component for the detection function above.

The following R code is the part of the function `covey.ml.log.lik` which calculates the log-likelihood for the detection model $\log_e\left(\mathcal{L}_{y|z}\right)$. It is nearly identical to the equivalent section from the Bayesian approach, except that we do not include the *State* covariate.

```
# Part 2 of covey.ml.log.lik():
# the likelihood component pertaining to the detection model
# calculating the f(y) for each observed distances
le<-nrow(datax$dis.object)
fe<-numeric(le)
alltype<-sort(unique(datax$dis.object$Type))
dis.type<-match(datax$dis.object$Type,alltype)
# the sigma(z) for each detection
allscale<-exp(scale.int+sig.t[dis.type])
# calculating the f(y) for each observation
# note that the truncation distance is stored in datax$w
for (e in 1:le){
fe[e]<-f.haz.function.pt(datax$dis.object$distance[e],allscale[e],shape)/
       integrate(f.haz.function.pt,0,datax$w,allscale[e],shape)$value
}
# the sum of the log(f(y))
log.e<-sum(log(fe))
```

If the user wishes to fit a half-normal detection function to point transect data, the function `f.hn.function.pt` should be used instead of `f.haz.function`. If the user is analyzing line transect data, the functions `f.haz.function.ln` and `f.hn.function.ln` should be used. For fitting models to binned distance data, the likelihood changes to the multinomial likelihood given by equation 6.26 of the book.

4.1.3 The likelihood component for the count model

As for the Bayesian approach, we use a Poisson likelihood for the count model where the expected value λ is modelled as a function of covariates x_q. Due to the repeated counts at the sites, we use the methods described in Section 8.2.5.1 of the book and include a random effect b_l for each site in the count model for which we assume normality with $b_l \sim N(0, \sigma_l^2)$. In the following subscript l refers to the different sites and subscript t to the repeated surveys. As each site also consisted of two points we also include a subscript for point k. The expected value λ is then given by equation 4 above.

As this survey design has multiple points per site and repeat visits, we combine the likelihood functions from equations 8.31 and 8.33 of the book. Due to including the random effect, we add the normal densities for the random effect coefficients to the Poisson likelihoods given for counts (Oedekoven et al. 2013). However, in contrast to the Bayesian approach above, the random effect is now integrated out analytically. As a consequence, we cannot omit the integral for the random effect from the likelihood which, after log-transforming, is given by:

$$\log_e \mathcal{L}_n = \sum_{l=1}^{L} \log_e \left[\int_{-\infty}^{\infty} \left\{ \prod_{k=1}^{K} \prod_{t=1}^{T_k} \frac{\lambda_{lkt}^{n_{lkt}} \exp[-\lambda_{lkt}]}{n_{lkt}!} \right\} \times \frac{1}{\sqrt{2\pi\sigma_b^2}} \exp\left[-\frac{b_l^2}{2\sigma_b^2} \right] db_l \right] ,$$

where L is the total number of sites (183 for our case study), K is the total number of points per site (2 for each site for our case study) and T_k is the number of repeat visits to the kth point (ranging between 1 and 4 for our case study).

We use the following function for integrating out the random effect (which is called internally by `covey.ml.log.lik`).

```
bl.function<-function(bl,obs,lam,std.ran){
  l<-length(bl)
  obs.prob<-array(NA,l)
  for (b in 1:l){
    lam2<-lam*exp(bl[b])
    obs.prob[b]<-prod(dpois(obs,lam2))*dnorm(bl[b],0,std.ran)
  }
  obs.prob
}
```

