Olivier Gimenez

Bayesian Analysis of Capture-Recapture Data with Hidden Markov Models

Theory and Case Studies in R

Contents

List of Tables

List of Figures

Welcome

Welcome to the online version of the book *Bayesian Analysis of Capture-Recapture Data with Hidden Markov Models – Theory and Case Studies in R*.

The HMM framework has gained much attention in the ecological literature over the last decade, and has been suggested as a general modelling framework for the demography of plant and animal populations. In particular, HMMs are increasingly used to analyse capture-recapture data and estimate key population parameters (e.g., survival, dispersal, recruitment or abundance) with applications in all fields of ecology.

In parallel, Bayesian statistics is well established and fast growing in ecology and related disciplines, because it resonates with scientific reasoning and allows accommodating uncertainty smoothly. The popularity of Bayesian statistics also comes from the availability of free pieces of software (WinBUGS, OpenBUGS, JAGS, Stan, NIMBLE) that allow practitioners to code their own analyses.

This book offers a Bayesian treatment of HMMs applied to capture-recapture data. You will learn to use the R package NIMBLE which is seen by many as the future of Bayesian statistical ecology to deal with complex models and/or big data. An important part of the book consists in case studies presented in a tutorial style to abide by the "learning by doing" philosophy.

I'm currently writing this book, and I welcome any feedback. You may raise an issue here[1], amend directly the R Markdown file that generated the page you're reading by clicking on the 'Edit this page' icon in the right panel, or email me[2]. Many thanks!

[1] https://github.com/oliviergimenez/banana-book/issues

[2] mailto:olivier.gimenez@cefe.cnrs.fr

Olivier Gimenez, Montpellier, France
Last updated: March 05, 2022

License

[3] http://creativecommons.org/licenses/by-nc-nd/4.0/

[4] https://creativecommons.org/publicdomain/zero/1.0/

Preface

Why this book?

To be completed. Why and what of capture-recapture data and models, with fields of application.[5] Brief history of capture-recapture, with switch to state-space/hidden Markov model (HMM) formulation. Flexibility of HMM to decompose complex problems in smaller pieces that are easier to understand, model and analyse. From satellite guidance to conservation of endangered species. Why Bayes? Also three of my fav research topics – capture-recapture, HMM and Bayes statistics – let's enjoy this great cocktail together.

Who should read this book?

This book is aimed at beginners who're comfortable using R and write basic code (including loops), as well as connoisseurs of capture-recapture who'd like to tap into the power of the Bayesian side of statistics. For both audiences, thinking in the HMM framework will help you in confidently building models and make the most of your capture-recapture data.

[5] Watch out nice Johnny Ball's video https://www.youtube.com/watch?v=tyX79mPm2xY.

What will you learn?

The book is divided into five parts. The first part is aimed at getting you up-to-speed with Bayesian statistics, NIMBLE, and hidden Markov models. The second part will teach you all about capture-recapture models for open populations, with reproducible R code to ease the learning process. In the third part, we will focus on issues in inferring states (dealing with uncertainty in assignment, modelling waiting time distribution). The fourth part provides real-world case studies from the scientific literature that you can reproduce using material covered in previous chapters. These problems can either i) be used to cement and deepen your understanding of methods and models, ii) be adapted for your own purpose, or iii) serve as teaching projects. The fifth and last chapter closes the book with take-home messages and recommendations, a list of frequently asked questions and references cited in the book. **Likely to be amended after feedbacks.**

What won't you learn?

There is hardly any maths in this book. The equations I use are either simple enough to be understood without a background in maths, or can be skipped without prejudice. I do not cover Bayesian statistics or even hidden Markov models fully, I provide just what you need to work with capture-recapture data. If you are interested in knowing more about these topics, hopefully the section Suggested reading at the end of each chapter will put you in the right direction. There are also a number of important topics specific to capture-recapture that I do not cover, including closed-population capture-recapture models [**?**], and spatial capture-recapture models [**?**]. These models can be treated as HMMs, but for now the usual formulation is just fine. **There will be spatial considerations in the Covariates chapter w/ splines and CAR. I'm not sure yet about SCR models (R. Glennie's Biometrics paper on HMMs**

and open pop SCR will not be easy to Bayes transform and implement in NIMBLE).

Prerequisites

This book uses primarily the R package NIMBLE, so you need to install at least R and NIMBLE. A bunch of other R packages are used. You can install them all at once by running:

```
install.packages(c(
  "magick", "MCMCvis", "nimble", "pdftools",
  "tidyverse", "wesanderson"
))
```

Acknowledgements

To be completed.

How this book was written

I am writing this book in RStudio[6] using bookdown[7]. The book website[8] is hosted with GitHub Pages[9], and automatically updated after every push by Github Actions[10]. The source is available from GitHub[11].

[6] http://www.rstudio.com/ide/
[7] http://bookdown.org/
[8] https://oliviergimenez.github.io/banana-book
[9] https://pages.github.com/
[10] https://github.com/features/actions
[11] https://github.com/oliviergimenez/banana-book

The version of the book you're reading was built with R version 4.1.0 (2021-05-18) and the following packages:

package	version	source
magick	2.7.3	CRAN (R 4.1.0)
MCMCvis	0.15.3	CRAN (R 4.1.0)
nimble	0.11.1	CRAN (R 4.1.0)
pdftools	3.0.1	CRAN (R 4.1.0)
tidyverse	1.3.1	CRAN (R 4.1.0)
wesanderson	0.3.6	CRAN (R 4.1.0)

About the author

My name is Olivier Gimenez (https://oliviergimenez.github.io/). I am a senior (euphemism for not so young anymore) scientist at the National Centre for Scientific Research (CNRS) in the beautiful city of Montpellier, France.

I struggled studying maths, obtained a PhD in applied statistics a long time ago in a galaxy of wine and cheese. I was awarded my habilitation (https://en.wikipedia.org/wiki/Habilitation) in ecology and evolution so that I could stop pretending to understand what my colleagues were talking about. More recently I embarked in sociology studies because hey, why not.

Lost somewhere at the interface of animal ecology, statistical modeling and social sciences, my so-called expertise lies in population dynamics and species distribution modeling to address questions in ecology and conservation biology about the impact of human activities and the management of large carnivores. I would be nothing without the students and colleagues who are kind enough to bear with me.

You may find me on Twitter (https://twitter.com/oaggimenez), GitHub (https://github.com/oliviergimenez), or get in touch by email[12].

[12] mailto:olivier.gimenez@cefe.cnrs.fr

Part I

I. Fundations

Introduction

1

Bayesian statistics & MCMC

1.1 Introduction

In this first chapter, you will learn what the Bayesian theory is, and how you may use it with a simple example. You will also see how to implement simulation algorithms to implement the Bayesian method for more complex analyses. This is not an exhaustive treatment of Bayesian statistics, but you should get what you need to navigate through the rest of the book.

1.2 Bayes' theorem

Let's not wait any longer and jump into it. Bayesian statistics relies on the Bayes' theorem (or law, or rule, whatever you prefer) named after Reverend Thomas Bayes (Figure 1.1). This theorem was published in 1763 two years after Bayes' death thanks to his friend's efforts Richard Price, and was independently discovered by Pierre-Simon Laplace [**?**].

As we will see in a minute, Bayes' theorem is all about conditional probabilities, which are somehow tricky to understand. Conditional probability of outcome or event A given event B, which we denote $\Pr(A \mid B)$, is the probability that A occurs, revised by considering the additional information that event B has occurred.[1] The order in which A and B

[1] For example, a friend of yours rolls a fair dice and asks you the probability that the outcome was a six (event A). Your answer is 1/6 because each side of the dice is equally likely to come up. Now imagine that you're told the number rolled was even (event B) before you answer your friend's question. Because there are only three even

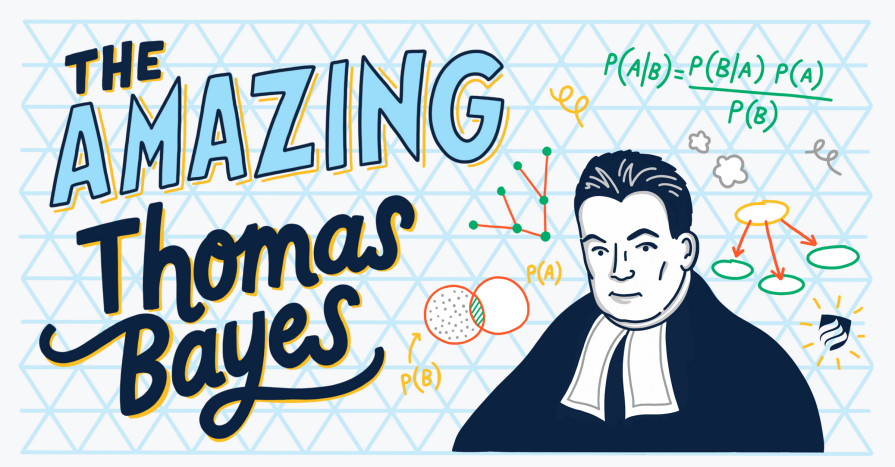


FIGURE 1.1: Cartoon of Thomas Bayes with Bayes' theorem in background. Source: [James Kulich](https://www.elmhurst.edu/blog/thomas-bayes/)

appear is important, make sure you do not confuse $\Pr(A \mid B)$ and $\Pr(B \mid A)$.

Bayes' theorem (Figure **??**) gives you $\Pr(A \mid B)$ using marginal probabilities $\Pr(A)$ and $\Pr(B)$ and $\Pr(B \mid A)$:

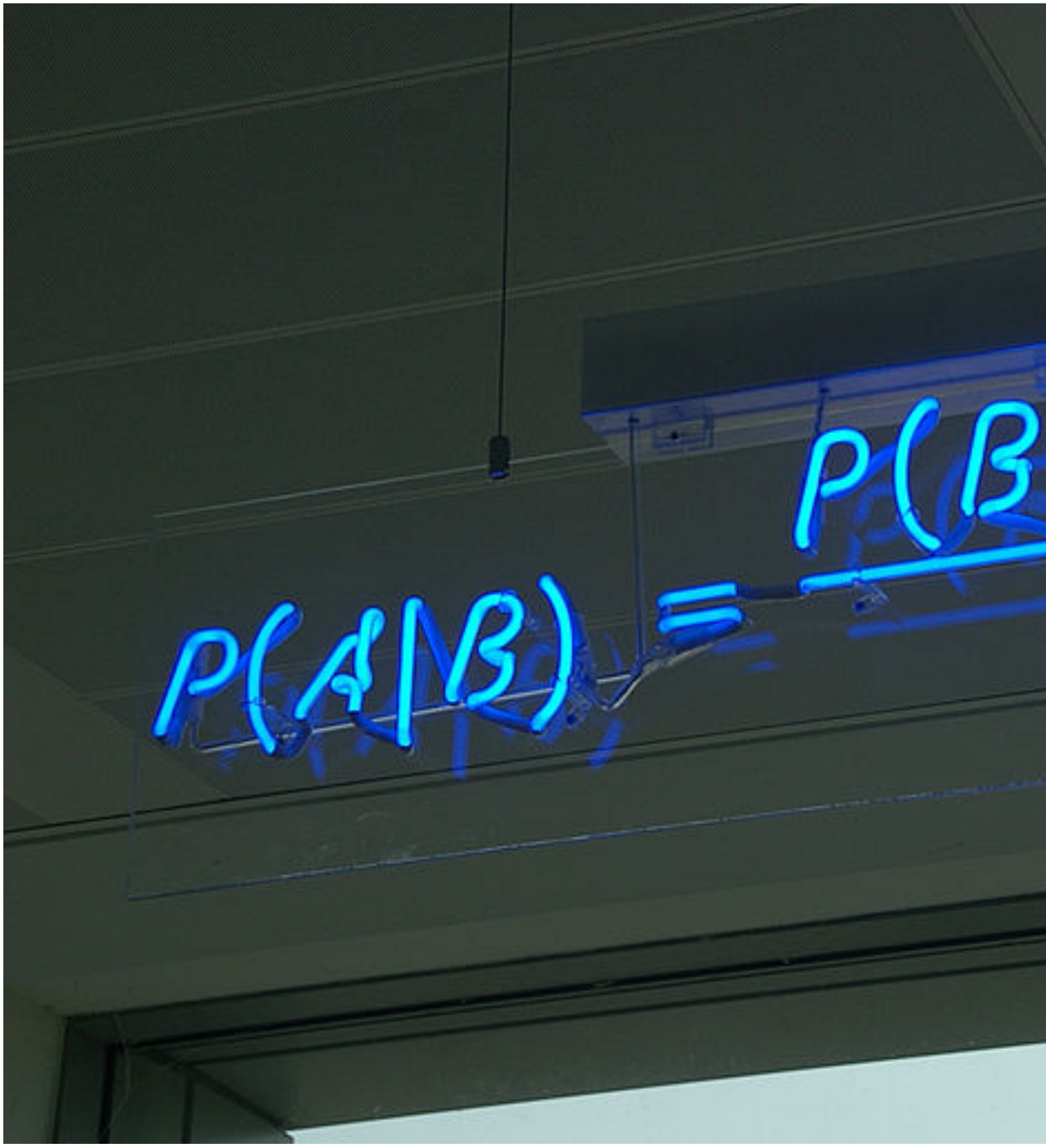
$$\Pr(A \mid B) = \frac{\Pr(B \mid A)\ \Pr(A)}{\Pr(B)}.$$

Originally, Bayes' theorem was seen as a way to infer an unkown cause A of a particular effect B, knowing the probability of effect B given cause A. Think for example of a situation where a medical diagnosis is needed, with A an unkown disease and B symptoms, the doctor knows P(symptoms|disease) and wants to derive P(disease|symptoms). This way of reversing $\Pr(B \mid A)$ into $\Pr(A \mid B)$ explains why Bayesian thinking used to be referred to as 'inverse probability'.

I don't know about you, but I need to think twice for not messing the letters around. I find it easier to remember Bayes' theorem written like this[3]:

numbers, one of which is six, you may revise your answer for the probability that a six was rolled from 1/6 to $\Pr(A \mid B) = 1/3$.

[3]When teaching Bayes' theorem, I am very much inspired by Tristan Mahr's slides



captionBayes' theorem spelt out in blue neon. Source: Wikipedia[2]}

$$\Pr(\text{hypothesis} \mid \text{data}) = \frac{\Pr(\text{data} \mid \text{hypothesis})\ \Pr(\text{hypothesis})}{\Pr(\text{data})}$$

The *hypothesis* is a working assumption about which you want to learn using *data*. In capture–recapture analyses, the hypothesis might be a parameter like detection probability, or regression parameters in a relationship between survival probability and a covariate. Bayes' theorem tells us how to obtain the probability of a hypothesis given the data we have.

This is great because think about it, this is exactly what the scientific method is! We'd like to know how plausible some hypothesis is based on some data we collected, and possibly compare several hypotheses among them. In that respect, the Bayesian reasoning matches the scientific reasoning, which probably explains why the Bayesian framework is so natural for doing and understanding statistics.

You might ask then, why is Bayesian statistics not the default in statistics? Clearly, because of futile wars between male statisticians (including Ronald Fisher, Jerzy Neyman and Egon Sharpe Pearson among others), little progress was made for over two centuries. Also, until recently, there were practical problems to implement Bayes' theorem. Recent advances in computational power coupled with the development of new algorithms have led to a great increase in the application of Bayesian methods within the last three decades.

1.3 What is the Bayesian approach?

Typical statistical problems involve estimating a parameter (or several parameters) θ with available data. To do so, you might be more used to the frequentist rather than the Bayesian method. The frequentist approach, and in particular maximum likelihood estimation (MLE),

from his introduction to Bayesian regression https://www.tjmahr.com/bayes-intro-lecture-slides-2017/

assumes that the parameters are fixed, and have unknown values to be estimated. Therefore classical estimates are generally point estimates of the parameters of interest. In contrast, the Bayesian approach assumes that the parameters are not fixed, and have some unknown distribution[4].

