Efficient Estimation of Large-Scale Spatial Capture-Recapture Models

Running Headline: Efficient Estimation of SCR Models

Daniel Turek[*1], Cyril Milleret[2], Torbjørn Ergon[3], Henrik Brøseth[4], and Perry de Valpine[5]

[*]Corresponding author

[1]Williams College, Department of Mathematics & Statistics, Williamstown, MA 01267, USA, dbt1@williams.edu

[2]Norwegian University of Life Sciences, Environmental Sciences and Natural Resource Management, NO-1432 Ås, Norway

[3]Centre for Ecological and Evolutionary Synthesis, Department of Biosciences, University of Oslo, Oslo, Norway

[4]Norwegian Institute for Nature Research, Department of Terrestrial Ecology, NO-7485 Trondheim, Norway

[5]University of California Berkeley, Department of Environmental Science, Policy & Management, Berkeley, CA 94720, USA

Abstract

Capture-recapture methods are a common tool in ecological statistics, which have been extended to spatial capture-recapture models for data accompanied by location information. However, standard formulations of these models can be unwieldy and computationally intractable for large spatial scales, many individuals, and/or activity center movement. We provide a cumulative series of methods that yield dramatic improvements in Markov chain Monte Carlo (MCMC) estimation for two examples. These include removing unnecessary computations, integrating out latent states, vectorizing declarations, and restricting calculations to the locality of individuals. Our approaches leverage the flexibility provided by the `nimble` R package. In our first example, we demonstrate an improvement in MCMC efficiency (the rate of generating effectively independent posterior samples) by a factor of 100. In our second example, we reduce the computing time required to generate 10,000 posterior samples from 4.5 hours down to five minutes, and realize an increase in MCMC efficiency by a factor of 25. We also explain how these approaches can be applied generally to other spatially-indexed hierarchical models. R code is provided for all examples, as well as an executable web-appendix.

Keywords:

Mark-recapture, MCMC, `nimble`, Sampling efficiency, Spatial capture-recapture.

1 Introduction

Capture-recapture methods are primary tools for estimating abundance and demographic parameters in populations. These methods model longitudinal encounter histories of individuals in a population. Spatial capture-recapture (SCR) models account for individual and trap-specific capture probabilities depending on individuals' latent centers of activity

and space-use in relation to the explicit location of traps or other detectors (Efford, 2004; Borchers and Efford, 2008). Closed SCR models provide more precise and robust estimates of population densities than non-spatial models, and also enable estimation of the spatial distribution of individuals and associated parameters .

Despite their popularity, SCR models encounter numerous computational challenges which pose serious obstacles for their practical use (Gardner et al., 2018). For large study areas with many detectors, determining the probability of a capture history becomes very computationally costly because it involves calculations for all detectors, which is problematic for large-scale studies (Milleret et al., 2018b). Modeling the movement of activity centers often induces inefficient MCMC updating, as do methods for imposing spatial constraints on activity center locations. And data augmentation of never-observed individuals can lead to unnecessary calculations.

Bayesian hierarchical models, such as SCR models, are often formulated using the BUGS modeling language (Lunn et al., 2009) and estimated using Markov chain Monte Carlo (MCMC; Brooks et al., 2011). Mainstream MCMC software includes WinBUGS, JAGS (Plummer, 2003), and Stan (Stan Development Team, 2014). Recently, the `nimble` R package has been developed, offering new degrees of customization for MCMC (de Valpine et al., 2017). Custom-written distributions and the flexibility of `nimble`'s MCMC system have provided substantial improvements in non-spatial capture-recapture models (Turek et al., 2016) and the study of MCMC algorithms (Turek et al., 2017).

We use `nimble` to demonstrate several generally applicable techniques for improving MCMC efficiency of (1) a simple but computationally-intense SCR model (Milleret et al., 2019), and (2) an open robust-design SCR model (Ergon and Gardner, 2014). We increase MCMC efficiency by vectorizing calculations, applying custom MCMC sampling strategies, implementing model-specific likelihood calculations, disabling unnecessary model calculations, and restricting trap calculations to the locality of each individual. Using these techniques, we achieve efficiency gains of a factor of 100 in the first example and a factor of 25

in the second example.

2 Materials and Methods

We consider two example SCR models which both present computational challenges. The first ("Wolverine") considers a simple closed SCR model for data from non-invasive genetic sampling of wolverines on a large spatial scale in Norway (*Gulo gulo*, Milleret et al., 2019). The second ("Vole") is a more complex SCR model on a smaller spatial scale, modeling an open population of field voles with activity-center movements (*Microtus agrestis*, Ergon and Gardner, 2014). We first describe each model, followed by the strategies used to improve MCMC efficiency. Finally, we describe the metric used to measure MCMC efficiency.

2.1 Wolverine Model

This example has a spatial extent over 200,000 km^2. The data, collected throughout Norway, consist of 453 detections from 196 individually identified female wolverines using noninvasive genetic sampling and search encounter methods (Milleret et al., 2019). The search area was discretized to a detector grid with a 2km resolution, and only searched grid cells were included in the analysis. This resulted in 17,266 unique detectors, with binary-valued detections of individuals within grid cells. Data and additional details are available at the dryad repository (Milleret et al., 2018a).

The Wolverine model combines a spatial point process model of individual activity centers (ACs), data augmentation to model the true population size, and an observation model for detection probabilities and capture histories. Define the AC of individual i as $\boldsymbol{s}_i = (s_i^x,\ s_i^y)$, where s_i^x and s_i^y follow independent uniform prior distributions spanning the study area. As some regions are unsuitable habitat (*i.e.*, water), AC locations must be constrained. We use a habitat mask by defining a binary matrix $\boldsymbol{H}$ over the study area, where $H_{x,y} = 1$ indicates that cell (x, y) is suitable habitat. AC locations are then constrained as $1 \sim \text{Bernoulli}(H_{s_i^x, s_i^y})$,

where 1 is a unit data value.

For data augmentation (Royle, 2009), we add N_{aug} virtual individuals. The augmented matrix y has dimension $(N_{\text{obs}} + N_{\text{aug}}) \times R$, with $R = 17,266$ detectors and $N_{\text{obs}} = 196$ unique individuals. Define binary variables z_i with independent $z_i \sim \text{Bernoulli}(\phi)$ prior distributions, representing inclusion in the population. For the N_{obs} sighted individuals, $z_i = 1$ is observed data, while the remaining z_i are unobserved. Total population size N is estimated as $N = \sum_{i=1}^{N_{\text{obs}}+N_{\text{aug}}} z_i$, using the prior distribution $\phi \sim \text{Uniform}(0, 1)$ to induce a flat prior on N (Royle et al., 2007).

The probability of detecting individual i at detector r is $p_{i,r} = p_0 \exp(-\frac{1}{2\sigma^2} \|\boldsymbol{s}_i - \boldsymbol{x}_r\|^2)$, where $\boldsymbol{x}_r$ is the location of detector r and p_0 and σ are the maximal and scale of decay for detection probability. Detections are modeled as $y_{i,r} \sim \text{Bernoulli}(p_{i,r}\, z_i)$. The complete Wolverine model definition is given in (1), where indices r take the range $1, \ldots, R$.

$$
\begin{aligned}
\phi &\sim \text{Uniform}(0, 1) \\
p_0 &\sim \text{Uniform}(0, 1) \\
\sigma &\sim \text{Uniform}(0, 50) \\
N &= \textstyle\sum_{i=1}^{N_{\text{obs}}+N_{\text{aug}}} z_i \\
i = 1, \ldots, &(N_{\text{obs}} + N_{\text{aug}}) : \\
s_i^x &\sim \text{Uniform}(x_{\min},\, x_{\max}) \\
s_i^y &\sim \text{Uniform}(y_{\min},\, y_{\max}) \\
1 &\sim \text{Bernoulli}(H_{s_i^x, s_i^y}) \\
z_i &\sim \text{Bernoulli}(\phi) \\
\boldsymbol{s}_i &= (s_i^x,\ s_i^y) \\
p_{i,r} &= p_0 \cdot \exp(-\tfrac{1}{2\sigma^2} \|\boldsymbol{s}_i - \boldsymbol{x}_r\|^2) \\
y_{i,r} &\sim \text{Bernoulli}(p_{i,r}\, z_i)
\end{aligned}
\tag{1}
$$

We use four refinements of the model and MCMC sampling, with the goal to improve MCMC efficiency: (1) Vectorize computations and put the habitat mask into a custom distribution, (2) jointly sample AC components, (3) restrict calculations to local detectors and sparse representation of data, and (4) skip unnecessary calculations when $z_i = 0$. We next describe each of these techniques, and `nimble` code corresponding to each cumulative

refinement appears in Appendix A.

2.1.1 Vectorized Computations

Vectorization refers to carrying out a set of matching model computations more efficiently, as is possible in `nimble` but neither WinBUGS or JAGS. `nimble` supports vectorized model declarations, reducing the total nodes in the model and potentially improving MCMC efficiency. We vectorized both detection probabilities and data likelihoods for each individual across the R detectors. For the vector of detection probabilities $\boldsymbol{p}_{i,1:R}$, we used a vectorized model declaration. For the vectorized data likelihood of $\boldsymbol{y}_{i,1:R}$, we used a custom likelihood function for the entire (length-R) observation history of one individual.