For the count model, we use the count data which is a data frame stored in `covey.data$glmm.data`. Here, each record represents a single visit to a point and numbers of detections within the truncation distance w are summed up for each visit in the column `covey.data$glmm.data$detections`. The part of the `covey.ml.log.lik` function that calculates the count model likelihood component is given in the following R code:

```
# Part 3 of covey.ml.log.lik(): the likelihood component pertaining to the count model
# setting up indices for the factor covariates and random effect coefficients
  State0<-match(datax$glmm.data$State,sort(unique(datax$glmm.data$State)))
  Type0<-match(datax$glmm.data$Type,sort(unique(datax$glmm.data$Type)))
  gr.id<-sort(unique(datax$glmm.data$gr.id))
  Ran0<-match(datax$glmm.data$gr.id,gr.id)
  J<-length(unique(datax$glmm.data$gr.id))
  # calculate the effective area for each visit to a point
  glmm.sig<-exp(scale.int+sig.t[Type0])#+sig.st[State0])
  n.ptvis<-nrow(covey.data$glmm.data)
  l.efa<-array(NA,n.ptvis)
  for (j in 1:n.ptvis){
    l.efa[j]<-log(integrate(f.haz.function.pt,0,500,glmm.sig[j],shape)$value*pi*2)
  }
  # calculate the log of the likelihood for each site while integrating out the random effect
  # the expected value for each count without the random effect coefficient
  lambda<-exp(int + (typ[Type0]) + (day*datax$glmm.data$JDctr) + st[State0]  + l.efa)
  lik<-array(NA,J)
  for (j in 1:J){
    j.rows<-which(Ran0==j)
    lik[j]<-integrate(bl.function,-lim,lim,obs=datax$glmm.data$detections[j.rows],
```

```
                    lam=lambda[j.rows],std.ran=std.ran)$value
  }
  log.lik<-sum(log(lik))
  log.lik.all<-log.lik+log.e
  # returns the negative log-likelihood as nlm minimizes
  -log.lik.all
}
```

In the case that the user wants to analyse binned distance data, the same modifications apply as described above. We note, however, that models where distances are analysed as exact and models where distances are analysed as binned cannot be compared using criteria such as AIC. (See R code below for obtaining AIC values using this approach). This is because the data change between the two approaches.

4.2 Obtaining maximum likelihood estimates

We use the `nlm` function for optimizing the likelihood function `covey.ml.log.lik`. This function requires that the user provides starting values for the parameters using its argument `p`. For simplicity, we use the same starting values as for our MH algorithm above. Setting the argument `hessian` to *TRUE* estimates the Hessian matrix which we require for obtaining standard errors for the parameter estimates.

```
## initial values for the detection function parameters
# MCDS with covariates type and state
# scale intercept and shape (enter the model on the log-scale)
sig.0<-log(130) # the scale intercept
sha.0<-log(2.5) # shape parameter
# type coefficient
sigtr.0<-0.2    # level treat
## initial values for count model parameters
int.0<- -13     # intercept
# type coefficient
typ.0<- 0.4     # level TREAT
# Julian Date coefficient (covariate JDctr)
day.0<- -0.01   # continuous covariate
# state coefficients
stms.0<- 0.014  # level MS
stnc.0<- -0.38  # level NC
sttn.0<- -1.36  # level TN
# random effects standard deviation and coefficients
std.ran.0<-log(1)
# combining the initial values in an array for input into the covey.ml.log.lik function
p <-c(sig.0,sha.0,sigtr.0,int.0,typ.0,day.0,stms.0,stnc.0,sttn.0,std.ran.0)
# testing the function using the initial values
covey.ml.log.lik(p=p,datax=covey.data,lim=5)
# minimizing the function (i.e. maximizing the likelihood)
nlm1<-nlm(f=covey.ml.log.lik,p=p,hessian=TRUE,datax=covey.data,lim=5)
```

For the purpose of continuing the exercise without waiting for the algorithm to converge, we can load the `nlm1` into the workspace:

```
# upload the resulting object into the workspace
load("nlm1.rdata")
```

We can use the object `nlm1` to extract parameter estimates, the AIC value and the standard errors for the parameter estimates. We have to remember, however, that the random effects standard deviation enters the model on the log-scale. Hence, we need to back-transform the estimate and the standard error. The random effects standard deviation is the 10^{th} parameter of the model.

```
# the negative log-likelihood value
nlm1$minimum
```

```
[1] 7535.147
```

```
# AIC
nlm1$minimum * 2 + (2 * length(nlm1$estimate))
```

```
[1] 15090.29
```

```
# parameter estimates
round(nlm1$estimate,4)
```

```
 [1]   5.8382   1.2912  -0.2466 -13.1559   0.6823  -0.0234  -0.4301
 [8]  -1.4633  -1.3005  -0.3450
```

```
# standard errors
round(sqrt(diag(solve(nlm1$hessian))),4)
```

```
Warning in sqrt(diag(solve(nlm1$hessian))): NaNs produced
```

```
 [1]    NaN 0.0747 0.0382 0.1130 0.0781 0.0021 0.1718 0.1198 0.1922 0.0904
```

```
# to convert the parameter estimate and standard error for the random effects
# standard deviation back from the log-scale we use:
round(exp(nlm1$estimate[10]),2)
```

```
[1] 0.71
```

```
round(sqrt(diag(solve(nlm1$hessian)))[10]*exp(nlm1$estimate[10]),4)
```

```
Warning in sqrt(diag(solve(nlm1$hessian))): NaNs produced
```

```
[1] 0.064
```

We note that the algorithm did not converge properly. This was indicated by the NA in the array of standard errors as well as the *code* returned by the `nlm` function with `nlm1$code` = 3.