The Bayesian approach is based upon the idea that you, as an experimenter, begin with some prior beliefs about the system. Then you collect data and update your prior beliefs on the basis of observations. These observations might arise from field work, lab work or from expertise of your esteemed colleagues. This updating process is based upon Bayes' theorem. Loosely, let's say $A = \theta$ and $B =$ data, then Bayes' theorem gives you a way to estimate parameter θ given the data you have:

$$\Pr(\theta \mid \text{data}) = \frac{\Pr(\text{data} \mid \theta) \times \Pr(\theta)}{\Pr(\text{data})}.$$

Let's spend some time going through each quantity in this formula.

On the left-hand side is the posterior distribution. It represents what you know after having seen the data. This is the basis for inference and clearly what you're after, a distribution, possibly multivariate if you have more than one parameter.

On the right-hand side, there is the likelihood. This quantity is the same as in the MLE approach. Yes, the Bayesian and frequentist approaches have the same likelihood at their core, which mostly explains why results often do not differ much. The likelihood captures the information you have in your data, given a model parameterized with θ.

Then we have the prior distribution. This quantity represents what you know before seeing the data. This is the source of much discussion about the Bayesian approach. It may be vague if you don't know any-

[4] A probability distribution is a mathematical expression that gives the probability for a random variable to take particular values. A probability distribution may be either discrete (e.g., the Bernoulli, Binomial or Poisson distribution) or continuous (e.g., the Gaussian distribution also known as the normal distribution)

thing about θ. Usually however, you never start from scratch, and you'd like your prior to reflect the information you have[5].

Last, we have $\Pr(\text{data})$ which is sometimes called the average likelihood because it is obtained by integrating the likelihood with respect to the prior $\Pr(\text{data}) = \int L(\text{data} \mid \theta) \Pr(\theta) d\theta$ so that the posterior is standardized, that is it integrates to one for the posterior to be a distribution. The average likelihood is an integral with dimension the number of parameters θ you need to estimate. This quantity is difficult, if not impossible, to calculate in general. This is one of the reasons why the Bayesian method wasn't used until recently, and why we need algorithms to estimate posterior distributions as I illustrate in the next section.

1.4 Approximating posteriors via numerical integration

Let's take an example to illustrate Bayes' theorem. Say we capture, mark and release $n = 57$ animals at the beginning of a winter, out of which we recapture $y = 19$ animals alive[6]. We'd like to estimate winter survival θ.

```
y <- 19 # nb of success
n <- 57 # nb of attempts
```

We build our model first. Assuming all animals are independent of each other and have the same survival probability, then y the number of alive animals at the end of the winter is a binomial distribution[7] with n trials and θ the probability of success:

$$y \sim \text{Binomial}(n, \theta) \qquad \text{[likelihood]}$$

[5] Shall I include a section on sensitivity analyses in this chapter or later in the book? Cross-reference section in Survival chapter where prior elicitation is covered.

[6] We used a similar example in **?**

[7] I follow **?** and use labels on the right to help remember what each line is about.

This likelihood can be visualised in R:

```
grid <- seq(0, 1, 0.01) # grid of values for survival
likelihood <- dbinom(y, n, grid) # compute binomial likelihood
df <- data.frame(survival = grid, likelihood = likelihood)
df %>%
  ggplot() +
  aes(x = survival, y = likelihood) +
  geom_line(size = 1.5)
```

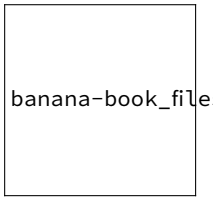


FIGURE 1.2: Binomial likelihood with $n = 57$ released animals and $y = 19$ survivors after winter. The value of survival (on the x-axis) that corresponds to the maximum of the likelihood function (on the y-axis) is the MLE, or the proportion of success in this example, close to 0.33.

Besides the likelihood, priors are another component of the model in the Bayesian approach. For a parameter that is a probability, the one thing we know is that the prior should be a continuous random variable that lies between 0 and 1. To reflect that, we often go for the uniform distribution $U(0, 1)$ to imply *vague* priors. Here vague means that survival has, before we see the data, the same probability of falling between 0.1 and 0.2 and between 0.8 and 0.9, for example.

$$\theta \sim \text{Uniform}(0, 1) \qquad \text{[prior for } \theta\text{]}$$

Now we apply Bayes' theorem. We write a R function that computes the product of the likelihood times the prior, or the numerator in Bayes' theorem: $\Pr(\text{data} \mid \theta) \times \Pr(\theta)$

```
numerator <- function(theta) dbinom(y, n, theta) * dunif(theta, 0, 1)
```

We write another function that calculates the denominator, the average likelihood: $\Pr(\text{data}) = \int L(\theta \mid \text{data}) \Pr(\theta) d\theta$

```
denominator <- integrate(numerator,0,1)$value
```

We use the R function `integrate` to calculate the integral in the denominator, which implements quadrature techniques to divide in little squares the area underneath the curve delimited by the function to integrate (here the numerator), and count them.

Then we get a numerical approximation of the posterior in Figure 1.3 by applying Bayes' theorem.

```
grid <- seq(0, 1, 0.01) # grid of values for theta
numerical_posterior <- data.frame(survival = grid,
                                  posterior = numerator(grid)/denominator) # Bayes' theorem
numerical_posterior %>%
  ggplot() +
  aes(x = survival, y = posterior) +
  geom_line(size = 1.5)
```

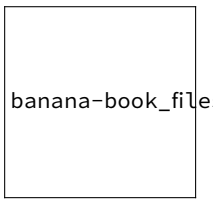


FIGURE 1.3: Winter survival posterior distribution obtained by numerical integration.

How good is our numerical approximation of survival posterior distribution? Ideally, we would want to compare the approximation to the true posterior distribution. Although a closed-form expression for the posterior distribution is in general intractable, when you combine a

binomial likelihood together with a beta distribution as a prior, then the posterior distribution is also a beta distribution, which makes it amenable to all sorts of exact calculations[8]. The beta distribution is continuous between 0 and 1, and extends the uniform distribution to situations where not all outcomes are equally likely. It has two parameters a and b that control its shape (Figure 1.4).

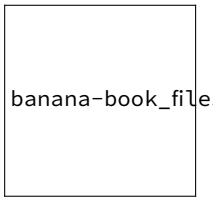
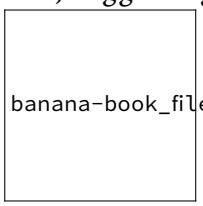


FIGURE 1.4: The distribution beta(a,b) for different values of a and b. Note that for $a = b = 1$, we get the uniform distribution between 0 and 1 in the top left panel. When a and b are equal, the distribution is symmetric, and the bigger a and b, the more peaked the distribution or the smaller the variance.

If the likelihood of the data y is binomial with n trials and probability of success θ, and the prior is a beta distribution with parameters a and b, then the posterior is a beta distribution with parameters $a + y$ and $b + n - y$[9]. In our example, we have $n = 57$ trials and $y = 19$ animals that survived and a uniform prior between 0 and 1 or a beta distribution with parameters $a = b = 1$, therefore survival has a beta posterior distribution with parameters 20 and 39. In Figure **??**, we superimpose the exact posterior and the numerical approximation. Clearly, the two distributions are indistinguishable, suggesting that the numerical approximation is more than fine.



In our example, we have a single parameter to estimate, winter survival.

[8] We say that the beta distribution is the conjugate prior distribution for the binomial distribution.

[9] **provide a sketch of the proof**

This means dealing with a one-dimensional integral in the denominator which is pretty easy with quadrature techniques and the R function `integrate()`. Now what if we had multiple parameters? For example, imagine you'd like to fit a capture-recapture model with detection probability p and regression parameters α and β for the intercept and slope of a relationship between survival probability and a covariate, then Bayes' theorem gives you the posterior distribution of all three parameters together:

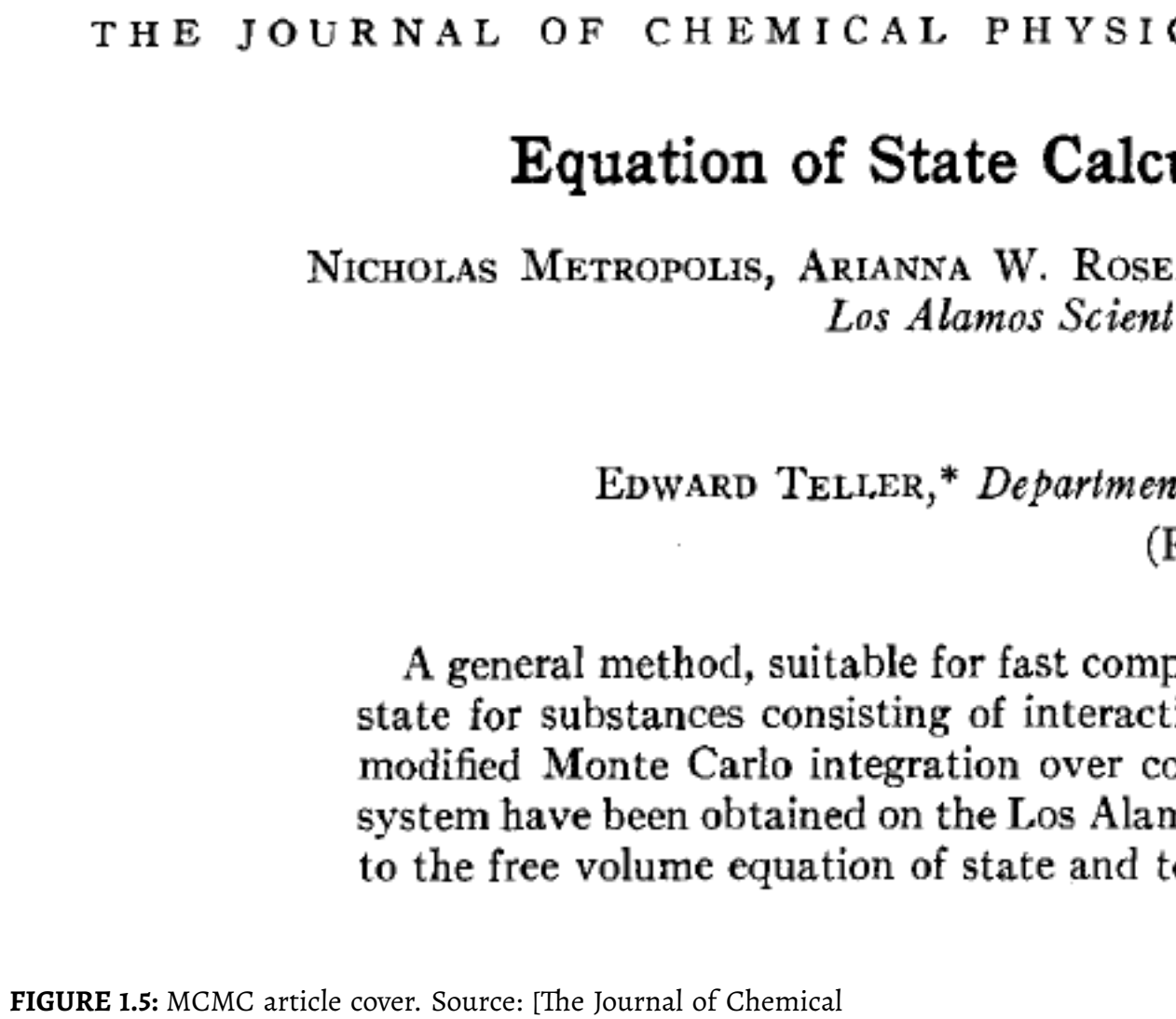
$$\Pr(\alpha, \beta, p \mid \text{data}) = \frac{\Pr(\text{data} \mid \alpha, \beta, p) \times \Pr(\alpha, \beta, p)}{\iiint \Pr(\text{data} \mid \alpha, \beta, p) \Pr(\alpha, \beta, p) d\alpha d\beta dp}$$

There are two computational challenges with this formula. First, do we really wish to calculate a three-dimensional integral? The answer is no, one-dimensional and two-dimensional integrals are so much further we can go with standard methods. Second, we're more interested in a posterior distribution for each parameter separately than the joint posterior distribution. The so-called marginal distribution of p for example is obtained by integrating over all the other parameters – a two-dimensional integral in this example. Now imagine with tens or hundreds of parameters to estimate, these integrals become highly multi-dimensional and simply intractable. In the next section, I introduce powerful simulation methods to circumvent this issue.

1.5 Markov chain Monte Carlo (MCMC)

In the early 1990s, statisticians rediscovered work from the 1950's in physics. In a famous paper that would lay the fundations of modern Bayesian statistics (Figure 1.5), the authors use simulations to approximate posterior distributions with some precision by drawing large samples. This is a neat trick to avoid explicit calculation of the multi-dimensional integrals we struggle with when using Bayes' theorem.

These simulation algorithms are called Markov chain Monte Carlo (MCMC), and they definitely gave a boost to Bayesian statistics. There



THE JOURNAL OF CHEMICAL PHYSI

Equation of State Calcu

NICHOLAS METROPOLIS, ARIANNA W. ROSE
Los Alamos Scient

EDWARD TELLER,* *Departmen*
(R

A general method, suitable for fast comp
state for substances consisting of interacti
modified Monte Carlo integration over co
system have been obtained on the Los Alam
to the free volume equation of state and t

FIGURE 1.5: MCMC article cover. Source: [The Journal of Chemical Physics](https://aip.scitation.org/doi/10.1063/1.1699114)

are two parts in MCMC, Markov chain and Monte Carlo, let's try and make sense of these terms.

1.5.1 Monte Carlo integration

What does Monte Carlo stand for? Monte Carlo integration is a simulation technique to calculate integrals of any function f of random variable X with distribution $\Pr(X)$ say $\int f(X) \Pr(X) dX$. You draw values $X_1, \ldots, X_k$ from $\Pr(X)$ the distribution of X, apply function f to these values, then calculate the mean of these new values $\frac{1}{k} \sum_{i=1}^{k} f(X_i)$ to approximate the integral. How is Monte Carlo integration used in a Bayesian context? The posterior distribution contains all the information we need about the parameter to be estimated. When dealing with many parameters however, you may want to summarise posterior results by calculating numerical summaries. The simplest numerical summary is the mean of the posterior distribution, $E(\theta) = \int \theta \Pr(\theta|\text{data})$, where X is θ now and f is the identity function. Posterior mean can be calculated with Monte Carlo integration:

```
sample_from_posterior <- rbeta(1000, 20, 39) # draw 1000 values from posterior survival beta(2
mean(sample_from_posterior) # compute mean with Monte Carlo integration
## [1] 0.3407
```

You may check that the mean we have just calculated matches closely the expectation of a beta distribution[10]:

```
20/(20+39) # expectation of beta(20,39)
## [1] 0.339
```

Another useful numerical summary is the credible interval within which our parameter falls with some probability, usually 0.95 hence a 95% credible interval. Finding the bounds of a credible interval requires calculating quantiles, which in turn involves integrals and the use of

[10] If X is a random variable with distribution beta(a, b), then $E(X) = \frac{a}{a+b}$

Monte Carlo integration. A 95% credible interval for winter survival can be obtained in R with:

```
quantile(sample_from_posterior, probs = c(2.5/100, 97.5/100))
##  2.5% 97.5%
## 0.222 0.464
```

1.5.2 Markov chains

What is a Markov chain? A Markov chain is a random sequence of numbers, in which each number depends only on the previous number. An example is the weather in my home town in Southern France, Montpellier, in which a sunny day is most likely to be followed by another sunny day, say with probability 0.8, and a rainy day is rarely followed by another rainy day, say with probability 0.1. The dynamic of this Markov chain is captured by the transition matrix $\mathbf{\Gamma}$:

$$\mathbf{\Gamma} = \begin{array}{c} \begin{array}{cc} \text{sunny tomorrow} & \text{rainy tomorrow} \end{array} \\ \begin{pmatrix} 0.8 & 0.2 \\ 0.9 & 0.1 \end{pmatrix} \end{array} \begin{array}{l} \text{sunny today} \\ \text{rainy today} \end{array}$$

In rows the weather today, and in columns the weather tomorrow. The cells give the probability of a sunny or rainy day tomorrow, given the day is sunny or rainy today. Under certain conditions[11], a Markov chain will converge to a unique stationary distribution. In our weather example, let's run the Markov chain for 20 steps:

```
weather <- matrix(c(0.8, 0.2, 0.9, 0.1), nrow = 2, byrow = T) # transition matrix
steps <- 20
for (i in 1:steps){
  weather <- weather %*% weather # matrix multiplication
}
round(weather, 2) # matrix product after 20 steps
##      [,1] [,2]
## [1,] 0.82 0.18
## [2,] 0.82 0.18
```

[11] The Markov chain is irreducible and aperiodic.