This technique is only beneficial when the *entire* joint likelihood of $\boldsymbol{y}_{i,1:R}$ is always calculated simultaneously, as is the case here for updates of p_0, σ, or z_i. In a different model, this technique could result in inefficiencies if any MCMC updates require likelihood calculation for only a subset of $\boldsymbol{y}_{i,1:R}$.

2.1.2 Joint Sampling of AC Locations

We apply joint (block) sampling of the s_i^x and s_i^y coordinates of each AC. `nimble` allows the assignment of block samplers to arbitrary variables, applying multi-dimensional Metropolis-Hastings sampling. This results in computational savings since an MCMC update of $\boldsymbol{s}_i$ requires only one calculation of all (length-R) relevant detection probabilities and data likelihoods. In contrast, independent updates of the s_i^x and s_i^y components will require two likelihood evaluations, one for each component.

2.1.3 Local Detector Evaluations and Sparse Observation Matrix

We move detection probability calculations inside the vectorized likelihood, and additionally restrict these calculations to detectors within a maximum realistic radius ($d_{\max}$) of the AC $\boldsymbol{s}_i$. In advance, we identify the set of detectors located within $d_{\max}$ from each cell of the

habitat matrix. The modified distribution identifies the grid cell containing $\boldsymbol{s}_i$, and the set of detectors within $d_{\max}$ from it. Calculations of p_{ir} are then restricted to this set of detectors.

We also convert to a sparse representation of the detection matrix y. In this representation, each row contains the detector identification numbers (values of r) that detected one individual. The number of columns is therefore equal to the maximum number of detections of any particular individual. This sparse representation allows for a smaller model and equivalent, but more efficient, likelihood calculations.

2.1.4 Skip Unnecessary Calculations

Calculations can be avoided when any $z_i = 0$, that is, an augmented virtual individual is not currently included in the population. In that case, neither the distances to each detector nor the detection probabilities need be calculated. We modify the custom likelihood again, to accept z_i as an argument. When $z_i = 1$, the calculations take place as before. When $z_i = 0$, the likelihood is one if the individual was never observed – always the case for augmented individuals – which can be calculated without any distances or detection probabilities. This modification can save substantial computation, especially when N_{aug} is large, that being the conservative approach.

2.2 Vole Robust-Design Model

Our second example considers a robust-design SCR model of field voles in the Kielder Forest of northern England (*Microtus agrestis*, Ergon and Gardner, 2014), with four primary sampling occasions and nested secondary trapping sessions. A total of 158 unique individuals are considered to have static ACs within primary occasions, but to disperse between primary occasions. See Ergon and Gardner (2014, Appendix S2) for further details, (Ergon and Lambin, 2013) for the data, and Appendix B.1 for the original JAGS code.

The Vole model contains individual survival between primary sampling occasions, dispersal of ACs between primary occasions, and spatial capture-recapture from capture histories.

Define the AC of individual i on primary occasion k as $\boldsymbol{s}_{i,k} = (s^x_{i,k},\ s^y_{i,k})$. On first capture, the components s^x_{i,F_i} and s^y_{i,F_i} are given uniform prior distributions spanning the mean location of captures during that occasion. The dispersal between primary occasions k and $k+1$ uses a uniformly-distributed dispersal angle θ_{ik}, and an exponentially-distributed dispersal distance d_{ik} with rate parameter λ_{G_i}, where G_i is the sex of individual i (1: female; 2: male), and λ_1 and λ_2 are sex-specific parameters. Thus, the AC components are related across primary occasions as $s^x_{i,k+1} = s^x_{i,k} + d_{ik}\ \cos(\theta_{ik})$ and $s^y_{i,k+1} = s^y_{i,k} + d_{ik}\ \sin(\theta_{ik})$.

The survival model uses binary indicator variables, where $z_{i,k} = 1$ indicates individual i is alive on occasion k. We condition on the first observation in primary occasion F_i, as $z_{i,F_i} = 1$. The survival process follows as $z_{i,k+1} \sim \text{Bernoulli}((\phi_{G_i})^{T_k}\ z_{i,k})$, where survival probability depends on sex and temporal duration. G_i gives the sex of individual i, T_k is the time (in months) between occasions k and $k+1$, and ϕ_1 and ϕ_2 are sex-specific survival rates. When ϕ_{G_i} is a function of a continuous covariate, the model is only invariant to the choice of time unit of T_k when using a loglog (log-hazard) link (Ergon et al., 2018).

The observation model uses hazard rates to calculate trap capture probabilities. For individual i, on secondary trapping session j of primary occasion k, the capture hazard rate $h_{ijkr} = b_{ijk} \cdot \exp\left(-\left(\frac{\|\boldsymbol{s}_{i,k} - \boldsymbol{x}_r\|}{\sigma_{G_i}}\right)^{\kappa_{G_i}}\right)$, where the location of trap r is $\boldsymbol{x}_r$, and each κ_j and σ_j are sex-specific observation parameters. Baseline hazard is $b_{ijk} = \lambda_0\ (\beta_1)^{I(TOD_{jk}=2)}\ (\beta_2)^{I(G_i=2)}$, using indicator function $I(\cdot)$, time of day TOD_{jk} (1: evening; 2: morning), and baseline hazard rate λ_0. β_1 is the effect of morning trapping sessions, and β_2 is that of males.

Total capture hazard rate is $h_{ijk*} = \sum_{r=1}^{R} h_{ijkr}$. Probability of "no capture" is $\pi_{ijk0} = \exp\left(-h_{ijk*}\ z_{i,k}\right)$, which is unity when $z_{i,k} = 0$. Probability of capture is $\pi_{ijkr} = (1 - \pi_{ijk0})\ \frac{h_{ijkr}}{h_{ijk*}}$ in trap r, accounting for competing risks among traps and satisfying $\sum_{r=0}^{R} \pi_{ijkr} = 1$.

The "ones trick" is used to induce the correct likelihood calculation. Observation data y is a 3-dimensional array, where $y_{ijk} = 0$ indicates that individual i was not captured in trapping session j of primary occasion k, and $y_{ijk} = r$ indicates a capture in trap r. The complete Vole model definition is given in (2), where all indices j take the range of the

number of secondary trapping sessions in the relevant primary occasion k, and all indices r assume the range $1, \ldots, R$.

$$
\begin{aligned}
\beta_1,\ \beta_2 &\sim \text{Uniform}(0.1, 10) \\
p &\sim \text{Uniform}(0.01, 0.99) \\
\lambda_0 &= -\log(1-p) \\
g = 1, 2 &: \\
\kappa_g &\sim \text{Uniform}(0, 50) \\
\sigma_g &\sim \text{Uniform}(0.1, 20) \\
\lambda_g &\sim \text{Uniform}(0, 100) \\
\phi_g &\sim \text{Uniform}(0, 1) \\
i = 1, \ldots, N_{\text{obs}} &: \\
s^x_{i,F_i} &\sim \text{Uniform}(x^i_{\text{min}},\ x^i_{\text{max}}) \\
s^x_{i,F_i} &\sim \text{Uniform}(y^i_{\text{min}}, y^i_{\text{max}}) \\
z_{i,F_i} &= 1 \\
k = F_i, \ldots,\ L-1 &: \\
\theta_{ik} &\sim \text{Uniform}(0, 2\pi) \\
d_{ik} &\sim \text{Exponential}(\lambda_{G_i}) \\
s^x_{i,k+1} &= s^x_{i,k} + d_{ik}\ \cos(\theta_{ik}) \\
s^y_{i,k+1} &= s^y_{i,k} + d_{ik}\ \sin(\theta_{ik}) \\
z_{i,k+1} &\sim \text{Bernoulli}((\phi_{G_i})^{T_k}\ z_{i,k}) \\
k = F_i, \ldots,\ L &: \\
\boldsymbol{s}_{ik} &= (s^x_{i,k},\ s^y_{i,k}) \\
b_{ijk} &= \lambda_0\ (\beta_1)^{I(TOD_{jk}=2)}\ (\beta_2)^{I(G_i=2)} \\
h_{ijkr} &= b_{ijk} \cdot \exp\left(-\left(\frac{\|\boldsymbol{s}_{ik} - \boldsymbol{x}_r\|}{\sigma_{G_i}}\right)^{\kappa_{G_i}}\right) \\
h_{ijk*} &= \textstyle\sum_{r=1}^{R} h_{ijkr} \\
\pi_{ijk0} &= \exp\left(-h_{ijk*}\ z_{ik}\right) \\
\pi_{ijkr} &= (1 - \pi_{ijk0})\ \frac{h_{ijkr}}{h_{ijk*}} \\
1 &\sim \text{Bernoulli}(\pi_{i\,j\,k\,y_{ijk}})
\end{aligned}
\tag{2}
$$

We apply three cumulative refinements to the model and MCMC sampling: (1) Jointly sample correlated dimensions and marginalize over z_i indicator variables, (2) use a custom bivariate dispersal distribution, and (3) restrict trap calculation to the vicinity of each AC.

Next we describe these techniques, and `nimble` code corresponding to each appears in Appendix B.

2.2.1 Joint Sampling and Marginalization

We apply joint samplers for updating two pairs of parameters: $\{\kappa_1, \sigma_1\}$ and $\{\kappa_2, \sigma_2\}$, as these pairs each determine the trap hazard rates for one sex. Trial runs confirm that these pairs exhibit high posterior correlation, so we expect block samplers will improve mixing.