4.2.1 Solving the convergence problem

One method for solving the convergence problem is to fix the shape parameter at the estimate obtained above. This requires a small alteration of the first part of the likelihood function (which we call `covey.ml.log.lik2`) where we set up the parameters:

The code chunk that follows is merely the first part of `covey.ml.log.lik2`; it does not contain the entire function.

```
# this function is uploaded into your workspace using the source command above
covey.ml.log.lik2<-function(p,datax,lim){
## Part 1 of covey.ml.log.lik2(): setting up the parameter values p for covariates
# the detection model
scale.int<-p[1]      # the scale intercept
shape<-exp(1.291197)# the fixed shape parameter is not estimated
```

```
sig.t<-c(0,p[2])     # coefficient for Type level "TREAT" (level "CONTROL"" is absorbed
                     # in the intercept)
# the count model
int<-p[3]            # the intercept
typ<-c(0,p[4])       # coefficient for Type level "TREAT" (level "CONTROL" is absorbed
                     # in the intercept)
day<-p[5]            # coefficient for JDctr
st<-c(0,p[6:8])      # state coefficients for levels "MS","NC","TN" (level "MO" is
                     # absorbed in the intercept)
std.ran<-exp(p[9])   # the random effects standard deviation on the log-scale
```

The remainder of the `covey.ml.log.lik2` function is equivalent to part 2 and part 3 of the `covey.ml.log.lik` function above. Again, we use the `nlm` function to maximize the likelihood. As we do not estimate the shape parameter of the hazard-rate detection function, we have one parameter less to estimate compared to `covey.ml.log.lik`.

```
# combining the initial values in an array for input into the covey.ml.log.lik2 function
p2 <-c(sig.0,sigtr.0,int.0,typ.0,day.0,stms.0,stnc.0,sttn.0,std.ran.0)
# minimizing the function (i.e. maximizing the likelihood)
nlm2<-nlm(f=covey.ml.log.lik2,p=p2,hessian=TRUE,datax=covey.data,lim=5)
```

For the purpose of continuing the exercise without waiting for the algorithm to converge, we can upload the `nlm2` object using:

```
# upload the resulting object into the workspace
load("nlm2.rdata")
```

```
# the negative log-likelihood value
nlm2$minimum
```

```
[1] 7535.146
```

```
# AIC
nlm2$minimum * 2 + (2 * length(nlm2$estimate))
```

```
[1] 15088.29
```

```
# parameter estimates
round(nlm2$estimate,4)
```

```
[1]   5.8379  -0.2463 -13.1543   0.6818  -0.0233  -0.4290  -1.4601  -1.3004
[9]  -0.3471
```

```
# standard errors
round(sqrt(diag(solve(nlm2$hessian))),4)
```

```
[1] 0.0432 0.0532 0.1254 0.0972 0.0023 0.1717 0.1843 0.1920 0.0919
```

```
# to convert the parameter estimate and standard error for the random effects standard
# deviation back from the log-scale we use:
round(exp(nlm2$estimate[9]),2)
```

```
[1] 0.71
```

```
round(sqrt(diag(solve(nlm2$hessian)))[9]*exp(nlm2$estimate[9]),4)
```

```
[1] 0.065
```

4.2.2 Comparing the parameter estimates from the Bayesian and maximum likelihood approaches

The means and standard deviations of the parameters obtained from the marginal posterior distribution conditional on the preferred model from the RJMCMC algorithm were similar to the maximum likelihood estimates and standard errors (Table 2). This is expected as we placed uniform priors on all parameters. All standard errors in Table 2 were slightly smaller compared to the standard deviations; however, this may be because one less parameter was estimated for the maximum likelihood approach.

Table 2. Mean and standard deviation (SD) of the marginal posterior distribution of parameter estimates conditional on the preferred model from the RJMCMC algorithm as well as maximum likelihood estimates (MLE) and standard errors (SE) obtained with a maximum likelihood approach of the same model with a fixed shape parameter.

Table 11: Mean and standard deviation (SD) of the marginal posterior distribution of parameter estimates conditional on the preferred model from the RJMCMC algorithm as well as maximum likelihood estimates (MLE) and standard errors (SE) obtained with a maximum likelihood approach of the same model with a fixed shape parameter.