Each row of the transition matrix converges to the same distribution $(0.82, 0.18)$ as the number of steps increases. Convergence happens no matter which state you start in, and you always have probability 0.82 of the day being sunny and 0.18 of the day being rainy.

Back to MCMC, the core idea is that you can build a Markov chain with a given stationary distribution set to be the desired posterior distribution.

> Putting Monte Carlo and Markov chains together, MCMC allows us to generate a sample of values (Markov chain) whose distribution converges to the posterior distribution, and we can use this sample of values to calculate any posterior summaries (Monte Carlo), such as posterior means and credible intervals.

1.5.3 Metropolis algorithm

There are several ways of constructing Markov chains for Bayesian inference[12]. Here I illustrate the Metropolis algorithm and how to implement it in practice[13].

Let's go back to our example on animal survival estimation. We illustrate sampling from survival posterior distribution. We write functions for likelihood, prior and posterior.

```
# 19 animals recaptured alive out of 57 captured, marked and released
survived <- 19
released <- 57

# binomial log-likelihood function
loglikelihood <- function(x, p){
  dbinom(x = x, size = released, prob = p, log = TRUE)
}
```

[12] You might have heard about the Metropolis-Hastings or the Gibbs sampler. Have a look to https://github.com/chi-feng/mcmc-demo for an interactive gallery of MCMC algorithms.

[13] This presentation is largely inspired by **?**

```r
# uniform prior density
logprior <- function(p){
  dunif(x = p, min = 0, max = 1, log = TRUE)
}

# posterior density function (log scale)
posterior <- function(x, p){
  loglikelihood(x, p) + logprior(p) # - log(Pr(data))
}
```

The Metropolis algorithm works as follows:

1. We pick a value of the parameter to be estimated. This is where we start our Markov chain – this is a *starting* value.
2. To decide where to go next, we propose to move away from the current value of the parameter – this is a *candidate* value. To do so, we add to the current value some random value from e.g. a normal distribution with some variance – this is a *proposal* distribution. The Metropolis algorithm is a particular case of the Metropolis-Hastings algorithm with symmetric proposals.
3. We compute the ratio of the probabilities at the candidate and current locations $R = \frac{\Pr(\text{candidate}|\text{data})}{\Pr(\text{current}|\text{data})}$. This is where the magic of MCMC happens, in that $\Pr(\text{data})$, the denominator in the Bayes' theorem, appears in both the numerator and the denominator in R therefore cancels out and does not need to be calculated.
4. If the posterior at the candidate location $\Pr(\text{candidate}|\text{data})$ is higher than at the current location $\Pr(\text{current}|\text{data})$, in other words when the candidate value is more plausible than the current value, we definitely accept the candidate value. If not, then we accept the candidate value with probability R and reject with probability $1 - R$. For example, if the candidate

value is ten times less plausible than the current value, then we accept with probability 0.1 and reject with probability 0.9. How does it work in practice? We use a continuous spinner that lands somewhere between 0 and 1 – call the random spin X. If X is smaller than R, we move to the candidate location, otherwise we remain at the current location. We do not want to accept or reject too often. In practice, the Metropolis algorithm should have an acceptance probability between 0.2 and 0.4, which can be achieved by *tuning* the variance of the normal proposal distribution.

5. We repeat 2-4 a number of times – or *steps*.

Enough of the theory, let's implement the Metropolis algorithm in R. Let's start by setting the scene.

```
steps <- 100 # number of steps
theta.post <- rep(NA, steps) # vector to store samples
accept <- rep(NA, steps) # keep track of accept/reject
set.seed(1234) # for reproducibility
```

Now follow the 5 steps we've just described. First, we pick a starting value, and store it (step 1).

```
inits <- 0.5
theta.post[1] <- inits
accept[1] <- 1
```

Then, we need a function to propose a candidate value. We add a value taken from a normal distribution with mean zero and standard deviation we call *away*. We work on the logit scale to make sure the candidate value for survival lies between 0 and 1.

```
move <- function(x, away = 1){ # by default, standard deviation of the proposal distribution t
  logitx <- log(x / (1 - x)) # apply logit transform (-infinity,+infinity)
  logit_candidate <- logitx + rnorm(1, 0, away) # add a value taken from N(0,sd=away) to curre
```

```
  candidate <- plogis(logit_candidate) # back-transform (0,1)
  return(candidate)
}
```

Now we're ready for steps 2, 3 and 4. We write a loop to take care of step 5. We start at initial value 0.5 and run the algorithm for 100 steps or iterations.

```
for (t in 2:steps){ # repeat steps 2-4 (step 5)

  # propose candidate value for survival (step 2)
  theta_star <- move(theta.post[t-1])

  # calculate ratio R (step 3)
  pstar <- posterior(survived, p = theta_star)
  pprev <- posterior(survived, p = theta.post[t-1])
  logR <- pstar - pprev # likelihood and prior are on the log scale
  R <- exp(logR)

  # accept candidate value or keep current value (step 4)
  X <- runif(1, 0, 1) # spin continuous spinner
  if (X < R){
    theta.post[t] <- theta_star # accept candidate value
    accept[t] <- 1 # accept
  }
  else{
    theta.post[t] <- theta.post[t-1] # keep current value
    accept[t] <- 0 # reject
  }
}
```

We get the following values.

```
head(theta.post) # first values
## [1] 0.5000 0.2302 0.2906 0.2906 0.2980 0.2980
```

```
tail(theta.post) # last values
## [1] 0.2622 0.2622 0.2622 0.3727 0.3232 0.3862
```

Visually, you may look at the chain in Figure 1.6 called a trace plot.

FIGURE 1.6: Visualisation of a Markov chain starting at value 0.5, with steps or iterations on the x-axis, and samples on the y-axis. This graphical representation is called a trace plot.

The acceptance probability is the average number of times we accepted a candidated value, which is 0.44 and almost satisfying.

Can we run another chain and start at initial value 0.2 this time? Yes, just go through the same algorithm again, and visualise the results in Figure 1.7.

FIGURE 1.7: Trace plot of survival for two chains starting at 0.2 (yellow) and 0.5 (blue) run for 100 steps.

Notice that we do not get the exact same results because the algorithm is stochastic. The question is to know whether we have reached the stationary distribution. Let's increase the number of steps and run a chain with 5000 iterations as in Figure 1.8.

FIGURE 1.8: Trace plot of survival for a chain starting at 0.5 and 1000 steps.

This is what we're after, a trace plot that looks like a beautiful lawn, see Section 1.6. I find it informative to look at the animated version of Figure 1.8, it helps understanding the stochastic behavior of the algorithm, and also to realise how the chains converge to their stationary distribution, see Figure 1.9.

Once the stationary distribution is reached, you may regard the realisations of the Markov chain as a sample from the posterior distribution, and obtain numerical summaries. In the next section, we consider several important implementation issues.

FIGURE 1.9: Animated trace plot of survival with three chains starting at 0.2, 0.5 and 0.7 run for 1000 steps.

1.6 Assessing convergence

When implementing MCMC, we need to determine how long it takes for our Markov chain to converge to the target distribution, and the number of iterations we need after achieving convergence to get reasonable Monte Carlo estimates of numerical summaries (posterior means and credible intervals).

1.6.1 Burn-in

In practice, we discard observations from the start of the Markov chain and just use observations from the chain once it has converged. The initial observations that we discard are usually referred to as the *burn-in*.

The simplest method to determine the length of the burn-in period is to look at trace plots. Going back to our example, we see from the trace plot in Figure 1.10 that we need at least 100 iterations to achieve convergence toward an average survival around 0.3. It is always better to be conservative when specifying the length of the burn-in period, and in this example, we would use 250 or even 500 iterations as a burn-in. The length of the burn-in period can be determined by performing preliminary MCMC short runs.

Inspecting the trace plot for a single run of the Markov chain is useful. However, we usually run the Markov chain several times, starting from different over-dispersed points, to check that all runs achieve the same stationary distribution. This approach is formalised by using the Brooks-Gelman-Rubin (BGR) statistic $\hat{R}$ which measures the ratio of the total variability combining multiple chains (between-chain plus within-chain) to the within-chain variability. The BGR statistic asks

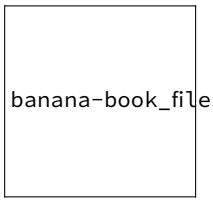


FIGURE 1.10: Determining the length of the burn-in period. The chain starts at value 0.99 and rapidly stabilises, with values bouncing back and forth around 0.3 from the 100th iteration onwards. You may choose the shaded area as the burn-in, and discard the corresponding values.

whether there is a chain effect, and is very much alike the F test in an analysis of variance. Values below 1.1 indicate likely convergence.

Back to our example, we run two Markov chains with starting values 0.2 and 0.8 using 100 up to 5000 iterations, and calculate the BGR statistic using half the number of iterations as the length of the burn-in. From Figure 1.11, we get a value of the BGR statistic near 1 by up to 2000 iterations, which suggests that with 2000 iterations as a burn-in, there is no evidence of a lack of convergence.

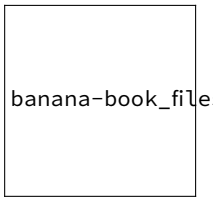


FIGURE 1.11: Brooks-Gelman-Rubin statistic as a function of the number of iterations.

It is important to bear in mind that a value near 1 for the BGR statistic is only a necessary *but not sufficient* condition for convergence. In other words, this diagnostic cannot tell you for sure that the Markov chain has achieved convergence, only that it has not.[14]

[14] Cross-reference sections on local minima and parameter redundancy for pathological cases.

1.6.2 Chain length

How long of a chain is needed to produce reliable parameter estimates? To answer this question, you need to keep in mind that successive steps in a Markov chain are not independent – this is usually referred to as *autocorrelation*. Ideally, we would like to keep autocorrelation as low as possible. Here again, trace plots are useful to diagnose issues with autocorrelation. Let's get back to our survival example. Figure 1.12 shows trace plots for different values of the standard deviation (parameter *away*) of the (normal) proposal distribution we use to propose a candidate value (Section 1.5.3). Small and big moves provide high correlations between successive observations of the Markov chain, whereas a standard deviation of 1 allows efficient exploration of the parameter space. The movement around the parameter space is referred to as *mixing*. Mixing is bad when the chain makes small and big moves, and good otherwise.

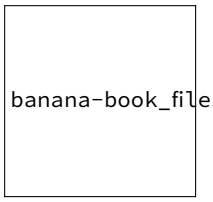


FIGURE 1.12: Trace plots for different values of the standard deviation (SD) of the proposal distribution. Left: The chain exhibits small moves and mixing is bad. Right: The chain exhibits big moves and mixing is bad. Middle: The chain exhibits adequate moves and mixing is good. Only the thousand last iterations are shown.

In addition to trace plots, autocorrelation function (ACF) plots are a convenient way of displaying the strength of autocorrelation in a given sample values. ACF plots provide the autocorrelation between successively sampled values separated by an increasing number of iterations, or *lag* (Figure 1.13).

Autocorrelation is not necessarily a big issue. Strongly correlated observations just require large sample sizes and therefore longer simulations. But how many iterations exactly? The effective sample size (`n.eff`) measures chain length while taking into account chain autocorrelation.

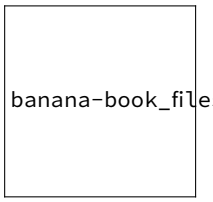


FIGURE 1.13: Autocorrelation function plots for different values of the standard deviation (SD) of the proposal distribution. Left and right: Autocorrelation is strong, decreases slowly with increasing lag and mixing is bad. Middle: Autocorrelation is weak, decreases rapidly with increasing lag and mixing is good.

You should check the `n.eff` of every parameter of interest, and of any interesting parameter combinations. In general, we need n.eff ≥ 1000 independent steps to get reasonable Monte Carlo estimates of model parameters. In the animal survival example, `n.eff` can be calculated with the R `coda::effectiveSize()` function.

Proposal SD	n.eff
0.1	224
1.0	1934
10.0	230

As expected, `n.eff` is less than the number of MCMC iterations because of autocorrelation. Only when the standard deviation of the proposal distribution is 1 and mixing is good (Figures 1.12 and 1.13) we get a satisfying effective sample size.

1.6.3 What if you have issues of convergence?

When diagnosing MCMC convergence, you will (very) often run into troubles. In this section you will find some helpful tips I hope.

When mixing is bad and effective sample size is small, you may just need to increase burn-in and/or sample more. Using more informative priors might also make Markov chains converge faster by helping your MCMC sampler (e.g. the Metropolis algorithm) navigating more efficiently the parameter space. In the same spirit, picking better initial values for starting the chain does not harm. For doing that, a strategy consists in using estimates from a simpler model for which your MCMC chains do converge.

If convergence issues persist, often there is a problem with your model[15]. A bug in the code? A typo somewhere? A mistake in your maths? As often when coding is involved, the issue can be identified by removing

[15] The quote 'When you have computational problems, often there's a problem with

complexities, and start with a simpler model until you find what the problem is.

A general advice is to see your model as a data generating tool in the first place, simulate data from it using some realistic values for the parameters, and try to recover these parameter values by fitting the model to the simulated data. Simulating from a model will help you understanding how it works, what it does not do, and the data you need to get reasonable parameter estimates.

We will see other strategies to improve convergence in the next chapters.[16]

1.7 Summary

- With the Bayes' theorem, you update your beliefs (prior) with new data (likelihood) to get posterior beliefs (posterior): posterior $\propto$ likelihood $\times$ prior.
- The idea of Markov chain Monte Carlo (MCMC) is to simulate values from a Markov chain which has a stationary distribution equal to the posterior distribution you're after.
- In practice, you run a Markov chain multiple times starting from over-dispersed initial values.
- You discard iterations in an initial burn-in phase and achieve convergence when all chains reach the same regime.
- From there, you run the chains long enough and proceed with calculating Monte Carlo estimates of numerical summaries (e.g. posterior means and credible intervals) for parameters.

[16] Cross reference relevant chapters. Option 1. Change your sampler. Option 2. Reparameterize (standardize covariates, plus non-centering: $\alpha \sim N(0, \sigma)$ becomes $\alpha = z\sigma$ with $z \sim N(0, 1)$).

1.8 Suggested reading

- Gelman, A. and Hill, J. (2006). Data Analysis Using Regression and Multilevel/Hierarchical Models (Analytical Methods for Social Research)[17]. Cambridge: Cambridge University Press.
- Gelman, A. and colleagues (2020). Bayesian workflow[18]. arXiv preprint.
- McCarthy, M. (2007). Bayesian Methods for Ecology[19]. Cambridge: Cambridge University Press.
- McElreath, R. (2020). Statistical Rethinking: A Bayesian Course with Examples in R and Stan (2nd ed.)[20]. CRC Press.