Next, we integrate (marginalize) over the latent $z_{i,k}$ indicator variables to directly calculate the unconditional likelihood of capture histories. This reduces the model size and the dimension of sampling, and can improve MCMC mixing since parameter updates are no longer conditional on the "current" values of each $z_{i,k}$. This is done in `nimble` using a custom likelihood. This calculation is a finite summation over the possible $z_{i,k}$ states, similar to the filtering employed in Turek et al. (2016, Section 2.3.2). When individuals are known to be alive (up to the final capture), the likelihood is survival multiplied by the probability of the observed capture history. Subsequent to the final capture, forward-filtering is used to calculate the likelihood of the remaining non-capture events, accounting for uncertainty in survival.

2.2.2 Custom Dispersal Distribution

We originally modeled dispersal distances and angles as random variables subject to MCMC sampling, a standard approach for movement models. This results in high computational cost because any proposed update to dispersal distance or angle (especially for *early* primary occasions) results in a large chain of calculations to determine the updated ACs, detection probabilities, and detection likelihoods *for all subsequent occasions*. Specifically, say we make an MCMC proposal for modifying d_{11}, the dispersal distance for the first individual, between the first and second primary occasions. This MCMC update will require re-evaluating each $\boldsymbol{s}_{1,2}$, $\boldsymbol{s}_{1,3}$, $\ldots$, $\boldsymbol{s}_{1,L}$, up through the AC of the final primary occasion. Further, detection

probabilities and data likelihoods for each AC also need be recalculated.

We reparameterize this model using a custom distribution of activity center locations that is induced by the distributions of turning angle and distance, as $\boldsymbol{s}_{i,k+1} \sim \text{Dispersal}(\boldsymbol{s}_{ik}, \lambda_{G_i})$. This distribution is centered around the current AC and is mathematically equivalent to the original $\{d, \theta\}$ parameterization. Now, updates of $\boldsymbol{s}_{i,k}$ do *not* induce a large chain of ensuing calculations, but rather, only the likelihoods corresponding to $\boldsymbol{s}_{i,k}$ and $\boldsymbol{s}_{i,k+1}$ must be calculated. The custom distribution is given by $p(\boldsymbol{s}_{ik+1} \,|\, \boldsymbol{s}_{ik}, \lambda_{G_i}) \;\propto\; \left(\frac{1}{d}\right) \cdot \lambda_{G_i}\; e^{-\lambda d}$, where $d = \|\boldsymbol{s}_{ik+1} - \boldsymbol{s}_{ik}\|$, and omitting constants of proportionality which are not necessary for sampling. We recognize $\lambda_{G_i}\, e^{-\lambda d}$ as the exponential density for the dispersal distance d. The factor of $\left(\frac{1}{d}\right)$ results from the Jacobian term in the change-of-variables between polar and Cartesian coordinates. From an implementation standpoint, these density calculations take place on a logarithmic scale. This technique can be similarly applied when using other distributions for dispersal distance, by substituting in the density of the alternate dispersal distribution.

2.2.3 Local Trap Calculations

During MCMC sampling, the capture hazard rate h_{ijkr} and associated likelihood terms are calculated for all traps, regardless of an individual's current AC. This is inefficient, since when $\boldsymbol{s}_{i,k}$ is "far" from a trap r, then h_{ijkr} will be extremely low, and a capture in trap r is exceedingly unlikely. Its contribution to $h_{ijk*} = \sum_{r=1}^{R} h_{ijkr}$ is negligible, as is the probability of capture in trap r. The original BUGS modeling language lacks the ability to conditionally disable calculations, and hence all the capture hazard rates must always be computed.

We introduce logic such that h_{ijkr} is only calculated for traps within a distance $d_{\min}$ of the individual's AC. For traps located further, we assign h_{ijkr} a small positive value. This will not affect the sum h_{ijk*}, but still allows for a non-zero probability of capture. Here, we let $h_{ijkr} = 10^{-14}$ for traps outside a radius $d_{\min} = 40$ from each individual's AC.

We introduce a discretized grid over the study area, and pre-compute indices of traps

within a radius $d_{\min}$ from each grid cell. Using this, a custom `nimble` function returns the indices of "local traps" nearest to any $\boldsymbol{s}_{i,k}$, and subsequently calculates hazard rates only for the "local" traps. This is similar to the local trap calculations used in the previous example, but implementations are different on account of the discretized habitat mask used there.

2.3 MCMC Efficiency

We define MCMC efficiency as the number of effectively independent posterior samples produced per second of MCMC runtime (excluding upfront time of model building and compilation). Distinct model parameters will typically mix at different rates, thus having distinct posterior effective sample sizes (ESS), and therefore a distinct measure of efficiency. In addition to presenting the MCMC runtimes and MCMC efficiency of all model parameters, we also summarize performance using the minimum and mean efficiencies among all model parameters. This definition of efficiency captures the tradeoff between quality of mixing and computational speed. Some algorithms may mix slowly (producing a low ESS) but execute sufficiently fast that they achieve high efficiency. Other algorithms may mix quickly (producing a high ESS) but require significantly longer execution time and thus achieve low efficiency.

3 Results

Here we describe the performance resulting from each formulation or sampling strategy of the Wolverine and Vole example models. All algorithm runtimes, ESS estimates, and MCMC efficiencies reflect independent chains of 10,000 posterior samples. We do not present the posterior inferences (*e.g.,* posterior mean, median, etc.), as they are qualitatively identical to the original published analyses.

3.1 Wolverine Model

We assess performance of the Wolverine model using total population size (N), probability of detection (p_0), and scale factor (σ). Results for the four stages of iterative improvement described in Section 2.1 will be denoted as Nimble1 (vectorization), Nimble2 (joint sampling), Nimble3 (evaluating local detectors), and Nimble4 (skipping unnecessary calculations).

As in Milleret et al. (2019), the JAGS model was unable to complete, crashing after 30 days. Transitioning to `nimble` considerably reduced memory usage and runtime, as we fit the Nimble1 model in 26 hours. ESS values were in the range of 100 to 200 for all parameters, indicating high posterior auto-correlation. In combination with the long runtime, this produced MCMC efficiencies on the order of 10^{-3} for all parameters. The addition of joint sampling in the Nimble2 version decreased runtime to 20 hours. Parameter ESS values were similar to the Nimble1 model, giving a small improvement in efficiency.

We observed major improvement in the Nimble3 version, using the local trap evaluations and a sparse representation of the observation matrix. MCMC runtime reduced to 30 minutes, by a factor of 40 relative to the Nimble2 model. As we expect, ESS values were unchanged, and the resulting MCMC efficiencies were in the range of 0.1 to 0.3 (Figure 1).

The Nimble4 version, disabling unnecessary model calculations, reduced MCMC runtime by an additional factor of two, down to 16 minutes. Accordingly, MCMC efficiencies increased by nearly a factor of two. Relative to the initial Nimble1 formulation, we have achieved increases in both the minimum and mean efficiencies of 100-fold. Concretely, while it was not even possible to fit the original version of this model using JAGS, the initial Nimble1 formulation would require 3.5 days to generate 1,000 ESS for all parameters, and the final Nimble4 model can accomplish the same in 51 minutes.

Figure 2 presents the minimum and mean efficiencies across all model parameters for each formulation of the Wolverine model, and all results for the Wolverine example appear in Table 4. An executable version of the Nimble4 Wolverine model is available at the web-appendix `http://danielturek.github.io/public/scr/wolverine_example.html`.

3.2 Vole Robust-Design Model

The Vole model contains a total of 11 hyper-parameters, which we use to assess MCMC efficiency. Results for the three stages of iterative improvement described in Section 2.2 will be denoted as Nimble1 (marginalization), Nimble2 (customizing dispersal distribution), and Nimble3 (evaluating local detectors). The original formulation of the model, running in JAGS, required over 4.5 hours to generate 10,000 posterior samples, and resulted in a minimum MCMC efficiency of 0.002, and a mean efficiency of 0.04.

The Nimble1 version introduced joint sampling of correlated parameters, and a custom likelihood to remove the z_{ij} latent states. This reduced the total model size from 4,460 nodes down to 3,562, while the number of unobserved nodes undergoing MCMC sampling was reduced from 1,437 down to 1,067. This model yielded an MCMC runtime of 15 minutes. ESS values were higher than those of JAGS, particularly for the jointly-sampled σ_i and κ_i parameters. MCMC efficiency was therefore higher for all parameters (Figure 3), while the average efficiency increased by a factor of 7.5 relative to JAGS.

The Nimble2 model introduced a custom bivariate dispersal distribution for individual ACs. This reduced the total model size from 3,562 nodes to 2,452, and the number of nodes for MCMC sampling from 1,067 to 697. Runtime decreased by a factor of two, to seven minutes, and all parameter MCMC efficiencies increased.

Using local trap calculations in the Nimble3 model reduced MCMC runtime further, to five minutes. Overall, relative to the initial analysis appearing in Ergon and Gardner (2014), these strategies reduced MCMC runtime by more than a factor of 50, and increased both minimum and mean MCMC efficiencies 25-fold. Concretely, the original model fitted in JAGS would require over seven days to produce 1,000 ESS for all parameters, whereas the Nimble3 formulation requires less than 6 hours to accomplish the same.

Figure 4 presents the minimum and mean efficiencies across all model parameters for each formulation of the Vole model, and all results for the Vole example appear in Table 4.

4 Discussion

SCR models are now commonplace given the abundance of geolocated ecological data, but remain computationally challenging. Indeed, large numbers of individuals, expansive study areas, and/or movement between seasons can render some problems intractable, without employing custom approaches.