	Mean	SD	MLE	SE
Scale int	5.829	0.008	5.838	0.0432
Shape	1.248	0.032	1.291	NA
Scale Type	-0.293	0.030	-0.246	0.0532
Int	-13.261	0.095	-13.154	0.1254
Type	0.783	0.034	0.682	0.0972
JDctr	-0.028	0.002	-0.023	0.0023
StateMS	-0.482	0.017	-0.429	0.1717
StateNC	-1.398	0.140	-1.460	0.1843
StateTN	-1.041	0.108	-1.300	0.192
RE SD	-0.283	0.018	-0.347	0.0919

4.2.3 Inference on plot density from the maximum likelihood approach

If we wish to draw inference on plot density, we can use the parameter estimates from the count model in the R object ***nlm2***. We note that while λ_{lkt} (equation 4) models the expected counts, $\exp\left(\sum_{q=1}^{Q} x_{qklt}\beta_q + b_l\right)$ from the λ_{lkt} model (i.e., without the effective area $\log_e(\nu_{lkt})$) models the densities (i.e., number of coveys per m^2)

As for the Bayesian approach, we can obtain posterior distributions of plot density for any given combination of covariates. For comparison with the Bayesian approach, we obtain estimates of baseline plot density where each covariate value is set to their baseline level (level for ***Type*** = "CONTROL", ***JDctr*** = 0, level for ***State*** = "MO"). We further need to add a contribution from the random effect to adjust for bias: $0.5 \times \sigma_l^2$. Hence, the expected baseline density $D_{bl.ml}$ can be expressed as: $D_{bl.ml} = exp(\beta_1 + (0.5 \times \sigma_l^2)))$, where β_1 is the intercept for the count model and σ_l is the random effects standard deviation. As distance measurements were metres (`covey.data$dis.unit` = m) we need to convert the density estimates from $D_{bl.ml}/m^2$ to $D_{bl.ml}/km^2$.

```
# parameter 3 of the covey.ml.log.lik2 function is the count model intercept,
# parameter 9 is the random effects standard deviation
Dbl.ml<-exp(nlm2$estimate[3] + (0.5 * exp(nlm2$estimate[9])^2))
# Density per km^2
Dbl.ml<-Dbl.ml*1000*1000
```

The estimate of $D_{bl.ml}$ is a function $f_D(\beta, \theta)$ of the parameter estimates in the R object `nlm2`, β, θ. Hence, the variance of $\hat{D}_{bl.ml}$ is a function of the Hessian in `nlm2`:

$$\widehat{var}(\hat{D}_{bl.ml}) = \left[\left.\frac{\partial f_D(\beta,\theta)}{\partial(\beta,\theta)}\right|_{(\beta,\theta)=(\hat{\beta},\hat{\theta})}\right]^T \mathbf{H^{-1}} \left[\left.\frac{\partial f_D(\beta,\theta)}{\partial(\beta,\theta)}\right|_{(\beta,\theta)=(\hat{\beta},\hat{\theta})}\right]$$

The derivatives of the parameters are generally difficult to obtain. However, a numerical approximation of the derivatives evaluated at the maximum likelihood estimates $\hat{\beta}, \hat{\theta}$ can be obtained via finite differences. For example, we use the following for the derivative of the ith element of β:

$$\left.\frac{\partial f_D(\beta_i,\hat{\beta}_{-\mathbf{i}},\theta)}{\partial\beta_i}\right|_{\beta_i=\hat{\beta}_i} = \frac{f_D(\hat{\beta}_i+\delta\hat{\beta}_i,\hat{\beta}_{-\mathbf{i}},\theta) - f_D(\hat{\beta}_i-\delta\hat{\beta}_i,\hat{\beta}_{-\mathbf{i}},\theta)}{2\delta\hat{\beta}_i} \quad (6)$$

The following R function uses this numerical approximation of the derivatives for obtaining a variance estimate for our baseline density estimate $D_{bl.ml}$ where the value for δ from equation 6 is user defined.

```
var.Dblml<-function(par,hessian,delta){
int<-par[3]
log.resd<-par[9]
deriv<-matrix(0,1,9)
deriv[1,3]<-(exp(int+(delta*int))*exp(0.5*exp(log.resd)^2)-exp(int-(delta*int))
             *exp(0.5*exp(log.resd)^2))/(2*delta*int)
deriv[1,9]<-(exp(int)*exp(0.5*exp(log.resd+(delta*log.resd))^2)-exp(int)
             *exp(0.5*exp(log.resd-(delta*log.resd))^2))/(2*delta*log.resd)
varian<-deriv%*%solve(hessian)%*%t(deriv)
varian
}
```