[17] https://www.cambridge.org/core/books/data-analysis-using-regression-and-multilevelhierarchical-models/32A29531C7FD730C3A68951A17C9D983

[18] https://arxiv.org/pdf/2011.01808.pdf

[19] https://www.cambridge.org/core/books/bayesian-methods-for-ecology/9225F65B8A25D69B0B6C50B5A9A78201

[20] https://xcelab.net/rm/statistical-rethinking/

2

NIMBLE tutorial

2.1 Introduction

In this second chapter, you will get familiar with NIMBLE, an R package that implements up-to-date MCMC algorithms for fitting complex models. NIMBLE spares you from coding the MCMC algorithms by hand, and requires only the specification of a likelihood and priors for model parameters. We will illustrate NIMBLE main features with a simple example, but the ideas hold for other problems.

2.2 What is NIMBLE?

NIMBLE stands for **N**umerical **I**nference for statistical **M**odels using **B**ayesian and **L**ikelihood **E**stimation. Briefly speaking, NIMBLE is an R package that implements for you MCMC algorithms to generate samples from the posterior distribution of model parameters. Freed from the burden of coding your own MCMC algorithms, you only have to specify a likelihood and priors to apply the Bayes theorem. To do so, NIMBLE uses a syntax very similar to the R syntax, which should make your life easier. This so-called BUGS language is also used by other programs like WinBUGS, OpenBUGS, and JAGS.

So why use NIMBLE you may ask? The short answer is that NIMBLE is capable of so much more than just running MCMC algorithms! First, you will work from within R, but in the background NIMBLE will translate your code in C++ for (in general) faster computation. Second, NIMBLE extends the BUGS language for writing new functions and distributions of your own, or borrow those written by others. Third, NIMBLE

gives you full control of the MCMC samplers, and you may pick other algorithms than the defaults. Fourth, NIMBLE comes with a library of numerical methods other than MCMC algorithms, including sequential Monte Carlo (for particle filtering) and Monte Carlo Expectation Maximization (for maximum likelihood). Last but not least, the development team is friendly and helpful, and based on users' feedbacks, NIMBLE folks work constantly at improving the package capabilities.

FIGURE 2.1: Logo of the NIMBLE R package designed by Luke Larson. **Ask Perry for context and meaning.**

2.3 Getting started

To run NIMBLE, you will need to:
1. Build a model consisting of a likelihood and priors.
2. Read in some data.
3. Specify parameters you want to make inference about.
4. Pick initial values for parameters to be estimated (for each chain).
5. Provide MCMC details namely the number of chains, the length of the burn-in period and the number of iterations following burn-in.

First things first, let's not forget to load the `nimble` package:

```
library(nimble)
```

Note that before you can install `nimble` like any other R package, Windows users will need to install `Rtools`, and Mac users will need to install `Xcode`. More at https://r-nimble.org/download.

Now let's go back to our example on animal survival from the previous chapter. First step is to build our model by specifying the binomial likelihood and a uniform prior on survival probability `theta`. We use the `nimbleCode()` function and wrap code within curly brackets:

```
model <- nimbleCode({
  # likelihood
  survived ~ dbinom(theta, released)
  # prior
  theta ~ dunif(0, 1)
  # derived quantity
  lifespan <- -1/log(theta)
})
```

You can check that the `model` R object contains your code:

```
model
## {
##     survived ~ dbinom(theta, released)
##     theta ~ dunif(0, 1)
##     lifespan <- -1/log(theta)
## }
```

In the code above, `survived` and `released` are known, only `theta` needs to be estimated. The line `survived ~ dbinom(theta, released)` states that the number of successes or animals that have survived over winter `survived` is distributed as (that's the `~`) as a binomial with `released` trials and probability of success or survival `theta`. Then the line `theta ~ dunif(0, 1)` assigns a uniform between 0 and 1 as a prior distribution to the survival probability. This is all you need, a likelihood and priors

for model parameters, NIMBLE knows the Bayes theorem. The last line `lifespan <- - 1/log(theta)` calculates a quantity derived from `theta`, which is the expected lifespan assuming constant survival[1].

A few comments:

- The most common distributions are available in NIMBLE. Among others, we will use later in the book `dbeta`, `dmultinom` and `dnorm`. If you cannot find what you need in NIMBLE, you can write your own distribution as illustrated in Section 2.4.
- It does not matter in what order you write each line of code, NIMBLE uses what is called a declarative language for building models. In brief, you write code that tells NIMBLE what you want to achieve, and not how to get there. In contrast, an imperative language requires that you write what you want your program to do step by step.
- You can think of models in NIMBLE as graphs as in Figure **??**. A graph is made of relations (or edges) that can be of two types. A stochastic relation is signaled by a ~ sign and defines a random variable in the model, such as `survived` or `theta`. A deterministic relation is signaled by a `<-` sign, like `lifespan`. Relations define nodes on the left - the children - in terms of other nodes on the right - the parents, and relations are directed edges from parents to children. Such graphs are called directed acyclic graph or DAG. banana-book_files/figure-latex/dag-survival-1.pdf

Second step in our workflow is to read in some data. We use a list in which each component corresponds to a known quantity in the model:

```
my.data <- list(released = 57, survived = 19)
```

You can proceed with data passed this way, but you should know a little more about how NIMBLE sees data. NIMBLE distinguishes data

[1] Cook LM, Brower LP, Croze HJ (1967) The accuracy of a population estimation from multiple recapture data. J Anim Ecol 36:57–60

and constants. Constants are values that do not change, e.g. vectors of known index values or the indices used to define for loops. Data are values that you might want to change, basically anything that only appears on the left of a ~. Declaring relevant values as constants is better for computational efficiency, but it is easy to forget, and fortunately NIMBLE will by itself distinguish data and constants. I will not use the distinction between data and constants in this chapter, but in the next chapters it will become important.

Third step is to tell NIMBLE which nodes in your model you would like to keep track of, in other words the quantities you'd like to do inference about. In our model we want survival `theta` and `lifespan`:

```
parameters.to.save <- c("theta", "lifespan")
```

In general you have many quantities in your model, including some of little interest that are not worth monitoring, and having full control on verbosity will prove handy.

Fourth step is to specify initial values for all model parameters. To make sure that the MCMC algorithm explores the posterior distribution, we start different chains with different parameter values. You can specify initial values for each chain in a list and put them in yet another list:

```
init1 <- list(theta = 0.1)
init2 <- list(theta = 0.5)
init3 <- list(theta = 0.9)
initial.values <- list(init1, init2, init3)
initial.values
## [[1]]
## [[1]]$theta
## [1] 0.1
##
##
## [[2]]
## [[2]]$theta
## [1] 0.5
```

```
##
##
## [[3]]
## [[3]]$theta
## [1] 0.9
```

Alternatively, you can write a simple R function that generates random initial values:

```
initial.values <- function() list(theta = runif(1,0,1))
initial.values()
## $theta
## [1] 0.1146
```

Firth and last step, you need to tell NIMBLE the number of chains to run, say `n.chain`, how long the burn-in period should be, say `n.burnin`, and the number of iterations following the burn-in period to be used for posterior inference. In NIMBLE, you specify the total number of iterations, say `n.iter`, so that the number of posterior samples per chain is `n.iter - n.burnin`. NIMBLE also allows discarding samples after burn-in, a procedure known as thinning, which I will not use in this book[2].

```
n.iter <- 5000
n.burnin <- 1000
n.chains <- 3
```

We now have all the ingredients to run model, that is to sample in the posterior distribution of model parameters using MCMC simulations. This is accomplished using function `nimbleMCMC()`:

```
mcmc.output <- nimbleMCMC(code = model,
                          data = my.data,
```

[2]Link, W.A. and Eaton, M.J. (2012), On thinning of chains in MCMC. Methods in Ecology and Evolution, 3: 112-115.

```
                        inits = initial.values,
                        monitors = parameters.to.save,
                        niter = n.iter,
                        nburnin = n.burnin,
                        nchains = n.chains)
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
```

NIMBLE goes through several steps that we will explain in Section 2.5. Function `nimbleMCMC()` takes other arguments that you might find useful. For example, you can suppress the progress bar if you find it too depressing when running long simulations with `progressBar = FALSE`. You can also get a summary of the outputs by specifying `summary = TRUE`. Check `?nimbleMCMC` for more details.

Now let's inspect what we have in `mcmc.output`:

```
str(mcmc.output)
## List of 3
##  $ chain1: num [1:4000, 1:2] 0.814 0.814 1.024 1.024 0.964 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "lifespan" "theta"
##  $ chain2: num [1:4000, 1:2] 0.852 0.852 0.801 0.902 0.902 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "lifespan" "theta"
##  $ chain3: num [1:4000, 1:2] 0.836 1.098 1.098 1.098 0.856 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "lifespan" "theta"
```

The R object `mcmc.output` is a list with three components, one for each MCMC chain. Let's have a look to `chain1` for example:

```
dim(mcmc.output$chain1)
## [1] 4000    2
head(mcmc.output$chain1)
##      lifespan  theta
## [1,]   0.8140 0.2928
## [2,]   0.8140 0.2928
## [3,]   1.0236 0.3764
## [4,]   1.0236 0.3764
## [5,]   0.9636 0.3542
## [6,]   0.9636 0.3542
```

Each component of the list is a matrix. In rows, you have 4000 samples from the posterior distribution of `theta`, which corresponds to `n.iter - n.burnin` iterations. In columns, you have the quantities we monitor, `theta` and `lifespan`. From there, you can compute the posterior mean of `theta`:

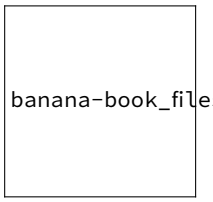
```
mean(mcmc.output$chain1[,'theta'])
## [1] 0.3388
```

You can also obtain the 95% credible interval for `theta`:

```
quantile(mcmc.output$chain1[,'theta'], probs = c(2.5, 97.5)/100)
##   2.5%  97.5%
## 0.2211 0.4659
```

Let's visualise the posterior distribution of `theta` with a histogram:

```
mcmc.output %>%
  as_tibble() %>%
  ggplot() +
  geom_histogram(aes(x = chain1[,"theta"]), color = "White") +
  labs(x = "survival probability")
```



There are less painful ways of doing posterior inference. In this book, I will use the R package `MCMCvis`[3] to summarise and visualize MCMC outputs, but there are other perfectly valid options out there like `ggmcmc`[4] and `basicMCMCplots`[5]. **Shall I demonstrate these other options?**

Let's load the package `MCMCvis`:

```
library(MCMCvis)
```

To get the most common numerical summaries, the function `MCMCsummary()` does the job:

```
MCMCsummary(object = mcmc.output, round = 2)
##          mean   sd 2.5%  50% 97.5% Rhat n.eff
## lifespan 0.94 0.16 0.67 0.92  1.32    1  2545
## theta    0.34 0.06 0.22 0.34  0.47    1  2675
```

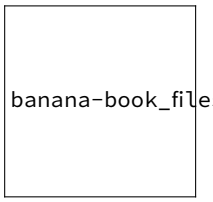
You can use a caterpillar plot to visualise the posterior distributions of `theta` with `MCMCplot()`:

```
MCMCplot(object = mcmc.output,
         params = 'theta')
```

[3] https://github.com/caseyyoungflesh/MCMCvis

[4] Fernández-i-Marín, X. (2016). ggmcmc: Analysis of MCMC Samples and Bayesian Inference. Journal of Statistical Software, 70(9), 1–20

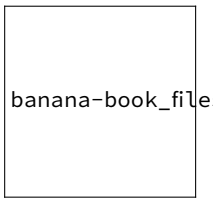
[5] https://cran.r-project.org/web/packages/basicMCMCplots/index.html



The point represents the posterior median, the thick line is the 50% credible interval and the thin line the 95% credible interval.

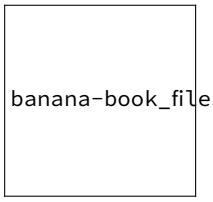
The trace and posterior density of theta can be obtained with `MCMCtrace()`:

```
MCMCtrace(object = mcmc.output,
          pdf = FALSE, # no export to PDF
          ind = TRUE, # separate density lines per chain
          params = "theta")
```



You can also add the diagnostics of convergence we discussed in the previous chapter:

```
MCMCtrace(object = mcmc.output,
          pdf = FALSE,
          ind = TRUE,
          Rhat = TRUE, # add Rhat
          n.eff = TRUE, # add eff sample size
          params = "theta")
```



We calculated lifespan directly in our model with `lifespan <- -1/log(theta)`. But you can also calculate this quantity from outside NIMBLE. This is a nice by-product of using MCMC simulations: you can obtain the posterior distribution of any quantity that is function of your model parameters by applying this function to samples from the posterior distribution of these parameters. In our example, all you need is samples from the posterior distribution of `theta`, which we pool between the three chains with:

```
theta_samples <- c(mcmc.output$chain1[,'theta'],
                   mcmc.output$chain2[,'theta'],
                   mcmc.output$chain3[,'theta'])
```

To get samples from the posterior distribution of lifespan, we apply the function to calculate lifespan to the samples from the posterior distribution of survival:

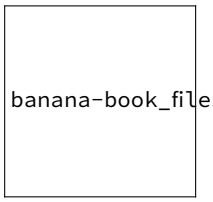
```
lifespan <- -1/log(theta_samples)
```

As usual then, you can calculate the posterior mean and 95% credible interval:

```
mean(lifespan)
## [1] 0.9385
quantile(lifespan, probs = c(2.5, 97.5)/100)
##   2.5%  97.5%
## 0.6684 1.3243
```

You can also visualise the posterior distribution of lifespan:

```
lifespan %>%
  as_tibble() %>%
  ggplot() +
  geom_histogram(aes(x = value), color = "white") +
  labs(x = "lifespan")
```



Now you're good to go. For convenience I have summarized the steps above in the box below. The NIMBLE workflow provided with `nimbleMCMC()` allows you to build models and make inference. This is what you can achieve with other software like WinBUGS or JAGS.

NIMBLE workflow:

```
# model building
model <- nimbleCode({
  # likelihood
  survived ~ dbinom(theta, released)
  # prior
  theta ~ dunif(0, 1)
  # derived quantity
  lifespan <- -1/log(theta)
})
# read in data
my.data <- list(released = 57, survived = 19)
# specify parameters to monitor
parameters.to.save <- c("theta", "lifespan")
# pick initial values
initial.values <- function() list(theta = runif(1,0,1))
# specify MCMC details
n.iter <- 5000
n.burnin <- 1000
n.chains <- 3
# run NIMBLE
mcmc.output <- nimbleMCMC(code = model,
                          data = my.data,
                          inits = initial.values,
                          monitors = parameters.to.save,
```

```
                          niter = n.iter,
                          nburnin = n.burnin,
                          nchains = n.chains)
# calculate numerical summaries
MCMCsummary(object = mcmc.output, round = 2)
# visualize parameter posterior distribution
MCMCplot(object = mcmc.output,
         params = 'theta')
# check convergence
MCMCtrace(object = mcmc.output,
          pdf = FALSE, # no export to PDF
          ind = TRUE, # separate density lines per chain
          params = "theta")
```

But NIMBLE is more than just another MCMC engine. It provides a programming environment so that you have full control when building models and estimating parameters. NIMBLE allows you to write your own functions and distributions to build models, and to choose alternative MCMC samplers or code new ones. This flexibility often comes with faster convergence.

I have to be honest, learning these improvements over other software takes some reading and experimentation, and it might well be that you do not need to use any of these features. And it's fine. In the next sections, I cover some of this advanced material. You may skip these sections and go back to this material later if you need it.