The techniques demonstrated here produce posterior results identical (within Monte Carlo error) to the original versions, with the exception of local trap evaluations. This attributed a small trap hazard rate (or probability of detection) outside a radius $d_{\min}$ from each individual AC. The choice of $d_{\min}$ is important: large values will produce identical inferences but offer no computational gain, while small values offer a large computational gain but may introduce bias. The choice of $d_{\min}$ is subjective, and will require expert opinion (or trial runs) to determine an appropriate value. Smaller values of $d_{\min}$ may be used for exploratory analyses, but a conservative higher value should be used to minimize any biases in the final inference.

We are aware that conditioning on the primary occasion of first capture, as in the Vole example, may induce bias into parameter estimates (Efford and Schofield, 2019, Appendix E). Simulations in Ergon and Gardner (2014) suggest minimal bias in mortality estimates, although the scale parameter in the observation model may be inflated. Thus, care should be taken when applying this model to other data. That said, our purpose has been to investigate efficiency of estimation methods rather than statistical properties (such as bias or goodness of fit) of paticular models. Indeed, the ability to perform inference more efficiently will support a deeper exploration of alternative models structures.

Many software packages are available for fitting SCR models, making these analyses faster and more accessible to practitioners (*e.g.*, `secr`, or `oSCR`, among others). The prevalence of specialized software underscores the complex nature of SCR problems, and furthermore that no single software package could be general enough to approach all SCR problems. `nimble` does not attempt to provide "canned" algorithms for SCR, or any other particular

application, but rather a flexible programming environment suitable for customized (and highly efficient) analysis of complex data.

We have made use of the `nimble` software package for `R`, to demonstrate techniques for improving the performance of SCR model fitting using MCMC. The techniques demonstrated are not exhaustive, but rather suggest the potential performance gains made possible using `nimble`, where we observed between one and two orders of magnitude improvement. These approaches can provide significant computational gain, permitting large-scale spatial and temporal analyses to support major conservation and management decisions, and the ability to fit increasingly complex models to large datasets. More broadly, similar techniques are also applicable to the analysis of general spatially-indexed hierarchical model structures.

Acknowledgements

We would like to thank Pierre Dupont and Richard Bischof for their help with the Wolverine example analysis. This work was partly funded by the Norwegian Environment Agency (Miljødirektoratet), the Swedish Environmental Protection Agency (Naturvårdsverket) and the Research Council of Norway (NFR 286886).

References

Borchers, David L and MG Efford (2008). "Spatially explicit maximum likelihood methods for capture–recapture studies". *Biometrics* 64.2, pp. 377–385.

Brooks, Steve, Andrew Gelman, Galin Jones, and Xiao-Li Meng (2011). *Handbook of Markov Chain Monte Carlo*. en. bibtex: Brooks2011. CRC Press.

de Valpine, Perry, Daniel Turek, Christopher J Paciorek, Clifford Anderson-Bergman, Duncan Temple Lang, and Rastislav Bodik (2017). "Programming with models: writing statistical algorithms for general model structures with NIMBLE". *J. Comput. Graph. Stat.* 26.2, pp. 403–413.

Efford, Murray (2004). "Density estimation in live-trapping studies". *Oikos* 106.3, pp. 598–610.

Efford, Murray G and Matthew R Schofield (2019). "A spatial open-population capture–recapture model". *Biometrics*.

Ergon, T. and X. Lambin (2013). *Data from: Separating mortality and emigration: Modelling space use, dispersal and survival with robust-design spatial-capture-recapture data*. Tech. rep. Dryad Digital Repository. URL http://dx.doi.org/10.5061/dryad.r17n5.

Ergon, T., rnulf Borgan, C. N. Nater, and Y. Vindenes (2018). "The utility of mortality hazard rates in population analyses". *Methods in Ecology and Evolution*.

Ergon, Torbjrn and Beth Gardner (2014). "Separating mortality and emigration: modelling space use, dispersal and survival with robust-design spatial capturerecapture data". en. *Methods in Ecology and Evolution* 5.12, pp. 1327–1336.

Gardner, Beth, Rahel Sollmann, N. Samba Kumar, Devcharan Jathanna, and K. Ullas Karanth (2018). "State space and movement specification in open population spatial capture-recapture models". en. *Ecology and Evolution* 8.20, pp. 10336–10344.

Lunn, David, David Spiegelhalter, Andrew Thomas, and Nicky Best (2009). "The BUGS project: Evolution, critique and future directions". *Statistics in Medicine* 28.25.

Milleret, C, P Dupont, C Bonenfant, H Brseth, Flagstad, C Sutherland, and R Bischof (2018a). *Data from: A local evaluation of the individual state-space to scale up Bayesian spatial capture recapture*.

Milleret, Cyril, Pierre Dupont, Henrik Brseth, Jonas Kindberg, J. Andrew Royle, and Richard Bischof (2018b). "Using partial aggregation in spatial capture recapture". en. *Methods in Ecology and Evolution* 9.8. Ed. by Nigel Yoccoz, pp. 1896–1907.

Milleret, Cyril, Pierre Dupont, Christophe Bonenfant, Henrik Brøseth, Øystein Flagstad, Chris Sutherland, and Richard Bischof (2019). "A local evaluation of the individual state-space to scale up Bayesian spatial capture-recapture". *Ecology and Evolution* 9.1, pp. 352–363.

Plummer, Martyn (2003). "JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling". *Proceedings of the 3rd international workshop on distributed statistical computing*. Vol. 124. bibtex: Plummer2003. Vienna, p. 125.

Royle, J Andrew (2009). "Analysis of capture–recapture models with individual covariates using data augmentation". *Biometrics* 65.1, pp. 267–274.

Royle, J Andrew, Robert M Dorazio, and William A Link (2007). "Analysis of multinomial models with unknown index using data augmentation". *Journal of Computational and Graphical Statistics* 16.1, pp. 67–85.

Stan Development Team (2014). "Stan: A C++ Library for Probability and Sampling, Version 2.5.0".

Turek, Daniel, Perry de Valpine, and Christopher J Paciorek (2016). "Efficient Markov chain Monte Carlo sampling for hierarchical hidden Markov models". *Environmental and ecological statistics* 23.4, pp. 549–564.

Turek, Daniel, Perry de Valpine, Christopher J Paciorek, and Clifford Anderson-Bergman (2017). "Automated parameter blocking for efficient Markov chain Monte Carlo sampling". *Bayesian Analysis* 12.2, pp. 465–490.

	Parameter		
	N	p_0	σ
Nimble1	0.005	0.004	0.003
Nimble2	0.006	0.005	0.003
Nimble3	0.284	0.247	0.192
Nimble4	0.390	0.394	0.362

Table 1: MCMC efficiency values for the Wolverine example, for all parameters and model formulations. Results are averaged over three independent chains.

	Parameter										
	ϕ_1	ϕ_2	β_1	β_2	λ_0	λ_1	λ_2	κ_1	κ_2	σ_1	σ_2
JAGS	0.23	0.05	0.11	0.003	0.002	0.02	0.03	0.002	0.004	0.002	0.003
Nimble1	2.21	0.99	0.74	0.033	0.017	0.16	0.21	0.013	0.065	0.010	0.050
Nimble2	4.49	1.09	1.55	0.054	0.034	0.28	0.25	0.032	0.097	0.022	0.091
Nimble3	5.70	1.43	2.24	0.096	0.062	0.39	0.28	0.055	0.162	0.048	0.140

Table 2: MCMC efficiency values for the Vole example, for all parameters and model formulations. Results are averaged over five independent chains.

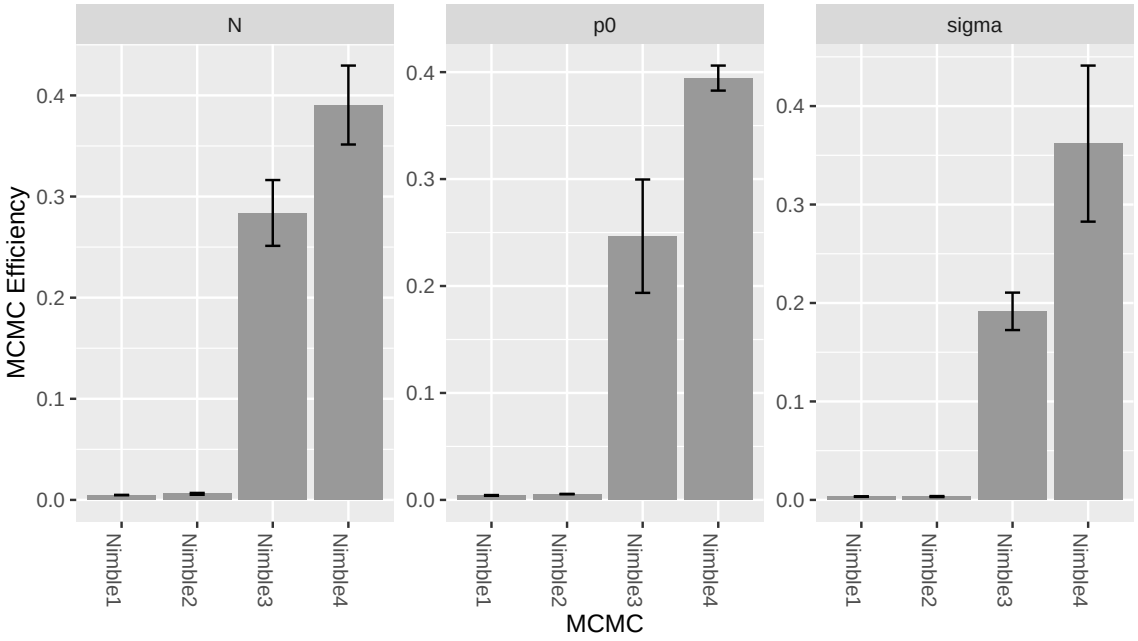

Figure 1: MCMC efficiency for the Wolverine example, for all parameters and model formulations. Efficiency values are averaged over three independent chains, error bars showing standard deviation.