We use this function to obtain standard errors for the density estimate. The units for density were originally m^2. However, we wish to obtain the standard error for density per km^2. Hence, we need to convert the standard errors accordingly.

```
# We take the square root to obtain the standard error for the density estimate
Dbl.ml.se<-sqrt(var.Dblml(nlm2$estimate,nlm2$hessian,delta=0.0001))*1000*1000
```

```
Dbl.RJ.2<-array(NA,nt)
# we only use iterations with detection model 2 after the burn-in
for (i in x2){
  Dbl.RJ.2[i]<-exp(countparam.mat[i,1] + (0.5 * exp(countparam.mat[i,7])^2))
}
# Density per km^2
Dbl.RJ.2<-Dbl.RJ.2*1000*1000
```

Our estimated baseline density of coveys (coveys per km^2) using the maximum likelihood approach is 2.49 (SE = 0.31). In comparison, the mean estimate of baseline density obtained using the RJMCMC approach (conditional on the preferred model, see above) is 2.32 (SD = 0.27).

5 Summary

We presented Bayesian as well as maximum likelihood methods including R code for analysing distance sampling data using the full likelihood from chapter 8. The full likelihood methods have the advantage that parameter estimates of the detection and count models are obtained simultaneously and do not require conditioning on a first stage detection model (as the two-stage approach from chapter 7). For the full likelihood approach, uncertainty from the detection model propagates into the count model. We used multiple covariate distance sampling methods for the detection model where the scale parameter was a function of covariates. For the count model, we used a plot count model which was extended with a random effect for site.

One of the benefits of the Bayesian approach is that it is possible to include prior information. This may be particularly useful, e.g. in the case that the detection model needs to be fitted to a small number of detections. It might also be easier to integrate

out the random effect using the data augmentation scheme where the individual coefficients are included in the model and updated during each iteration of the chain. Selection between and averaging over a large range of possible models can be implemented with RJMCMC methods. Some of the disadvantages for the Bayesian approach are that the algorithm may take a long time to complete and that it might be difficult to find appropriate proposal distributions, in particular for the RJ step.

The maximum likelihood approach does not require defining prior or proposal distributions. One of the disadvantages for the maximum likelihood approach is that, in the case of random effect models, the analytic integration of the random effect may lead to problems for calculating the parameter estimates and the Hessian matrix. No prior information can be included.

6 Acknowledgements

The national CP33 monitoring program was coordinated and delivered by the Department of Wildlife, Fisheries, and Aquaculture and the Forest and Wildlife Research Center, Mississippi State University. The national CP33 monitoring program was funded by the Multistate Conservation Grant Program (Grants MS M-1-T, MS M-2-R), a program supported with funds from the Wildlife and Sport Fish Restoration Program and jointly managed by the Association of Fish and Wildlife Agencies, U.S. Fish and Wildlife Service, USDA-Farm Service Agency, and USDA-Natural Resources Conservation Service-Conservation Effects Assessment Project.

References

Davison, A. C. 2003. Statistical Models. Cambridge University Press, Cambridge.

Gelman, A., G. O. Roberts, and W. R. Gilks. 1996. Efficient Metropolis jumping rules. Pages 599–608 *in* J. M. Bernardo, J. O. Berger, A. P. Dawid, and A. F. M. Smith, editors. Bayesian Statistics. Oxford University Press, Oxford.

Hastings, W. K. 1970. Monte Carlo sampling methods using Markov Chains and their applications. Biometrika 57:97–109.

Oedekoven, C. S., S. T. Buckland, M. L. Mackenzie, K. O. Evans, and L. W. Burger. 2013. Improving distance sampling: Accounting for covariates and non-independency between sampled sites. Journal of Applied Ecology 50:786–793.

Oedekoven, C. S., S. T. Buckland, M. L. Mackenzie, R. King, K. O. Evans, and L. W. Burger. 2014. Bayesian methods for hierarchical distance sampling models. Journal of Agricultural, Biological, and Environmental Statistics 19:219–239.

Oedekoven, C. S., R. King, S. T. Buckland, M. L. Mackenzie, K. O. Evans, and L. W. Burger. submitted. Using hierarchial centering to facilitate a reversible jump MCMC algorithm for random effects models.