2.4 Programming

In NIMBLE you can write and use your own functions, or use existing R or C/C++ functions. This allows you to customize models the way you want.

2.4.1 NIMBLE functions

NIMBLE provides `nimbleFunctions` for programming. A `nimbleFunction` is like an R function, plus it can be compiled for faster computation. Going back to our animal survival example, we can write a `nimbleFunction` to compute lifespan:

```
computeLifespan <- nimbleFunction(
    run = function(theta = double(0)) { # type declarations
        ans <- -1/log(theta)
        return(ans)
        returnType(double(0))  # return type declaration
    } )
```

Within the nimbleFunction, the `run` section gives the function to be executed. It is written in the NIMBLE language. The `theta = double(0)` and `returnType(double(0))` arguments tell NIMBLE that the input and output are single numeric values (scalars). Alternatively, `double(1)` and `double(2)` are for vectors and matrices, while `logical()`, `integer()` and `character()` are for logical, integer and character values.

You can use your `nimbleFunction` in R:

```
computeLifespan(0.8)
## [1] 4.481
```

You can compile it and use the C++ code for faster computation:

```
CcomputeLifespan <- compileNimble(computeLifespan)
CcomputeLifespan(0.8)
## [1] 4.481
```

You can also use your `nimbleFunction` in a model:

```
model <- nimbleCode({
  # likelihood
  survived ~ dbinom(theta, released)
```

```
  # prior
  theta ~ dunif(0, 1)
  # derived quantity
  lifespan <- computeLifespan(theta)
})
```

The rest of the workflow remains the same:

```
my.data <- list(survived = 19, released = 57)
parameters.to.save <- c("theta", "lifespan")
initial.values <- function() list(theta = runif(1,0,1))
n.iter <- 5000
n.burnin <- 1000
n.chains <- 3
mcmc.output <- nimbleMCMC(code = model,
                          data = my.data,
                          inits = initial.values,
                          monitors = parameters.to.save,
                          niter = n.iter,
                          nburnin = n.burnin,
                          nchains = n.chains)
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
MCMCsummary(object = mcmc.output, round = 2)
##          mean   sd 2.5%  50% 97.5% Rhat n.eff
## lifespan 0.93 0.16 0.67 0.92  1.28    1  2756
## theta    0.34 0.06 0.22 0.34  0.46    1  2781
```

With `nimbleFunctions`, you can mimic basic R syntax, do linear algebra (e.g. compute eigenvalues), operate on vectors and matrices (e.g. inverse a matrix), use logical operators (e.g. and/or) and flow control

(e.g. if-else). There is also a long list of common and less common distributions that can be used with `nimbleFunctions`.

To learn everything you need to know on writing `nimbleFunctions`, make sure to read chapter 11 of the NIMBLE manual at https://r-nimble.org/html_manual/cha-RCfunctions.html#cha-RCfunctions.

2.4.2 Calling R/C++ functions

If you're like me, and too lazy to write your own functions, you can rely on the scientific community and use existing C, C++ or R code. The trick is to write a `nimbleFunction` that wraps access to that code which can then be used by NIMBLE. As an example, imagine you'd like to use an R function `myfunction()`, either a function you wrote yourself, or a function available in your favorite R package:

```
myfunction <- function(x) {
  -1/log(x)
}
```

Now wrap this function using `nimbleRcall()` or `nimbleExternalCall()` for a C or C++ function:

```
Rmyfunction <- nimbleRcall(prototype = function(x = double(0)){},
                           Rfun = 'myfunction',
                           returnType = double(0))
```

In the call to `nimbleRcall()` above, the argument `prototype` specifies inputs (a single numeric value `double(0)`) of the R function `Rfun` that generates outputs `returnType` (a single numeric value `double(0)`).

Now you can call your R function from a model (or any `nimbleFunctions`):

```
model <- nimbleCode({
  # likelihood
  survived ~ dbinom(theta, released)
  # prior
```

```
  theta ~ dunif(0, 1)
  lifespan <- Rmyfunction(theta)
})
```

The rest of the workflow remains the same:

```
my.data <- list(survived = 19, released = 57)
parameters.to.save <- c("theta", "lifespan")
initial.values <- function() list(theta = runif(1,0,1))
n.iter <- 5000
n.burnin <- 1000
n.chains <- 3
mcmc.output <- nimbleMCMC(code = model,
                          data = my.data,
                          inits = initial.values,
                          monitors = parameters.to.save,
                          niter = n.iter,
                          nburnin = n.burnin,
                          nchains = n.chains)
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
MCMCsummary(object = mcmc.output, round = 2)
##          mean   sd 2.5%  50% 97.5% Rhat n.eff
## lifespan 0.94 0.16 0.67 0.92  1.30    1  2549
## theta    0.34 0.06 0.23 0.34  0.46    1  2595
```

Evaluating an R function from within NIMBLE slows MCMC sampling down, but if you can live with it, the cost is easily offset by the convenience of being able to use existing R functions.

Another advantage of using `nimbleRcall()` (or `nimbleExternalCall()`) is that you can keep large objects out of your model, so that NIMBLE does not have to handle them in MCMC sampling. These objects should be

constants and not change when you run NIMBLE. Letting R manipulating these objects will save you time, usually more than the time you lose by calling R from within NIMBLE.

2.4.3 User-defined distributions

With `nimbleFunctions` you can provide user-defined distributions to NIMBLE. You need to write functions for density (`d`) and simulation (`r`) for your distribution. As an example, we write our own binomial distribution:

```
# density
dmybinom <- nimbleFunction(
  run = function(x = double(0),
                 size = double(0),
                 prob = double(0),
                 log = integer(0, default = 1)) {
    returnType(double(0))
    # compute binomial coefficient
    lchoose <- lfactorial(size) - lfactorial(x) - lfactorial(size - x)
    # binomial density function
    logProb <- lchoose + x * log(prob) + (size - x) * log(1 - prob)
    if(log) return(logProb)
    else return(exp(logProb))
  })
# simulation using the coin flip method (p. 524 in Devroye 1986)
rmybinom <- nimbleFunction(
  run = function(n = integer(0, default = 1),
                 size = double(0),
                 prob = double(0)) {
    returnType(double(0))
    x <- 0
    y <- runif(n = size, min = 0, max = 1)
    for (j in 1:size){
      if (y[j] < prob){
        x <- x + 1
      }else{
```

```
        x <- x
      }
    }
    return(x)
  })
```

You need to define the `nimbleFunctions` in R's global environment for them to be accessed:

```
assign('dmybinom', dmybinom, .GlobalEnv)
assign('rmybinom', rmybinom, .GlobalEnv)
```

You can try out your function and simulate a random value from a binomial distribution with size 5 and probability 0.1:

```
rmybinom(n = 1, size = 5, prob = 0.1)
## [1] 1
```

All set. You can run your workflow:

```
model <- nimbleCode({
 # likelihood
 survived ~ dmybinom(prob = theta, size = released)
 # prior
 theta ~ dunif(0, 1)
})
my.data <- list(released = 57, survived = 19)
initial.values <- function() list(theta = runif(1,0,1))
n.iter <- 5000
n.burnin <- 1000
n.chains <- 3
mcmc.output <- nimbleMCMC(code = model,
 data = my.data,
 inits = initial.values,
 niter = n.iter,
```

```
 nburnin = n.burnin,
 nchains = n.chains)
## Registering the following user-provided distributions: dmybinom
## NIMBLE has registered dmybinom as a distribution based on its use in BUGS code. Note that i
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
MCMCsummary(mcmc.output)
##         mean      sd   2.5%    50%  97.5% Rhat n.eff
## theta 0.3399 0.05976 0.2285 0.3367 0.4625    1  2720
```

Having `nimbleFunctions` offers infinite possibilities to customize your models and algorithms. Besides what we covered already, you can write your own samplers. We will see an example in a minute, but I first need to tell you more about the NIMBLE workflow.

2.5 Under the hood

So far, you have used `nimbleMCMC()` which runs the default MCMC workflow. This is perfecly fine for most applications. However, in some situations you need to customize the MCMC samplers to improve or fasten convergence. NIMBLE allows you to look under the hood by using a detailed workflow in several steps: `nimbleModel()`, `configureMCMC()`, `buildMCMC()`, `compileNimble()` and `runMCMC()`. Note that `nimbleMCMC()` does all of this at once.

We write the model code, read in data and pick initial values as before:

```
model <- nimbleCode({
  # likelihood
  survived ~ dbinom(theta, released)
```

```
  # prior
  theta ~ dunif(0, 1)
  # derived quantity
  lifespan <- -1/log(theta)
})
my.data <- list(survived = 19, released = 57)
initial.values <- list(theta = 0.5)
```

First step is to create the model as an R object (uncompiled model) with `nimbleModel()`:

```
survival <- nimbleModel(code = model,
                        data = my.data,
                        inits = initial.values)
```

You can look at its nodes:

```
survival$getNodeNames()
## [1] "theta"    "lifespan" "survived"
```

You can look at the values stored at each node:

```
survival$theta
## [1] 0.5
survival$survived
## [1] 19
survival$lifespan
## [1] 1.443
# this is -1/log(0.5)
```

We can also calculate the log-likelihood at the initial value for `theta`:

```
survival$calculate()
## [1] -5.422
# this is dbinom(x = 19, size = 57, prob = 0.5, log = TRUE)
```

The ability in NIMBLE to access the nodes of your model and to evaluate the model likelihood can help you in identifying bugs in your code. **Give example? Provide negative initial value for theta, or released in data < survived.**

You can obtain the graph of the model as in Figure **??** with:

```
survival$plotGraph()
```

Second we compile the model with `compileNimble()`:

```
Csurvival <- compileNimble(survival)
```

With `compileNimble()`, the C++ code is generated, compiled and loaded back into R so that it can be used in R (compiled model):

```
Csurvival$theta
## [1] 0.5
```

Now you have two versions of the model, `survival` is in R and `Csurvival` in C++. Being able to separate the steps of model building and parameter estimation is a strength of NIMBLE. This gives you a lot of flexibility at both steps. For example, imagine you would like to fit your model with maximum likelihood, then you can do it by wrapping your model in an R function that gets the likelihood and maximise this function. Using the C version of the model, you can write:

```
# function for negative log-likelihood to minimize
f <- function(par) {
    Csurvival[['theta']] <- par # assign par to theta
```

```
    ll <- Csurvival$calculate() # update log-likelihood with par value
    return(-ll) # return negative log-likelihood
}
# evaluate function at 0.5 and 0.9
f(0.5)
## [1] 5.422
f(0.9)
## [1] 55.41
# minimize function
out <- optimize(f, interval = c(0,1))
round(out$minimum, 2)
## [1] 0.33
```

By maximising the likelihood (or minimising the negative log-likelihood), you obtain the maximum likelihood estimate of animal survival, which is exactly 19 surviving animals over 57 released animals or 0.33.

Third we create a MCMC configuration for our model with `configureMCMC()`:

```
survivalConf <- configureMCMC(survival)
## ===== Monitors =====
## thin = 1: theta
## ===== Samplers =====
## RW sampler (1)
##   - theta
```

This steps tells you the nodes that are monitored by default, and the MCMC samplers than have been assigned to them. Here `theta` is monitored, and samples from its posterior distribution are simulated with a random walk sampler similar to the Metropolis sampler we coded in the previous chapter in Section 1.5.3.

To monitor `lifespan` in addition to `theta`, you write:

```
survivalConf$addMonitors(c("lifespan"))
## thin = 1: theta, lifespan
survivalConf
## ===== Monitors =====
## thin = 1: theta, lifespan
## ===== Samplers =====
## RW sampler (1)
##   - theta
```

Third, we create a MCMC function with `buildMCMC()` and compile it with `compileNimble()`:

```
survivalMCMC <- buildMCMC(survivalConf)
CsurvivalMCMC <- compileNimble(survivalMCMC, project = survival)
```

Note that models and `nimbleFunctions` need to be compiled before they can be used to specify a project.

Fourth, we run NIMBLE with `runMCMC()`:

```
n.iter <- 5000
n.burnin <- 1000
samples <- runMCMC(mcmc = CsurvivalMCMC,
                   niter = n.iter,
                   nburnin = n.burnin)
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
```

We run a single chain but `runMCMC()` allows you to use multiple chains as with `nimbleMCMC()`.

You can look into `samples` which contains values simulated from the posterior distribution of the parameters we monitor:

```
head(samples)
##      lifespan  theta
```

```
## [1,]   1.1031 0.4039
## [2,]   1.0390 0.3820
## [3,]   0.7976 0.2854
## [4,]   0.9425 0.3461
## [5,]   0.9425 0.3461
## [6,]   0.9425 0.3461
```

From here, you can obtain numerical summaries with `samplesSummary()`:

```
samplesSummary(samples)
##            Mean Median St.Dev. 95%CI_low 95%CI_upp
## lifespan 0.9342 0.9176 0.15934    0.6682    1.2794
## theta    0.3381 0.3363 0.06099    0.2239    0.4577
```

I have summarized the steps above in the box below.

Detailed NIMBLE workflow:

```
# model building
model <- nimbleCode({
  # likelihood
  survived ~ dbinom(theta, released)
  # prior
  theta ~ dunif(0, 1)
  # derived quantity
  lifespan <- -1/log(theta)
})
# read in data
my.data <- list(released = 57, survived = 19)
# pick initial values
initial.values <- function() list(theta = runif(1,0,1))
# create model as an R object (uncompiled model)
survival <- nimbleModel(code = model,
                        data = my.data,
                        inits = initial.values())
# compile model
```

```
Csurvival <- compileNimble(survival)
# create a MCMC configuration
survivalConf <- configureMCMC(survival)
# add lifespan to list of parameters to monitor
survivalConf$addMonitors(c("lifespan"))
# create a MCMC function and compile it
survivalMCMC <- buildMCMC(survivalConf)
CsurvivalMCMC <- compileNimble(survivalMCMC, project = survival)
# specify MCMC details
n.iter <- 5000
n.burnin <- 1000
n.chains <- 2
# run NIMBLE
samples <- runMCMC(mcmc = CsurvivalMCMC,
                   niter = n.iter,
                   nburnin = n.burnin,
                   nchain = n.chains)
# calculate numerical summaries
MCMCsummary(object = samples, round = 2)
# visualize parameter posterior distribution
MCMCplot(object = samples,
         params = 'theta')
# check convergence
MCMCtrace(object = samples,
          pdf = FALSE, # no export to PDF
          ind = TRUE, # separate density lines per chain
          params = "theta")
```

At first glance, using several steps instead of doing all these at once with `nimbleMCMC()` seems odds. Why is it useful? Mastering the whole sequence of steps allows you to play around with samplers, by changing the samplers NIMBLE picks by default, or even writing your own samplers.