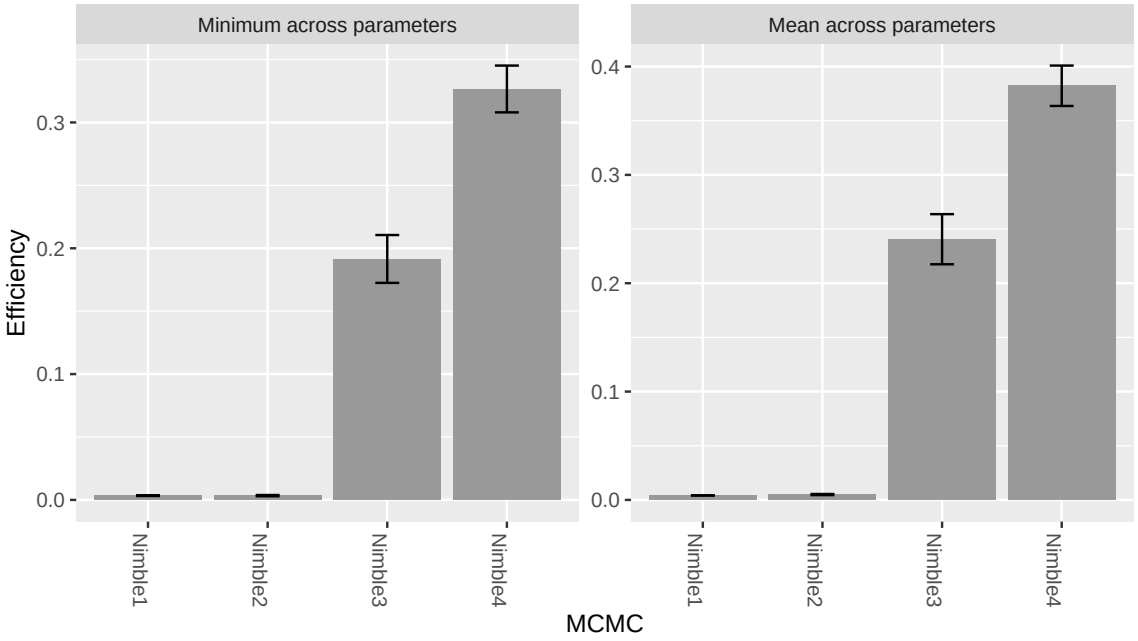

Figure 2: Minimum and mean MCMC efficiency among the three model parameters for the Wolverine example. Values are averaged over three independent chains, error bars showing standard deviation.

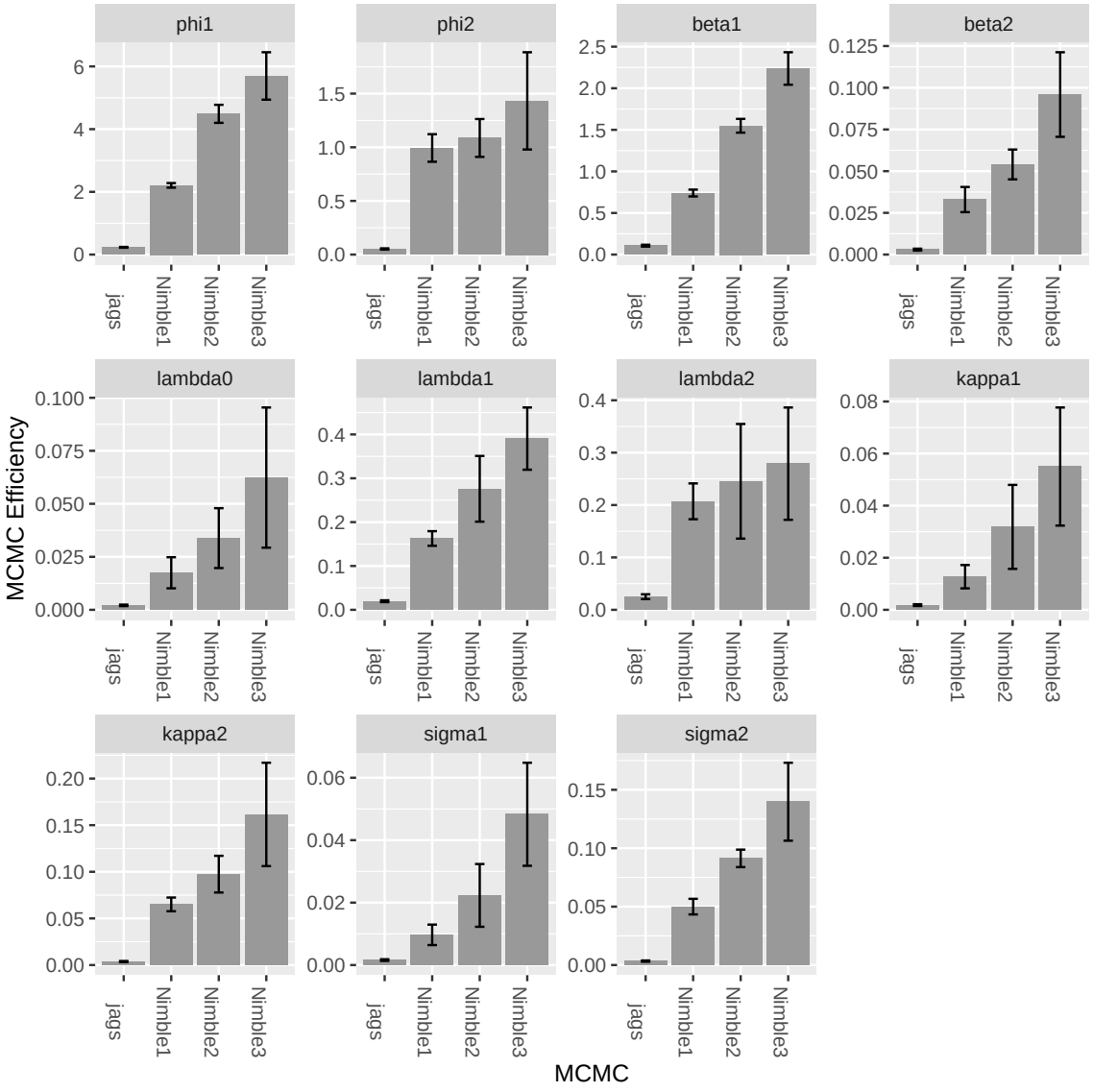

Figure 3: MCMC efficiency for the Vole example, for all parameters and model formulations. Efficiency values are averaged over five independent chains, error bars showing standard deviation.

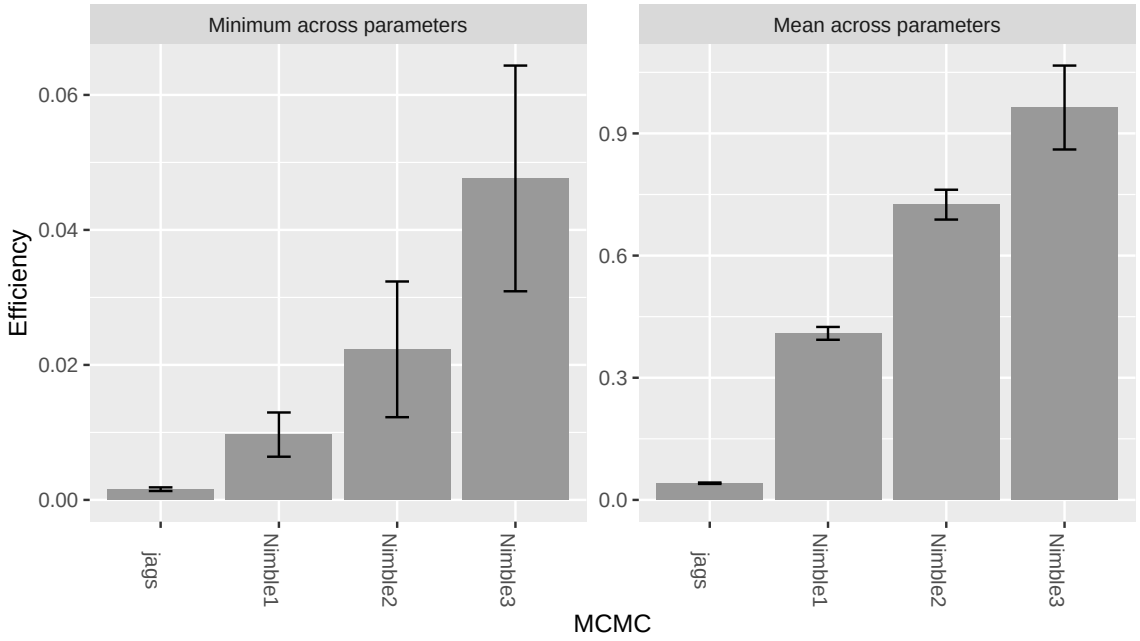

Figure 4: Minimum and mean MCMC efficiency among the eleven model parameters for the Vole example. Values are averaged over five independent chains, error bars showing standard deviation.

A Wolverine Example: Model Code

A.1 Vectorized Computations Code

```
dHabitat <- nimbleFunction(
    run = function(x = double(0), sxy = double(1), lower = double(1),
                   upper = double(1), habitat = double(2), log = double()) {
        if(sxy[1] < lower[1]) return(-Inf)   # x-coordinates
        if(sxy[1] > upper[1]) return(-Inf)   # x-coordinates
        if(sxy[2] < lower[2]) return(-Inf)   # y-coordinates
        if(sxy[2] > upper[2]) return(-Inf)   # y-coordinates
        returnType(double())
        if(habitat[trunc(sxy[2])+1, trunc(sxy[1])+1] == 0) return(-Inf) else return(0)
    }
)

dBernoulliVector <- nimbleFunction(
    run = function(x = double(1), prob = double(1),
                   trials = double(1), log = integer(0)) {
        returnType(double(0))
        logProb <- sum(dbinom(x, prob = prob, size = trials, log = TRUE))
        return(logProb)
    }
)

code <- nimbleCode({
    for(i in 1:n.individuals) {
        sxy[i,1] ~ dunif(0, x.max)
        sxy[i,2] ~ dunif(0, y.max)
        ones[i] ~ dHabitat(sxy = sxy[i,1:2], lower = lowerCoords[1:2],
                    upper = upperCoords[1:2], habitat = habitat.mx[1:y.max,1:x.max])
    }
    psi ~ dunif(0,1)
    for (i in 1:n.individuals) {
        z[i] ~ dbern(psi)
    }
    sigma ~ dunif(0, 50)
    alpha <- -1 / (2 * sigma^2)
    p0 ~ dunif(0, 1)
    for(i in 1:n.individuals) {
        d2[i, 1:n.detectors] <- (sxy[i,1] - detector.xy[1:n.detectors,1])^2 +
                                (sxy[i,2] - detector.xy[1:n.detectors,2])^2
        p[i, 1:n.detectors] <- p0 * exp(alpha * d2[i,1:n.detectors])
        y[i, 1:n.detectors] ~ dBernoulliVector(prob = p[i,1:n.detectors]*z[i],
                                               trials =  trials[1:n.detectors])
```

```
453       }
454       N <- sum(z[1:n.individuals])
455   })
```

A.2 Local Detector Evaluations and Sparse Observation Matrix Code

```
dBernoulliVector2 <- nimbleFunction(
                    run = function(x = double(1), pZero = double(0),
                                   sxy = double(1), sigma = double(0),
                                   nbDetections = double(0), yDets = double(1),
                                   detector.xy = double(2), trials = double(1),
                                   detectorIndex = double(2), nDetectorsLESS = double(1),
                                   ResizeFactor = double(0, default = 1),
                                   maxNBDets = double(0), habitatID = double(2),
                                   log = integer(0, default = 0)){
  returnType(double(0))
  nDetectors <- length(trials)
  sxyID <- habitatID[trunc(sxy[2]/ResizeFactor)+1, trunc(sxy[1]/ResizeFactor)+1]
  index <- detectorIndex[sxyID,1:nDetectorsLESS[sxyID]]
  n.detectors <- length(index)
  y <- nimNumeric(length = nDetectors, value = 0, init = TRUE)
  if(nbDetections > 0){
    for(r in 1:nbDetections){
      y[yDets[r]] <- x[r]
      if(sum(yDets[r]==index)==0){
        if(log == 0) return(0.0)
        else return(-Inf)
      }
    }
  }
  alpha <- -1.0 / (2.0 * sigma * sigma)
  logProb <- 0.0
  count <- 1
  index1 <- c(index,0)
  for(r in 1:nDetectors){
      if(index1[count] == r){
        d2 <- pow(detector.xy[r,1] - sxy[1], 2) + pow(detector.xy[r,2] - sxy[2], 2)
        p <- pZero * exp(alpha * d2)
        logProb <- logProb + dbinom(y[r], prob = p, size = trials[r], log = TRUE)
        count <- count + 1
      }
  }
  if(log)return(logProb)
  return(exp(logProb))
})

code <- nimbleCode({
    for(i in 1:n.individuals) {
```

```
        sxy[i,1] ~ dunif(0, x.max)
        sxy[i,2] ~ dunif(0, y.max)
        ones[i] ~ dHabitat(sxy = sxy[i,1:2], lower = lowerCoords[1:2],
                           upper = upperCoords[1:2], habitat = habitat.mx[1:y.max,1:x.max])
    }
    psi ~ dunif(0,1)
    for (i in 1:n.individuals) {
        z[i] ~ dbern(psi)
    }
    sigma ~ dunif(0, 50)
    p0 ~ dunif(0, 1)
    for(i in 1:n.individuals) {
        y[i,1:n.detectors] ~
            dBernoulliVector2(pZero = p0*z[i], sxy = sxy[i,1:2], sigma = sigma,
                              nbDetections[i], yDets = yDets[i,1:nMaxDetectors],
                              detector.xy =  detector.xy[1:n.detectors,1:2],
                              trials = trials[1:n.detectors],
                              detectorIndex = detectorIndex[1:n.cells,1:maxNBDets],
                              nDetectorsLESS = nDetectorsLESS[1:n.cells],
                              ResizeFactor = ResizeFactor, maxNBDets = maxNBDets,
                              habitatID = habitatIDDet[1:y.maxDet,1:x.maxDet])
    }
    N <- sum(z[1:n.individuals])
})
```

A.3 Skip Local Calculations Code

```
dBernoulliVector3 <- nimbleFunction(
                    run = function(x = double(1), pZero = double(0),
                                   sxy = double(1), sigma = double(0),
                                   nbDetections = double(0), yDets = double(1),
                                   detector.xy = double(2), trials = double(1),
                                   detectorIndex = double(2), nDetectorsLESS = double(1),
                                   ResizeFactor = double(0, default = 1),
                                   maxNBDets = double(0), habitatID = double(2),
                                   indicator = double(0, default = 1.0),
                                   log = integer(0, default = 0)) {
  returnType(double(0))
  nDetectors <- length(trials)
  if(indicator == 0){
    if(nbDetections == 0){
      if(log == 0) return(1.0)
      else return(0.0)
    } else {
      if(log == 0) return(0.0)
      else return(-Inf)
    }
  }
  sxyID <- habitatID[trunc(sxy[2]/ResizeFactor)+1, trunc(sxy[1]/ResizeFactor)+1]
  index <- detectorIndex[sxyID,1:nDetectorsLESS[sxyID]]
  n.detectors <- length(index)
  y <- nimNumeric(length = nDetectors, value = 0, init = TRUE)
  if(nbDetections > 0){
    for(r in 1:nbDetections){
      y[yDets[r]] <- x[r]
      if(sum(yDets[r]==index)==0){
        if(log == 0) return(0.0)
        else return(-Inf)
      }
    }
  }
  alpha <- -1.0 / (2.0 * sigma * sigma)
  logProb <- 0.0
  count <- 1
  index1 <- c(index,0)
  for(r in 1:nDetectors){
      if(index1[count] == r){
        d2 <- pow(detector.xy[r,1] - sxy[1], 2) + pow(detector.xy[r,2] - sxy[2], 2)
        p <- pZero * exp(alpha * d2)
        logProb <- logProb + dbinom(y[r], prob = p, size = trials[r], log = TRUE)
        count <- count + 1
```

```
      }
  }
  if(log)return(logProb)
  return(exp(logProb))
})

code <- nimbleCode({
    for(i in 1:n.individuals) {
        sxy[i,1] ~ dunif(0, x.max)
        sxy[i,2] ~ dunif(0, y.max)
        ones[i] ~ dHabitat(sxy = sxy[i,1:2], lower = lowerCoords[1:2],
                    upper = upperCoords[1:2], habitat = habitat.mx[1:y.max,1:x.max])
    }
    psi ~ dunif(0,1)
    for (i in 1:n.individuals) {
        z[i] ~ dbern(psi)
    }
    sigma ~ dunif(0, 50)
    p0 ~ dunif(0, 1)
    for(i in 1:n.individuals) {
    y[i, 1:nMaxDetectors] ~ dBernoulliVector3(pZero = p0, sxy = sxy[i,1:2],
            sigma = sigma, nbDetections[i], yDets = yDets[i,1:nMaxDetectors],
            detector.xy =  detector.xy[1:n.detectors,1:2],
            trials = trials[1:n.detectors],
            detectorIndex = detectorIndex[1:n.cells,1:maxNBDets],
            nDetectorsLESS = nDetectorsLESS[1:n.cells],
            ResizeFactor = ResizeFactor, maxNBDets = maxNBDets,
            habitatID = habitatIDDet[1:y.maxDet,1:x.maxDet],
             indicator = z[i])
    }
    N <- sum(z[1:n.individuals])
})
```