2.6 MCMC samplers

2.6.1 Default samplers

What is the default sampler used by NIMBLE in our example? You can answer this question by inspecting the MCMC configuration obtained with `configureMCMC()`:

```
#survivalConf <- configureMCMC(survival)
survivalConf$printSamplers()
## [1] RW sampler: theta
```

Now that we have control on the MCMC configuration, let's mess it up. We start by removing the default sampler:

```
survivalConf$removeSamplers(c('theta'))
survivalConf$printSamplers()
```

And we change it for a slice sampler:

```
survivalConf$addSampler(target = c('theta'),
                        type = 'slice')
survivalConf$printSamplers()
## [1] slice sampler: theta
```

Now you can resume the workflow:

```
# create a new MCMC function and compile it:
survivalMCMC2 <- buildMCMC(survivalConf)
CsurvivalMCMC2 <- compileNimble(survivalMCMC2,
                                project = survival,
                                resetFunctions = TRUE) # to compile new functions
                                                       # into existing project,
                                                       # need to reset nimbleFunctions

# run NIMBLE:
```

```
samples2 <- runMCMC(mcmc = CsurvivalMCMC2,
                    niter = n.iter,
                    nburnin = n.burnin)
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
# obtain numerical summaries:
samplesSummary(samples2)
##            Mean Median St.Dev. 95%CI_low 95%CI_upp
## lifespan 0.9361 0.9238  0.1583    0.6682    1.2826
## theta    0.3389 0.3388  0.0604    0.2239    0.4586
```

NIMBLE implements many samplers, and a list is available with `?samplers`. For example, high correlation in (regression) parameters can make independent samplers inefficient. In that situation, block sampling might help which consists in proposing candidate values from a multivariate distribution that acknowledges correlation between parameters. **Say something on how default samplers are chosen by NIMBLE?**

2.6.2 User-defined samplers

Allowing you to code your own sampler is another topic on which NIMBLE thrives. As an example, we focus on the Metropolis algorithm of Section 1.5.3 which we coded in R. In this section, we make it a `nimbleFunction` so that we can use it within our model:

```
my_metropolis <- nimbleFunction(
  name = 'my_metropolis', # fancy name for our MCMC sampler
  contains = sampler_BASE,
  setup = function(model, mvSaved, target, control) {
    # i) get dependencies for 'target' in 'model'
    calcNodes <- model$getDependencies(target)
    # ii) get sd of proposal distribution
    scale <- control$scale
  },
  run = function() {
```

```
    # (1) log-lik at current value
    initialLP <- model$getLogProb(calcNodes)
    # (2) current parameter value
    current <- model[[target]]
    # (3) logit transform
    lcurrent <- log(current / (1 - current))
    # (4) propose candidate value
    lproposal <- lcurrent  + rnorm(1, mean = 0, scale)
    # (5) back-transform
    proposal <- plogis(lproposal)
    # (6) plug candidate value in model
    model[[target]] <<- proposal
    # (7) log-lik at candidate value
    proposalLP <- model$calculate(calcNodes)
    # (8) compute lik ratio on log scale
    lMHR <- proposalLP - initialLP
    # (9) spin continuous spinner and compare to ratio
    if(runif(1,0,1) < exp(lMHR)) {
      # (10) if candidate value is accepted, update current value
      copy(from = model, to = mvSaved, nodes = calcNodes, logProb = TRUE, row = 1)
    } else {
      ## (11) if candidate value is accepted, keep current value
      copy(from = mvSaved, to = model, nodes = calcNodes, logProb = TRUE, row = 1)
    }
  },
  methods = list(
    reset = function() {}
  )
)
```

Compared to `nimbleFunctions` we wrote earlier, `my_metropolis()` contains a `setup` function which i) gets the dependencies of the parameter to update in the `run` function with Metropolis, the target node, that would be `theta` in our example and ii) extracts control parameters, that would be `scale` the standard deviation of the proposal distribution in our example. Then the `run` function implements the steps of the

Metropolis algorithm: (1) get the log-likelihood function evaluated at the current value, (2) get the current value, (3) apply the logit transform to it, (4) propose a candidate value by perturbing the current value with some normal noise controled by the standard deviation `scale`, (5) back-transform the candidate value and (6) plug it in the model, (7) calculate the log-likelihood function at the candidate value, (8) compute the Metropolis ratio on the log scale, (9) compare output of a spinner and the Metropolis ratio to decide whether to (10) accept the candidate value and copy from the model to `mvSaved` or (11) reject it and keep the current value by copying from `mvSaved` to the model. Because this `nimbleFunction` is to be used as a MCMC sampler, several constraints need to be respected like having a `contains = sampler_BASE` statement or using the four arguments `model`, `mvSaved`, `target` and `control` in the `setup` function. Of course, NIMBLE implements a more advanced and efficient version of the Metropolis algorithm, you can look into it at https://github.com/cran/nimble/blob/master/R/MCMC_samplers.R#L184.

Now that we have our user-defined MCMC algorithm, we can change the default sampler for our new sampler as in Section 2.6.1. We start from scratch:

```
model <- nimbleCode({
  # likelihood
  survived ~ dbinom(theta, released)
  # prior
  theta ~ dunif(0, 1)
})
my.data <- list(survived = 19, released = 57)
initial.values <- function() list(theta = runif(1,0,1))
survival <- nimbleModel(code = model,
                        data = my.data,
                        inits = initial.values())
Csurvival <- compileNimble(survival)
survivalConf <- configureMCMC(survival)
## ===== Monitors =====
## thin = 1: theta
## ===== Samplers =====
```

```
## RW sampler (1)
##   - theta
```

We print the samplers used by default, remove the default sampler for `theta`, replace it with our `my_metropolis()` sampler with the standard deviation of the proposal distribution set to 0.1, and print again to make sure NIMBLE now uses our new sampler:

```
survivalConf$printSamplers()
## [1] RW sampler: theta
survivalConf$removeSamplers(c('theta'))
survivalConf$addSampler(target = 'theta',
                        type = 'my_metropolis',
                        control = list(scale = 0.1)) # standard deviation
                                                     # of proposal distribution
survivalConf$printSamplers()
## [1] my_metropolis sampler: theta,  scale: 0.10000000000000001
```

The rest of the workflow is unchanged:

```
survivalMCMC <- buildMCMC(survivalConf)
CsurvivalMCMC <- compileNimble(survivalMCMC,
                               project = survival)
samples <- runMCMC(mcmc = CsurvivalMCMC,
                   niter = 5000,
                   nburnin = 1000)
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
samplesSummary(samples)
##         Mean Median St.Dev. 95%CI_low 95%CI_upp
## theta 0.3347 0.3306 0.06278    0.2291    0.4644
```

You can re-run the analysis by setting the standard deviation of the proposal to different values, say 1 and 10, and compare Figure 2.2 to traceplots we obtained with our R implementation of the Metropolis algorithm in the previous chapter at Figure 1.12:

FIGURE 2.2: Trace plots for different values of the standard deviation (scale) of the proposal distribution.

2.7 Tips and tricks

Before closing this chapter on NIMBLE, I thought it'd be useful to have a section gathering a few tips and tricks that would make your life easier. **These are my tips and tricks, NIMBLE users, I'd be happy to hear yours: email me[6], edit the chapter[7] or file an issue[8] on GitHub.**

2.7.1 Precision vs standard deviation

In other sotware like JAGS, the normal distribution is parameterized with mean `mu` and a parameter called precision, often denoted `tau`, the inverse of the variance you are used to. Say we use a normal prior on some parameter `epsilon` with `epsilon ~ dnorm(mu, tau)`. We'd like this prior to be vague, therefore `tau` should be small, say 0.01 so that the variance of the normal distribution is large, 1/0.01 = 100 here. This subtlety is the source of problems (and frustration) when you forget that the second parameter is precision and use `epsilon ~ dnorm(mu, 100)`, because then the variance is actually 1/100 = 0.01 and the prior is very informative, and peaked on `mu`. In NIMBLE you can use this parameterisation as well as the more natural parameterisation `epsilon ~ dnorm(mu, sd = 100)` which avoids confusion.

2.7.2 Indexing

NIMBLE does not guess the dimensions of objects. In other software like JAGS you can write `sum.x <- sum(x[])` to calculate the sum over all components of `x`. In NIMBLE you need to write `sum.x <- sum(x[1:n])` to sum the components of `x` from 1 up to n. Specifying dimensions can

[6] mailto:olivier.gimenez@cefe.cnrs.fr

[7] https://github.com/oliviergimenez/banana-book/edit/master/nimble.Rmd

[8] https://github.com/oliviergimenez/banana-book/issues

be annoying, but I find it useful as it forces me to think of what I am doing and to keep my code self-explaining.

2.7.3 Faster compilation

You might have noticed that compilation in NIMBLE takes time. When you have large models (with lots of nodes), compilation can take forever. You can set `calculate = FALSE` in `nimbleModel()` to disable the calculation of all deterministic nodes and log-likelihood. You can also use `useConjugacy = FALSE` in `configureMCMC()` to disable the search for conjugate samplers. With the animal survival example, you would do:

```
model <- nimbleCode({
  # likelihood
  survived ~ dbinom(theta, released)
  # prior
  theta ~ dunif(0, 1)
})
my.data <- list(survived = 19, released = 57)
initial.values <- function() list(theta = runif(1,0,1))
survival <- nimbleModel(code = model,
                        data = my.data,
                        inits = initial.values(),
                        calculate = FALSE) # first tip
Csurvival <- compileNimble(survival)
survivalConf <- configureMCMC(survival)
## ===== Monitors =====
## thin = 1: theta
## ===== Samplers =====
## RW sampler (1)
##   - theta
survivalMCMC <- buildMCMC(survivalConf, useConjugacy = FALSE) # second tip
CsurvivalMCMC <- compileNimble(survivalMCMC,
                               project = survival)
samples <- runMCMC(mcmc = CsurvivalMCMC,
                   niter = 5000,
                   nburnin = 1000)
```

```
## |-------------|-------------|-------------|-------------|
## |-----------------------------------------------------|
samplesSummary(samples)
##         Mean Median St.Dev. 95%CI_low 95%CI_upp
## theta 0.3391 0.3382 0.06244      0.22    0.4691
```

2.7.4 Updating MCMC chains

Sometimes it is useful to run your MCMC chains a little bit longer to improve convergence. Re-starting from the run in previous section, you can use:

```
niter_ad <- 6000
CsurvivalMCMC$run(niter_ad, reset = FALSE)
## |-------------|-------------|-------------|-------------|
## |-----------------------------------------------------|
## NULL
```

Then you can extract the matrix of previous MCMC samples augmented with new ones and obtain numerical summaries:

```
more_samples <- as.matrix(CsurvivalMCMC$mvSamples)
samplesSummary(more_samples)
##         Mean Median St.Dev. 95%CI_low 95%CI_upp
## theta 0.3409  0.339 0.06097    0.2267     0.469
```

You can check that `more_samples` contains 10000 samples, 4000 from the call to `runMCMC()` plus 6000 additional samples.

2.7.5 Reproducibility

If you want your results to be reproducible, you can control the state of R the random number generator with the `setSeed` argument in functions `nimbleMCMC()` and `runMCMC()`. Going back to the animal survival example, you can check that two calls to `nimbleMCMC()` give the same results when `setSeed` is set to the same value:

```
# first call to nimbleMCMC()
mcmc.output1 <- nimbleMCMC(code = model,
                           data = my.data,
                           inits = initial.values,
                           niter = 5000,
                           nburnin = 1000,
                           nchains = 3,
                           summary = TRUE,
                           setSeed = 123)
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
# second call to nimbleMCMC()
mcmc.output2 <- nimbleMCMC(code = model,
                           data = my.data,
                           inits = initial.values,
                           niter = 5000,
                           nburnin = 1000,
                           nchains = 3,
                           summary = TRUE,
                           setSeed = 123)
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
# outputs from both calls are the same
mcmc.output1$summary$all.chains
##         Mean Median St.Dev. 95%CI_low 95%CI_upp
## theta 0.3387  0.336 0.05968    0.2282    0.4608
mcmc.output2$summary$all.chains
```

```
##         Mean Median St.Dev. 95%CI_low 95%CI_upp
## theta 0.3387  0.336 0.05968    0.2282    0.4608
```

2.7.6 Parallelization

To speed up your analyses, you can run MCMC chains in parallel. This is what the package `jagsUI`[9] accomplishes for JAGS users. Here, we use the `parallel` package for parallel computation:

```
library(parallel)
```

First you create a cluster using the total amount of cores you have but one to make sure your computer can go on working:

```
nbcores <- detectCores() - 1
my_cluster <- makeCluster(nbcores)
```

Then you wrap your workflow in a function to be run in parallel:

```
workflow <- function(seed, data) {

  library(nimble)

  model <- nimbleCode({
    # likelihood
    survived ~ dbinom(theta, released)
    # prior
    theta ~ dunif(0, 1)
  })

  set.seed(123) # for reproducibility
  initial.values <- function() list(theta = runif(1,0,1))
```

[9] https://github.com/kenkellner/jagsUI

```
  survival <- nimbleModel(code = model,
                          data = data,
                          inits = initial.values())
  Csurvival <- compileNimble(survival)
  survivalMCMC <- buildMCMC(Csurvival)
  CsurvivalMCMC <- compileNimble(survivalMCMC)

  samples <- runMCMC(mcmc = CsurvivalMCMC,
                     niter = 5000,
                     nburnin = 1000,
                     setSeed = seed)

  return(samples)
}
```

Now we run the code using `parLapply()`, which uses cluster nodes to execute our workflow:

```
output <- parLapply(cl = my_cluster,
                    X = c(2022, 666),
                    fun = workflow,
                    data = list(survived = 19, released = 57))
```

In the call to `parLapply`, we specify `X = c(2022, 666)` to ensure reproducibility. We use two alues 2022 and 666 to set the seed in `workflow()`, which means we run two instances of our workflow, or two MCMC chains. Note that we also have a line `set.seed(123)` in the `workflow()` function to ensure reproducibility while drawing randomly initial values.

It's good practice to close the cluster with `stopCluster()` so that processes do not continue to run in the background and slow down other processes:

```
stopCluster(my_cluster)
```

By inspecting the results, you can see that the object `output` is a list with two components, one for each MCMC chain:

```
str(output)
## List of 2
##  $ : num [1:4000, 1] 0.393 0.369 0.346 0.346 0.346 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr "theta"
##  $ : num [1:4000, 1] 0.435 0.435 0.435 0.435 0.243 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr "theta"
```

Eventually, you can obtain numerical summaries:

```
MCMCsummary(output)
##         mean      sd   2.5%    50%  97.5% Rhat n.eff
## theta 0.3361 0.06148 0.2215 0.3335 0.4594    1  1779
```

2.7.7 Incomplete initialization

When you run `nimbleMCMC()` or `nimbleModel()`, you may get warnings thrown at you by NIMBLE like 'This model is not fully initialized' or 'value is NA or NaN even after trying to calculate'. This is not necessarily an error, but it 'reflects missing values in model variables' (incomplete initialization). In this situation, NIMBLE will initialize nodes with NAs by drawing from priors, and it will work or not. When possible, I try to initialize all nodes (full initialization). The process can be a bit of a headache, but it helps understanding the model structure better. Going back to our animal survival example, let's purposedly forget to provide an initial value for `theta`:

```
model <- nimbleCode({
  # likelihood
  survived ~ dbinom(theta, released)
```

```
  # prior
  theta ~ dunif(0, 1)
})
#initial.values <- list(theta = runif(1,0,1))
survival <- nimbleModel(code = model,
                        data = list(survived = 19, released = 57))
```

To see which variables are not initialized, we use `initializeInfo()`:

```
# survival$calculate() # gives NA
survival$initializeInfo()
```

Now that we know `theta` was not initialized, we can fix the issue and resume our workflow:

```
survival$theta <- 0.5 # assign initial value to theta
survival$calculate()
## [1] -5.422

Csurvival <- compileNimble(survival)
survivalMCMC <- buildMCMC(Csurvival)
## ===== Monitors =====
## thin = 1: theta
## ===== Samplers =====
## RW sampler (1)
##   - theta
CsurvivalMCMC <- compileNimble(survivalMCMC)

samples <- runMCMC(mcmc = CsurvivalMCMC,
                   niter = 5000,
                   nburnin = 1000)
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|

samplesSummary(samples)
```

```
##         Mean Median St.Dev. 95%CI_low 95%CI_upp
## theta 0.3359 0.3335 0.06088    0.2191    0.4602
```

2.7.8 Vectorization

Vectorization is the process of replacing a loop by a vector so that instead of processing a single value at a time, you process a set of values at once. As an example, instead of writing:

```
for(i in 1:n){
  x[i] <- mu + epsilon[i]
}
```

you would write:

```
x[1:n] <- mu + epsilon[1:n]
```

Vectorization can make your code more efficient by manipulating one vector node `x[1:n]` instead of `n` nodes `x[1]`, ..., `x[n]`. **Think of an example in relation to animal survival? Illustrate with vectorized Bernoulli or vectorized Binomial[10]?**

2.8 Summary

- NIMBLE is an R package that implements for you MCMC algorithms to generate samples from the posterior distribution of model parameters. You only have to specify a likelihood and priors using the BUGS language to apply the Bayes theorem.
- NIMBLE is more than just another MCMC engine. It provides a programming environment so that you have full control when building models and estimating parameters.