B Vole Example: Model Code

B.1 JAGS Code

```
code <- nimbleCode({
    for(sex in 1:2){
        kappa[sex] ~ dunif(0,50)
        sigma[sex] ~ dunif(0.1,20)
    }
    for(sex in 1:2){
        for(TOD in 1:2){
            lambda[TOD, sex] <- lambda0 * pow(beta[1],(TOD-1)) * pow(beta[2],(sex-1))
        }
    }
    PL ~ dunif(0.01,0.99)
    lambda0 <- -log(1-PL)
    beta[1] ~ dunif(0.1,10)
    beta[2] ~ dunif(0.1,10)
    for(sex in 1:2){
        Phi[sex] ~ dunif(0,1)
        for(k in 1:(n.prim-1)){
            phi[sex,k] <- pow(Phi[sex], dt[k])
        }
    }
    for(sex in 1:2){
        dmean[sex] ~ dunif(0,100)
        dlambda[sex] <- 1/dmean[sex]
    }
    for(i in 1:N[1]){
        z[i,first[i]] ~ dbern(1)
        S[i,1,first[i]] ~ dunif(xlow[i], xupp[i]) # Prior for the first x coordinate
        S[i,2,first[i]] ~ dunif(ylow[i], yupp[i]) # Prior for the first y coordinate
        g[i,first[i],1] <- 0
        for(r in 1:R){ # trap
            D[i,r,first[i]] <- sqrt(pow(S[i,1,first[i]]-X[r,1],2) +
                                   pow(S[i,2,first[i]]-X[r,2],2))
            g[i,first[i],r+1] <- exp(-pow(D[i,r,first[i]]/sigma[gr[i]], kappa[gr[i]]))
        }
        G[i,first[i]] <- sum(g[i,first[i],1:(R+1)]) # Total trap exposure
        for(j in 1:J[i,first[i]]){
            P[i,j,first[i]] <- 1 - exp(-lambda[tod[first[i],j],gr[i]]*G[i,first[i]])
            PPII[i,first[i],j] <- step(H[i,j,first[i]]-2) *
                                  (g[i,first[i],H[i,j,first[i]]] /
                                  (G[i,first[i]]+ 0.0001)) *
                                  P[i,j,first[i]] +
```

```
                                (1-step(H[i,j,first[i]]-2)) * (1-P[i,j,first[i]])
            Ones[i,j,first[i]] ~ dbern(PPII[i,first[i],j])
        }
    }
    for(i in (N[1]+1):N[2]){
        z[i,first[i]] ~ dbern(1)
        S[i,1,first[i]] ~ dunif(xlow[i], xupp[i]) # Prior for the first x coordinate
        S[i,2,first[i]] ~ dunif(ylow[i], yupp[i]) # Prior for the first y coordinate
        ## First primary session:
        g[i,first[i],1] <- 0
        for(r in 1:R){ # trap
            D[i,r,first[i]] <- sqrt(pow(S[i,1,first[i]]-X[r,1],2) +
                                    pow(S[i,2,first[i]]-X[r,2],2))
            g[i,first[i],r+1] <- exp(-pow(D[i,r,first[i]]/sigma[gr[i]], kappa[gr[i]]))
        }
        G[i,first[i]] <- sum(g[i,first[i],1:(R+1)]) # Total trap exposure
        for(j in 1:J[i,first[i]]){
            P[i,j,first[i]] <- 1 - exp(-lambda[tod[first[i],j],gr[i]]*G[i,first[i]])
            PPII[i,first[i],j] <- step(H[i,j,first[i]]-2) *
                                      (g[i,first[i],H[i,j,first[i]]] /
                                      (G[i,first[i]]+ 0.0001)) *
                                      P[i,j,first[i]] +
                                      (1-step(H[i,j,first[i]]-2))*(1-P[i,j,first[i]])
            Ones[i,j,first[i]] ~ dbern(PPII[i,first[i],j])
        }
        for(k in (first[i]+1):K[i]){ # primary session
            theta[i,k-1] ~ dunif(-3.141593,3.141593) # Prior for dispersal direction
            z[i,k] ~ dbern(Palive[i,k-1])
            Palive[i,k-1] <- z[i,k-1]*phi[gr[i],k-1] # Pr(alive in primary session k)
            d[i,k-1] ~ dexp(dlambda[gr[i]])
            S[i,1,k] <- S[i,1,k-1] + d[i,k-1]*cos(theta[i,k-1])
            S[i,2,k] <- S[i,2,k-1] + d[i,k-1]*sin(theta[i,k-1])
            g[i,k,1] <- 0
            for(r in 1:R){ # trap
                D[i,r,k] <- sqrt(pow(S[i,1,k]-X[r,1],2) + pow(S[i,2,k]-X[r,2],2))
                g[i,k,r+1] <- exp(-pow(D[i,r,k]/sigma[gr[i]], kappa[gr[i]]))
            }
            G[i,k] <- sum(g[i,k,1:(R+1)]) # Total trap exposure
            for(j in 1:J[i,k]){
                P[i,j,k] <- (1 - exp(-lambda[tod[k,j],gr[i]]*G[i,k]))*z[i,k]
                PPII[i,k,j] <- step(H[i,j,k]-2) *
                               (g[i,k,H[i,j,k]] /
                               (G[i,k] + 0.0001))*P[i,j,k] +
                               (1-step(H[i,j,k]-2))*(1-P[i,j,k])
                Ones[i,j,k] ~ dbern(PPII[i,k,j])
            }
        }
```

```
692         }
693     })
```

B.2 Joint Sampling and Marginalization Code

```
dLikelihood <- nimbleFunction(
    run = function(x = double(2), first = double(), last = double(),
                   J = double(1), lambda = double(1), tod = double(2),
                   g = double(2), G = double(1), z = double(1),
                   phi = double(1), log = double()) {
        pAlive <- 1
        pDead <- 0
        lp <- 0
        for(k in first:last) {
            if(z[k] == 1) {    # known to be alive
                if(k > first)           # survived
                    lp <- lp + log(phi[k-1])
                for(j in 1:J[k]) {
                    pNoCaptureGivenAlive <- exp(-lambda[tod[k,j]] * G[k])
                    if(x[j,k] == 1) {   # not captured
                        lp <- lp + log(pNoCaptureGivenAlive)
                    } else {            # captured
                        lp <- lp + log(1-pNoCaptureGivenAlive) +
                              log(g[k, x[j,k]-1]) - log(G[k])
                    }
                }
            } else {           # could be dead or alive
                pTheseNonSightings <- 1
                for(j in 1:J[k]) {
                    pNoCaptureGivenAlive <- exp(-lambda[tod[k,j]] * G[k])
                    pTheseNonSightings <- pTheseNonSightings * pNoCaptureGivenAlive
                }
                pAlive_new <- phi[k-1] * pAlive
                pDead_new <- (1-phi[k-1]) * pAlive + pDead
                L <- pAlive_new * pTheseNonSightings + pDead_new
                pAlive <- (pAlive_new * pTheseNonSightings) / L
                pDead <- pDead_new / L
                lp <- lp + log(L)
            }
        }
        returnType(double())
        if(log) return(lp) else return(exp(lp))
    }
)

code <- nimbleCode({
    PL ~ dunif(0.01, 0.99)
    lambda0 <- -log(1-PL)
    for(sex in 1:2) {
```

```
        kappa[sex] ~ dunif(0,   50)
        sigma[sex] ~ dunif(0.1, 20)
        beta[sex]  ~ dunif(0.1, 10)
        for(TOD in 1:2) {
            lambda[TOD, sex] <- lambda0 * beta[1]^(TOD-1) * beta[2]^(sex-1)
        }
        Phi[sex] ~ dunif(0, 1)
        for(k in 1:(nPrimary-1)) {
            phi[sex, k] <- Phi[sex]^dt[k]
        }
        dmean[sex] ~ dunif(0, 100)
        dlambda[sex] <- 1/dmean[sex]
    }
    for(i in 1:nInd) {
        S[i, 1, first[i]] ~ dunif(xlow[i], xupp[i])
        S[i, 2, first[i]] ~ dunif(ylow[i], yupp[i])
        for(k in first[i]:last[i]) {
            D[i, k, 1:R] <- sqrt((S[i, 1, k] - X[1:R, 1])^2 + (S[i, 2, k] - X[1:R, 2])^2)
            g[i, k, 1:R] <- exp(-(D[i, k, 1:R]/sigma[gr[i]])^kappa[gr[i]])
            G[i, k] <- sum(g[i, k, 1:R])
        }
        for(k in first[i]:(last[i]-1)) {
            theta[i, k] ~ dunif(-3.141593, 3.141593)   # dispersal direction
            d[i, k] ~ dexp(dlambda[gr[i]])
            S[i, 1, k+1] <- S[i, 1, k] + d[i, k] * cos(theta[i, k])
            S[i, 2, k+1] <- S[i, 2, k] + d[i, k] * sin(theta[i, k])
        }
        H[i, 1:nSecondary, 1:nPrimary] ~ dLikelihood(
            first = first[i], last = last[i], J = J[i,1:nPrimary],
            lambda = lambda[1:2,gr[i]], tod = tod[1:nPrimary,1:nSecondary],
            g = g[i,1:nPrimary,1:R], G = G[i,1:nPrimary],
            z = z[i,1:nPrimary], phi = phi[gr[i],1:(nPrimary-1)])
    }
})
```

B.3 Custom Dispersal Distribution Code

```
dDispersal <- nimbleFunction(
    run = function(x = double(1), S = double(1), lam = double(), log = double()) {
        dist <- sqrt(sum((x-S)^2))
        lp <- dexp(dist, rate = lam, log = TRUE) - log(dist)
        returnType(double())
        if(log) return(lp) else return(exp(lp))
    }
)

code <- nimbleCode({
    PL ~ dunif(0.01, 0.99)
    lambda0 <- -log(1-PL)
    for(sex in 1:2) {
        kappa[sex] ~ dunif(0,   50)
        sigma[sex] ~ dunif(0.1, 20)
        beta[sex]  ~ dunif(0.1, 10)
        for(TOD in 1:2) {
            lambda[TOD, sex] <- lambda0 * beta[1]^(TOD-1) * beta[2]^(sex-1)
        }
        Phi[sex] ~ dunif(0, 1)
        for(k in 1:(nPrimary-1)) {
            phi[sex, k] <- Phi[sex]^dt[k]
        }
        dmean[sex] ~ dunif(0, 100)
        dlambda[sex] <- 1/dmean[sex]
    }
    for(i in 1:nInd) {
        S[i, 1, first[i]] ~ dunif(xlow[i], xupp[i])
        S[i, 2, first[i]] ~ dunif(ylow[i], yupp[i])
        for(k in first[i]:last[i]) {
            D[i, k, 1:R] <- sqrt((S[i, 1, k] - X[1:R, 1])^2 + (S[i, 2, k] - X[1:R, 2])^2)
            g[i, k, 1:R] <- exp(-(D[i, k, 1:R]/sigma[gr[i]])^kappa[gr[i]])
            G[i, k] <- sum(g[i, k, 1:R])
        }
        for(k in first[i]:(last[i]-1)) {
            S[i, 1:2, k+1] ~ dDispersal(S[i, 1:2, k], dlambda[gr[i]])
        }
        H[i, 1:nSecondary, 1:nPrimary] ~ dLikelihood(
            first = first[i], last = last[i], J = J[i,1:nPrimary],
            lambda = lambda[1:2,gr[i]], tod = tod[1:nPrimary,1:nSecondary],
            g = g[i,1:nPrimary,1:R], G = G[i,1:nPrimary],
            z = z[i,1:nPrimary], phi = phi[gr[i],1:(nPrimary-1)])
    }
})
```

B.4 Local Trap Calculations Code

```
makeGrid <- function(xmin=0, ymin=0, xmax, ymax, resolution=1, buffer=0) {
    makeVals <- function(min, max, buf, res) {
        unique(c(rev(seq(min, min-buf, by = -res)), seq(min, max+buf, by = res)))
    }
    xvals <- makeVals(xmin, xmax, buffer, resolution)
    yvals <- makeVals(ymin, ymax, buffer, resolution)
    grid <- expand.grid(xvals, yvals)
    colnames(grid) <- c('x', 'y')
    ## unique ids:
    mult <- diff(range(grid$y/resolution)) + 1
    ids <- grid$x/resolution * mult + grid$y/resolution
    offset <- 1 - min(ids)
    require(nimble)
    makeIDdef <- substitute(
        nimbleFunction(
            run = function(xy = double(1)) {
                id <- xy[1]/RES * MULT + xy[2]/RES + OFFSET
                returnType(double())
                return(id)
            }
        ),
        list(RES = resolution,
             MULT = mult,
             OFFSET = offset))
    makeID <- eval(makeIDdef)
    ids2 <- apply(grid, 1, function(xy) makeID(xy))
    sorted <- sort(ids2, index.return = TRUE)
    gridReordered <- grid[sorted$ix, ]
    gridReordered$id <- sorted$x
    return(list(grid = gridReordered, makeID = makeID))
}

xr <- range(constants$X[, 1])
yr <- range(constants$X[, 2])
buffer <- 40
exposureRadius <- 40
resolution <- 7
makeGridReturn <- makeGrid(xmin=xr[1], xmax = xr[2],
                           ymin=yr[1], ymax = yr[2],
                           buffer = buffer,
                           resolution = resolution)
grid <- makeGridReturn$grid
makeID <- makeGridReturn$makeID

```

```
findLocalTraps <- function(grid, traps, exposureRadius) {
    trtrapsBool <- apply(grid, 1, function(row) {
        apply(traps, 1, function(tp) {
            sqrt(sum((row[1:2]-tp)^2)) <= exposureRadius
        })
    })
    trapsBool <- t(trtrapsBool)
    trapsInd <- apply(trapsBool, 1, which)
    numsTraps <- sapply(trapsInd, length)
    localTraps <- array(as.numeric(NA), c(dim(grid)[1], max(numsTraps)+1))
    for(i in seq_along(trapsInd)) {
        n <- numsTraps[i]
        localTraps[i,1] <- n
        if(n > 0)    localTraps[i, 2:(n+1)] <- trapsInd[[i]]
    }
    localTraps
}

## n = localTraps[i,1] gives the number of local traps
## localTraps[i, 2:(n+1)] gives the indices of the local traps
localTraps <- findLocalTraps(grid, constants$X, exposureRadius)

getNumLocalTraps6 <- nimbleFunction(
    run = function(idarg = double(), localTrapNumbers = double(1), LTD1arg = double()) {
        if(idarg < 1)       {   return(0)   }
        if(idarg > LTD1arg) {   return(0)   }
        n <- localTrapNumbers[idarg]
        returnType(double())
        return(n)
    }
)

getLocalTrapIndices6 <- nimbleFunction(
    run = function(MAXNUM = double(), localTraps = double(2),
                   n = double(), idarg = double()) {
        indices <- numeric(MAXNUM, 0)
        if(n > 0) {
            indices[1:n] <- localTraps[idarg, 2:(n+1)]
        }
        returnType(double(1))
        return(indices)
    }
)

calcLocalTrapDists6 <- nimbleFunction(
    run = function(MAXNUM = double(), n = double(),
                   localTrapInd = double(1), S = double(1), X = double(2)) {
```

```
        Ds <- numeric(MAXNUM, 0)
        if(n > 0) {
            Ds[1:n] <- sqrt((S[1] - X[localTrapInd[1:n],1])^2 +
                            (S[2] - X[localTrapInd[1:n],2])^2)
        }
        returnType(double(1))
        return(Ds)
    }
)

calcLocalTrapExposure6 <- nimbleFunction(
    run = function(R = double(), n = double(), Ds = double(1),
                   localTrapInd = double(1), sigma = double(), kappa = double()) {
        g <- numeric(R, 0.00000000000001)      ## small value
        if(n > 0) {
            g[localTrapInd[1:n]] <- exp(-(Ds[1:n]/sigma)^kappa)
        }
        returnType(double(1))
        return(g)
    }
)

code <- nimbleCode({
    PL ~ dunif(0.01, 0.99)
    lambda0 <- -log(1-PL)
    for(sex in 1:2) {
        kappa[sex] ~ dunif(0,   50)
        sigma[sex] ~ dunif(0.1, 20)
        beta[sex]  ~ dunif(0.1, 10)
        for(TOD in 1:2) {
            lambda[TOD, sex] <- lambda0 * beta[1]^(TOD-1) * beta[2]^(sex-1)
        }
        Phi[sex] ~ dunif(0, 1)
        for(k in 1:(nPrimary-1)) {
            phi[sex, k] <- Phi[sex]^dt[k]
        }
        dmean[sex] ~ dunif(0, 100)
        dlambda[sex] <- 1/dmean[sex]
    }
    for(i in 1:nInd) {
        S[i, 1, first[i]] ~ dunif(xlow[i], xupp[i])
        S[i, 2, first[i]] ~ dunif(ylow[i], yupp[i])
        Sdiscrete[i, 1, first[i]] <- round(S[i, 1, first[i]]/7) * 7   ## resolution = 7
        Sdiscrete[i, 2, first[i]] <- round(S[i, 2, first[i]]/7) * 7   ## resolution = 7
        for(k in first[i]:last[i]) {
            id[i, k] <- makeID(Sdiscrete[i,1:2,k])
            nLocalTraps[i, k] <-
```

```
                getNumLocalTraps6(idarg=id[i,k], localTrapNumbers =
                                  localTraps[1:LTD1,1], LTD1arg = LTD1)
            localTrapIndices[i, k, 1:maxTraps] <-
                getLocalTrapIndices6(MAXNUM = maxTraps,
                                     localTraps = localTraps[1:LTD1,1:LTD2],
                                     n = nLocalTraps[i, k], idarg = id[i,k])
            Ds[i, k, 1:maxTraps] <-
                calcLocalTrapDists6(MAXNUM = maxTraps, n = nLocalTraps[i,k],
                                    localTrapInd = localTrapIndices[i,k,1:maxTraps],
                                    S = S[i,1:2,k], X = X[1:R,1:2])
            g[i, k, 1:R] <- calcLocalTrapExposure6(
                R = R, n = nLocalTraps[i,k], Ds = Ds[i,k,1:maxTraps],
                localTrapInd = localTrapIndices[i,k,1:maxTraps],
                sigma = sigma[gr[i]], kappa = kappa[gr[i]])
            G[i, k] <- sum(g[i, k, 1:R])
        }
        for(k in first[i]:(last[i]-1)) {
            S[i, 1:2, k+1] ~ dDispersal(S[i, 1:2, k], dlambda[gr[i]])
            Sdiscrete[i, 1:2, k+1] <- round(S[i, 1:2, k+1]/7) * 7
        }
        H[i, 1:nSecondary, 1:nPrimary] ~ dLikelihood(
            first = first[i], last = last[i], J = J[i,1:nPrimary],
            lambda = lambda[1:2,gr[i]], tod = tod[1:nPrimary,1:nSecondary],
            g = g[i,1:nPrimary,1:R], G = G[i,1:nPrimary],
            z = z[i,1:nPrimary], phi = phi[gr[i],1:(nPrimary-1)])
    }
})
```