[10] https://github.com/nimble-dev/nimbleSCR/blob/master/nimbleSCR/R/dbinom_vector.R

- At the core of NIMBLE are `nimbleFunctions` which you can write and compile for faster computation. With `nimbleFunctions` you can mimic basic R syntax, work with vectors and matrices, use logical operators and flow control, and specify many distributions.
- There are two workflows to run NIMBLE. In most situations, `nimbleMCMC()` will serve you well. When you need more control, you can adopt a detailed workflow with `nimbleModel()`, `configureMCMC()`, `buildMCMC()`, `compileNimble()` and `runMCMC()`.
- By having full control of the workflow, you can change default MCMC samplers and even write your own samplers.

2.9 Suggested reading

In this chapter, I have only scratched the surface of what NIMBLE is capable of. Below is a list of pointers that should help you going further with NIMBLE.

- The NIMBLE folks make a lot of useful resources available through the official website https://r-nimble.org.
- The NIMBLE manual https://r-nimble.org/html_manual/cha-welcome-nimble.html reads like a book with clear explanations and relevant examples.
- You can learn a lot by going through examples at https://r-nimble.org/examples and training material from NIMBLE workshops at https://github.com/nimble-training.
- You can keep the NIMBLE cheatsheet https://r-nimble.org/cheatsheets/NimbleCheatSheet.pdf near you to remind yourself of the workflow, how to write and use models, or which functions and distributions are available.
- The motivation to write this book comes from a workshop I co-teach with colleagues, including Perry de Valpine and Daniel Turek from the NIMBLE development team. The material (slides and videos) is

available at https://github.com/oliviergimenez/bayesian-cr-workshop.

- If you have questions, feel free to get in touch with the community of NIMBLE users by emailing the discussion group https://groups.google.com/forum/#!forum/nimble-users. This is a great place to learn, and folks who take the time to answer questions are kind and provide constructive answers. When possible, make sure to provide a reproducible example illustrating your problem.
- Last, you can cite the following reference when using NIMBLE in a publication:

---

de Valpine, P., D. Turek, C. J. Paciorek, C. Anderson-Bergman, D. Temple Lang, and R. Bodik (2017). Programming With Models: Writing Statistical Algorithms for General Model Structures With NIMBLE[11]. *Journal of Computational and Graphical Statistics* **26** (2): 403–13.

---

[11] https://arxiv.org/pdf/1505.05093.pdf

3

Hidden Markov models

3.1 Introduction

In this third chapter, you will learn the basics on Markov models and how to fit them to longitudinal data using NIMBLE. In real life however, individuals may go undetected and their status be unknown. You will also learn how to manipulate the extension of Markov models to hidden states, so-called hidden Markov models.

3.2 Longitudinal data

Let's get back to our survival example, and denote z_i the state of individual i with $z_i = 1$ if alive and $z_i = 0$ if dead. We have a total of $z = \sum_{i=1}^{n} z_i$ survivors out of n released animals with winter survival probability ϕ. Our model so far is a combination of a binomial likelihood and a Beta prior with parameters 1 and 1, which is also a uniform distribution between 0 and 1. It can be written as[1]:

$$
\begin{aligned}
z &\sim \text{Binomial}(n, \phi) && \text{[likelihood]} \\
\phi &\sim \text{Beta}(1, 1) && \text{[prior for } \phi\text{]}
\end{aligned}
$$

[1] I write models the way Richard McElreath does it in his book and video lectures Statistical Rethinking[2].

Because the binomial distribution is just a sum of independent Bernoulli outcomes, you can rewrite this model as:

$$
\begin{aligned}
z_i &\sim \text{Bernoulli}(\phi),\ i = 1, \ldots, N && \text{[likelihood]} \\
\phi &\sim \text{Beta}(1, 1) && \text{[prior for } \phi\text{]}
\end{aligned}
$$

It is like flipping a coin for each individual and get a survivor with probability ϕ.

In this set up, we consider a single winter. But for many species, we need to collect data on the long term to get a representative estimate of survival. Therefore what if we had say $T = 5$ winters?

Let us denote $z_{i,t} = 1$ if individual i alive at winter t, and $z_{i,t} = 2$ if dead. Then longitudinal data look like in the table below. Each row is an individual i, and columns are for winters t, or sampling occasions. Variable z is indexed by both i and t, and takes value 1 if individual i is alive in winter t, and 2 otherwise.

id	winter 1	winter 2	winter 3	winter 4	winter 5
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	1	1	1
4	1	1	1	1	2
5	1	1	1	1	1
6	1	1	2	2	2
7	1	1	1	1	1
8	1	2	2	2	2
9	1	1	1	1	2
10	1	2	2	2	2
11	1	1	1	1	1
12	1	1	1	1	1
13	1	1	1	2	2
14	1	1	1	1	1
15	1	2	2	2	2
16	1	1	1	1	1
17	1	1	1	1	1
18	1	1	1	1	2
19	1	1	1	1	1
20	1	1	2	2	2
21	1	1	2	2	2
22	1	2	2	2	2
23	1	1	1	1	2
24	1	1	1	1	2
25	1	1	1	1	1
26	1	1	1	1	2
27	1	1	1	1	2
28	1	2	2	2	2
29	1	1	2	2	2
30	1	1	2	2	2
31	1	2	2	2	2
32	1	2	2	2	2
33	1	2	2	2	2
34	1	1	2	2	2
35	1	1	2	2	2
36	1	2	2	2	2
37	1	1	1	1	1
38	1	1	1	2	2
39	1	1	1	1	1
40	1	1	1	1	2
41	1	1	1	1	1
42	1	1	1	2	2
43	1	1	1	1	1

3.3 A Markov model for longitudinal data

Let's think of a model for these data. The objective remains the same, estimating survival. To build this model, we'll make assumptions, go through its components and write down its likelihood. Note that we already encountered Markov models in Section 1.5.2.

3.3.1 Assumptions

First, we assume that the state of an animal in a given winter, alive or dead, is only dependent on its state the winter before. In other words, the future depends only on the present, not the past. This is a Markov process.

Second, if an animal is alive in a given winter, the probability it survives to the next winter is ϕ. The probability it dies is $1 - \phi$.

Third, if an animal is dead a winter, it remains dead, unless you believe in zombies.

Our Markov process can be represented this way:

An example of this Markov process is, for example:

Here the animal remains alive over the first two time intervals ($z_{i,1} = z_{i,2} = z_{i,3} = 1$) with probability ϕ until it dies over the fourth time

interval ($z_{i,4} = 2$) with probability $1 - \phi$ then remains dead from then onwards ($z_{i,5} = 2$) with probability 1.

3.3.2 Transition matrix

You might have figured it out already (if not, not a problem), the core of our Markov process is made of transition probabilities between states alive and dead. For example, the probability of transitioning from state alive at $t - 1$ to state alive at t is $\Pr(z_{i,t} = 1|z_{i,t-1} = 1) = \gamma_{1,1}$. It is the survival probability ϕ. The probability of dying over the interval $(t-1, t)$ is $\Pr(z_{i,t} = 2|z_{i,t-1} = 1) = \gamma_{1,2} = 1 - \phi$. Now if an animal is dead at $t - 1$, then $\Pr(z_t = 1|z_{t-1} = 2) = 0$ and $\Pr(z_{i,t} = 2|z_{i,t-1} = 2) = 1$.

We can gather these probabilities of transition between states from one occasion to the next in a matrix, say $\mathbf{\Gamma}$, which we will call the transition matrix:

$$\mathbf{\Gamma} = \begin{pmatrix} \gamma_{1,1} & \gamma_{1,2} \\ \gamma_{2,1} & \gamma_{2,2} \end{pmatrix} = \begin{pmatrix} \phi & 1-\phi \\ 0 & 1 \end{pmatrix}$$

To try and remember that the states at $t - 1$ are in rows, and the states at t are in columns, I will often write:

$$\mathbf{\Gamma} = \begin{array}{c} \begin{array}{cc} z_t = 1 & z_t = 2 \end{array} \\ \begin{pmatrix} \phi & 1-\phi \\ 0 & 1 \end{pmatrix} \end{array} \begin{array}{l} z_{t-1} = 1 \text{ (alive)} \\ z_{t-1} = 2 \text{ (dead)} \end{array}$$

Take the time you need to navigate through this matrix, and get familiar with it. For example, you may start alive at t (first row) then end up dead at $t + 1$ (first column) with probability $1 - \phi$.

3.3.3 Initial states

A Markov process has to start somewhere. We need the probabilities of initial states, i.e. the states of an individual at $t = 1$. We will gather the probability of being in each state (alive or 1 and dead or 2) in the first winter in a vector. We will use $\delta = (\Pr(z_{i,1} = 1), \Pr(z_{i,1} = 2))$. For simplicity, we will assume that all individuals are marked and released

in the first winter, hence alive when first captured, which means that they are all in state alive or 1 for sure. Therefore we have $\delta = (1, 0)$.

3.3.4 Likelihood

Now that we have built a Markov model, we need its likelihood to apply the Bayes theorem. The likelihood is the probability of the data, given the model. Here the data are the z, therefore we need $\Pr(\mathbf{z}) = \Pr(z_1, z_2, \ldots, z_{T-2}, z_{T-1}, z_T)$.

We're gonna work backward, starting from the last sampling occasion. Using conditional probabilities, the likelihood can be written as the product of the probability of z_T i.e. you're alive or not on the last occasion given your past history, that is the states at previous occasions, times the probability of your past history:

$$\begin{aligned}\Pr(\mathbf{z}) &= \Pr(z_T, z_{T-1}, z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T | z_{T-1}, z_{T-2}, \ldots, z_1) \Pr(z_{T-1}, z_{T-2}, \ldots, z_1)\end{aligned}$$

Then because we have a Markov model, we're memory less, that is the probabilty of next state, here z_T, depends only on the current state, that is z_{T-1}, and not the previous states:

$$\begin{aligned}\Pr(\mathbf{z}) &= \Pr(z_T, z_{T-1}, z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T | z_{T-1}, z_{T-2}, \ldots, z_1) \Pr(z_{T-1}, z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T | z_{T-1}) \Pr(z_{T-1}, z_{T-2}, \ldots, z_1)\end{aligned}$$

You can apply the same reasoning to $T - 1$. First use conditional probabilities:

$$\begin{aligned}\Pr(\mathbf{z}) &= \Pr(z_T, z_{T-1}, z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T|z_{T-1}, z_{T-2}, \ldots, z_1)\Pr(z_{T-1}, z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T|z_{T-1})\Pr(z_{T-1}, z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T|z_{T-1})\Pr(z_{T-1}|z_{T-2}, \ldots, z_1)\Pr(z_{T-2}, \ldots, z_1)\end{aligned}$$

Then apply the Markovian property:

$$\begin{aligned}\Pr(\mathbf{z}) &= \Pr(z_T, z_{T-1}, z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T|z_{T-1}, z_{T-2}, \ldots, z_1)\Pr(z_{T-1}, z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T|z_{T-1})\Pr(z_{T-1}, z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T|z_{T-1})\Pr(z_{T-1}|z_{T-2}, \ldots, z_1)\Pr(z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T|z_{T-1})\Pr(z_{T-1}|z_{T-2})\Pr(z_{T-2}, \ldots, z_1)\end{aligned}$$

And so on up to z_2. You end up with this expression for the likelihood:

$$\begin{aligned}\Pr(\mathbf{z}) &= \Pr(z_T, z_{T-1}, z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T|z_{T-1}, z_{T-2}, \ldots, z_1)\Pr(z_{T-1}, z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T|z_{T-1})\Pr(z_{T-1}, z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T|z_{T-1})\Pr(z_{T-1}|z_{T-2}, \ldots, z_1)\Pr(z_{T-2}, \ldots, z_1) \\ &= \Pr(z_T|z_{T-1})\Pr(z_{T-1}|z_{T-2})\Pr(z_{T-2}, \ldots, z_1) \\ &= \ldots \\ &= \Pr(z_T|z_{T-1})\Pr(z_{T-1}|z_{T-2})\ldots\Pr(z_2|z_1)\Pr(z_1)\end{aligned}$$

This is a product of conditional probabilities of states given previous states, and the probability of initial states $\Pr(z_1)$. Using a more compact notation for the product of conditional probabilities, we get:

$$
\begin{aligned}
\Pr(\mathbf{z}) &= \Pr(z_T, z_{T-1}, z_{T-2}, \ldots, z_1) \\
&= \Pr(z_T | z_{T-1}, z_{T-2}, \ldots, z_1) \Pr(z_{T-1}, z_{T-2}, \ldots, z_1) \\
&= \Pr(z_T | z_{T-1}) \Pr(z_{T-1}, z_{T-2}, \ldots, z_1) \\
&= \Pr(z_T | z_{T-1}) \Pr(z_{T-1} | z_{T-2}, \ldots, z_1) \Pr(z_{T-2}, \ldots, z_1) \\
&= \Pr(z_T | z_{T-1}) \Pr(z_{T-1} | z_{T-2}) \Pr(z_{T-2}, \ldots, z_1) \\
&= \ldots \\
&= \Pr(z_T | z_{T-1}) \Pr(z_{T-1} | z_{T-2}) \ldots \Pr(z_2 | z_1) \Pr(z_1) \\
&= \Pr(z_1) \prod_{t=2}^{T} \Pr(z_t | z_{t-1})
\end{aligned}
$$

In the product, you can recognize the transition parameters γ we defined above, so that the likelihood of a Markov model can be written as:

$$
\begin{aligned}
\Pr(\mathbf{z}) &= \Pr(z_T, z_{T-1}, z_{T-2}, \ldots, z_1) \\
&= \Pr(z_1) \prod_{t=2}^{T} \gamma_{z_{t-1}, z_t}
\end{aligned}
$$

3.3.5 Example

I realise these calculations are a bit difficult to follow. Let's take an example to fix ideas. Let's assume an animal is alive, alive at time 2 then dies at time 3. We have $\mathbf{z} = (1, 1, 2)$. What is the contribution of this animal to the likelihood? Let's apply the formula we just derived:

$$
\begin{aligned}
\Pr(\mathbf{z} = (1, 1, 2)) &= \Pr(z_1 = 1)\ \gamma_{z_1=1, z_2=1}\ \gamma_{z_2=1, z_3=2} \\
&= 1\ \phi\ (1 - \phi).
\end{aligned}
$$

The probability of having the sequence alive, alive and dead is the probability of being alive first, then to stay alive, eventually to die. The probability of being alive at first occasion being 1, we have that the contribution of this individual to the likelihood is $\phi(1 - \phi)$.

3.4 Bayesian formulation

Before implementing this model in NIMBLE, we provide a Bayesian formulation of our model. We first note that the likelihood is a product of conditional probabilities of binary events (alive or dead). Usually binary events are associated with the Bernoulli distribution. Here however, we will use its extension to several outcomes (from a coin with two sides to a dice with more than two faces) known as the categorical distribution[3]. To get a better idea of how the categorical distribution works, let's simulate from it with the `rcat()` function. Consider for example a random value drawn from a categorical distribution with probability 0.1, 0.3 and 0.6. Think of a dice with three faces, face 1 has probability 0.1 of occurring, face 2 probability 0.3 and face 3 has probability 0.6, the sum of these probabilities being 1. We expect to get a 3 more often than a 2 and rarely a 1[4]:

```
rcat(n = 1, prob = c(0.1, 0.3, 0.6))
## [1] 3
```

Here is another example in which we sample 20 times in a categorical distribution with probabilities 0.1, 0.1, 0.4, 0.2 and 0.2, hence a dice with 5 faces:

```
rcat(n = 20, prob = c(0.1, 0.1, 0.4, 0.2, 0.2))
##  [1] 5 3 4 5 4 3 4 2 2 3 1 3 5 5 3 4 2 3 2 3
```

In this chapter, you will familiarise yourself with the categorical distribution in binary situations, which should make the transition to more states than just alive and dead smoother in the next chapters.

Initial state is a categorical random variable with probability δ. That is you have a dice with two faces, or a coin, and you have some probability

[3] The categorical distribution is a multinomial distribution with a single draw.

[4] Alternatively, you can use the `sample()` function and `sample(x = 1:3, size = 1, replace = FALSE, prob = c(0.1, 0.3, 0.6))`

to be alive, and one minus that probability to be dead. Of course, it you want your Markov chain to start, you'd better say it's alive so that δ is just $(1, 0)$:

$$z_1 \sim \text{Categorical}(\delta) \qquad \text{[likelihood, } t = 1\text{]}$$

Now the main part is the dynamic of the states. The state z_t at t depends only on the known state z_{t-1} at $t-1$, and is a categorical random variable which probabilities are given by row z_{t-1} of the transition matrix $\mathbf{\Gamma} = \gamma_{z_{t-1}, z_t}$:

$$\begin{aligned} z_1 &\sim \text{Categorical}(\delta) && \text{[likelihood, } t = 1\text{]} \\ z_t | z_{t-1} &\sim \text{Categorical}(\gamma_{z_{t-1}, z_t}) && \text{[likelihood, } t > 1\text{]} \end{aligned}$$

For example, if individual i is alive over $(t-1, t)$ i.e. $z_{t-1} = 1$, we need the first row in $\mathbf{\Gamma}$,

$$\mathbf{\Gamma} = \begin{pmatrix} \phi & 1-\phi \\ 0 & 1 \end{pmatrix}$$

that is $\gamma_{z_{t-1}=1, z_t} = (\phi, 1-\phi)$ and $z_t | z_{t-1} = 1 \sim \text{Categorical}((\phi, 1-\phi))$.

Otherwise, if individual i dies over $(t-1, t)$ i.e. $z_{t-1} = 2$, we need the second row in $\mathbf{\Gamma}$:

$$\mathbf{\Gamma} = \begin{pmatrix} \phi & 1-\phi \\ 0 & 1 \end{pmatrix}$$

that is $\gamma_{z_{t-1}=2, z_t} = (0, 1)$ and $z_t | z_{t-1} = 2 \sim \text{Categorical}((0, 1))$ (if the individual is dead, it remains dead with probability 1).

We also need a prior on survival. Without surprise, we will use a uniform distribution between 0 and 1, which is also a Beta distribution with parameters 1 and 1. Overall our model is:

$$
\begin{aligned}
z_1 &\sim \text{Categorical}(\delta) && \text{[likelihood, } t = 1\text{]} \\
z_t|z_{t-1} &\sim \text{Categorical}(\gamma_{z_{t-1},z_t}) && \text{[likelihood, } t > 1\text{]} \\
\phi &\sim \text{Beta}(1,1) && \text{[prior for } \phi\text{]}
\end{aligned}
$$

3.5 NIMBLE implementation

How to implement in NIMBLE the Markov model we just built? We need to put in place a few bricks before running our model. Let's start with the prior on survival, the vector of initial state probabilities and the transition matrix:

```
markov.survival <- nimbleCode({
  phi ~ dunif(0, 1) # prior
  delta[1] <- 1           # Pr(alive t = 1) = 1
  delta[2] <- 0           # Pr(dead t = 1) = 0
  gamma[1,1] <- phi       # Pr(alive t -> alive t+1)
  gamma[1,2] <- 1 - phi   # Pr(alive t -> dead t+1)
  gamma[2,1] <- 0         # Pr(dead t -> alive t+1)
  gamma[2,2] <- 1         # Pr(dead t -> dead t+1)
...
```

Alternatively, you can define vectors and matrices in NIMBLE like you would do it in R. You can write:

```
markov.survival <- nimbleCode({
  phi ~ dunif(0, 1) # prior
  delta[1:2] <- c(1, 0) # vector of initial state probabilities
  gamma[1:2,1:2] <- matrix( c(phi, 0, 1 - phi, 1), nrow = 2) # transition matrix
...
```

Now there are two important dimensions to our model, along which we need to repeat tasks, namely individual and time. As for time, we

describe the successive events of survival using the categorical distribution `dcat()`, say for individual i:

```
z[i,1] ~ dcat(delta[1:2])           # t = 1
z[i,2] ~ dcat(gamma[z[i,1], 1:2])   # t = 2
z[i,3] ~ dcat(gamma[z[i,2], 1:2])   # t = 3
...
z[i,T] ~ dcat(gamma[z[i,T-1], 1:2]) # t = T
```

There is a more efficient way to write this piece of code by using a for loop, that is a sequence of instructions that we repeat. Here, we condense the previous code into:

```
z[i,1] ~ dcat(delta[1:2])             # t = 1
for (t in 2:T){ # loop over time t
  z[i,t] ~ dcat(gamma[z[i,t-1], 1:2]) # t = 2,...,T
}
```

Now we just need to do the same for all individuals. We use another loop:

```
for (i in 1:N){ # loop over individual i
  z[i,1] ~ dcat(delta[1:2]) # t = 1
  for (j in 2:T){ # loop over time t
    z[i,j] ~ dcat(gamma[z[i,j-1], 1:2]) # t = 2,...,T
  } # t
} # i
```

Puting everything together, the NIMBLE code for our Markov model is:

```
markov.survival <- nimbleCode({
  phi ~ dunif(0, 1) # prior
  delta[1] <- 1          # Pr(alive t = 1) = 1
  delta[2] <- 0          # Pr(dead t = 1) = 0
  gamma[1,1] <- phi      # Pr(alive t -> alive t+1)
  gamma[1,2] <- 1 - phi  # Pr(alive t -> dead t+1)
```

```
  gamma[2,1] <- 0         # Pr(dead t -> alive t+1)
  gamma[2,2] <- 1         # Pr(dead t -> dead t+1)
  # likelihood
  for (i in 1:N){ # loop over individual i
    z[i,1] ~ dcat(delta[1:2]) # t = 1
    for (j in 2:T){ # loop over time t
      z[i,j] ~ dcat(gamma[z[i,j-1], 1:2]) # t = 2,...,T
    } # t
  } # i
})
```

Note that in this example, δ is used as a placeholder for more complex models we will build in chapters to come. Here, you could simply write `z[i,1] <- 1`.

Now we're ready to resume our NIMBLE workflow. First we read in data. Because we have loops and indices that do not change, we use constants as explained in Section 2.3:

```
my.constants <- list(N = 57, T = 5)
my.data <- list(z = z)
```

We also specify initial values for survival with a function:

```
initial.values <- function() list(phi = runif(1,0,1))
initial.values()
## $phi
## [1] 0.1265
```

There is a single parameter to monitor:

```
parameters.to.save <- c("phi")
parameters.to.save
## [1] "phi"
```

We run 2 chains with 5000 iterations including 1000 iterations as burnin:

```
n.iter <- 5000
n.burnin <- 1000
n.chains <- 2
```

Let's run NIMBLE:

```
mcmc.output <- nimbleMCMC(code = markov.survival,
                          constants = my.constants,
                          data = my.data,
                          inits = initial.values,
                          monitors = parameters.to.save,
                          niter = n.iter,
                          nburnin = n.burnin,
                          nchains = n.chains)
```

Let's calculate the usual posterior numerical summaries for survival:

```
MCMCsummary(mcmc.output, round = 2)
##     mean   sd 2.5%  50% 97.5% Rhat n.eff
## phi 0.79 0.03 0.73 0.79  0.85    1  1755
```

Posterior mean and median are close to 0.8. This is fortunate since the data was simulated with (actual) survival $\phi = 0.8$. The code I used was:

```
# 1 = alive, 2 = dead
nind <- 57
nocc <- 5
phi <- 0.8 # survival probability
delta <- c(1,0) # (Pr(alive at t = 1), Pr(dead at t = 1))
Gamma <- matrix(NA, 2, 2) # transition matrix
Gamma[1,1] <- phi      # Pr(alive t -> alive t+1)
Gamma[1,2] <- 1 - phi  # Pr(alive t -> dead t+1)
Gamma[2,1] <- 0        # Pr(dead t -> alive t+1)
```

```
Gamma[2,2] <- 1         # Pr(dead t -> dead t+1)
z <- matrix(NA, nrow = nind, ncol = nocc)
set.seed(2022)
for (i in 1:nind){
  z[i,1] <- rcat(n = 1, prob = delta) # 1 for sure
  for (t in 2:nocc){
    z[i,t] <- rcat(n = 1, prob = Gamma[z[i,t-1],1:2])
  }
}
head(z)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    1    1    1
## [2,]    1    1    1    1    1
## [3,]    1    1    1    1    1
## [4,]    1    1    1    1    2
## [5,]    1    1    1    1    1
## [6,]    1    1    2    2    2
```

We could `dcat()` by `dbern()` everywhere in the code because we have binary events alive/dead. Would it make any difference? Although `dcat()` uses less efficient samplers than `dbern()` (**check w/ Perry/Daniel**), `dcat()` is convenient for model building to accomodate more than two outcomes, a feature that will become handy in the next chapters.

3.6 Hidden Markov models

3.6.1 Capture-recapture data

Unfortunately, the data with alive and dead states is the data we wish we had. In real life, animals cannot be monitored exhaustively, like humans in a medical trial. This is why we use capture-recapture protocols[5], in which animals are captured, individually marked, and released alive. Then, these animals may be detected again, or go unde-

tected. Whenever animals go undetected, it might be that they were alive but missed, or because they were dead and therefore could not be detected. This issue is usually referred to as that of imperfect detection. As a consequence of imperfect detection, the Markov process for survival is only partially observed: You know an animal is alive when you detect it, but when an animal goes undetected, whether it is alive or dead is unknown to you. This is where hidden Markov models (HMMs) come in.

Let's get back to the data we had in the previous section. The truth is in z which contains the fate of all individuals with $z = 1$ for alive, and $z = 2$ for dead:

id	winter 1	winter 2	winter 3	winter 4	winter 5
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	1	1	1
4	1	1	1	1	2
5	1	1	1	1	1
6	1	1	2	2	2
7	1	1	1	1	1
8	1	2	2	2	2
9	1	1	1	1	2
10	1	2	2	2	2
11	1	1	1	1	1
12	1	1	1	1	1
13	1	1	1	2	2
14	1	1	1	1	1
15	1	2	2	2	2
16	1	1	1	1	1
17	1	1	1	1	1
18	1	1	1	1	2
19	1	1	1	1	1
20	1	1	2	2	2
21	1	1	2	2	2
22	1	2	2	2	2
23	1	1	1	1	2
24	1	1	1	1	2
25	1	1	1	1	1
26	1	1	1	1	2
27	1	1	1	1	2
28	1	2	2	2	2
29	1	1	2	2	2
30	1	1	2	2	2
31	1	2	2	2	2
32	1	2	2	2	2
33	1	2	2	2	2
34	1	1	2	2	2
35	1	1	2	2	2
36	1	2	2	2	2
37	1	1	1	1	1
38	1	1	1	2	2
39	1	1	1	1	1
40	1	1	1	1	2
41	1	1	1	1	1
42	1	1	1	2	2
43	1	1	1	1	1

Unfortunately, we have only partial access to z. What we do observe is y the detections and non-detections. How are z and y connected?

The easiest connection is with dead animals which go undetected for sure. Therefore when an animal is dead i.e. $z = 2$, it cannot be detected, therefore $y = 0$: **why not use 1 for non-detected and 2 for detected from here, and mention somewhere that usually people use 0 and 1?**

id	winter 1	winter 2	winter 3	winter 4	winter 5
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	1	1	1
4	1	1	1	1	0
5	1	1	1	1	1
6	1	1	0	0	0
7	1	1	1	1	1
8	1	0	0	0	0
9	1	1	1	1	0
10	1	0	0	0	0
11	1	1	1	1	1
12	1	1	1	1	1
13	1	1	1	0	0
14	1	1	1	1	1
15	1	0	0	0	0
16	1	1	1	1	1
17	1	1	1	1	1
18	1	1	1	1	0
19	1	1	1	1	1
20	1	1	0	0	0
21	1	1	0	0	0
22	1	0	0	0	0
23	1	1	1	1	0
24	1	1	1	1	0
25	1	1	1	1	1
26	1	1	1	1	0
27	1	1	1	1	0
28	1	0	0	0	0
29	1	1	0	0	0
30	1	1	0	0	0
31	1	0	0	0	0
32	1	0	0	0	0
33	1	0	0	0	0
34	1	1	0	0	0
35	1	1	0	0	0
36	1	0	0	0	0
37	1	1	1	1	1
38	1	1	1	0	0
39	1	1	1	1	1
40	1	1	1	1	0
41	1	1	1	1	1
42	1	1	1	0	0
43	1	1	1	1	1

Now alive animals may be detected or not. If an animal is alive $z = 1$, it is detected $y = 1$ with probability p or not $y = 0$ with probability $1 - p$. In our example, first detection coincides with first winter for all individuals.

id	winter 1	winter 2	winter 3	winter 4	winter 5
1	1	0	0	0	0
2	1	0	1	0	1
3	1	0	0	0	0
4	1	1	1	1	0
5	1	1	1	1	1
6	1	0	0	0	0
7	1	0	1	1	1
8	1	1	1	1	0
9	1	1	1	1	0
10	1	1	0	0	1
11	1	1	0	0	0
12	1	0	0	0	0
13	1	0	0	0	0
14	1	1	0	0	1
15	1	0	0	0	0
16	1	0	0	0	0
17	1	1	0	0	0
18	1	0	0	0	0
19	1	1	0	0	0
20	1	0	0	0	0
21	1	1	1	0	1
22	1	1	1	1	0
23	1	0	0	0	0
24	1	0	0	0	0
25	1	0	0	0	0
26	1	1	0	0	1
27	1	1	0	0	1
28	1	0	0	0	0
29	1	1	0	1	1
30	1	1	1	0	0
31	1	0	0	0	0
32	1	0	0	0	0
33	1	0	0	0	0
34	1	0	0	0	0
35	1	1	0	0	0
36	1	1	0	1	0
37	1	0	0	1	0
38	1	0	0	0	0
39	1	1	0	0	0
40	1	0	1	0	0
41	1	0	0	0	0
42	1	1	1	0	0
43	1	1	1	0	0

Compare with previous table. Some 1's for alive have become 0's for non-detection, other 1's for alive have remained 1's for detection. This table y is what we observe in real life. I hope I have convinced you that to make the connection between observations, the y, and true states, the z, we need to describe how observations are made (or emitted in the HMM terminology) from the states.

3.6.2 Observation matrix

The novelty in HMMs is the link between observations and states. This link is made through observation probabilities. For example, the probability of detecting an animal i at t given it is alive at t is $\Pr(y_{i,t} = 2 | z_{i,t} = 1) = \omega_{1,2}$. It is the detection probability p. If individual i is dead at t, then it is missed for sure, and $\Pr(y_{i,t} = 1 | z_{i,t} = 2) = \omega_{2,1} = 1$.

We can gather these observation probabilities into an observation matrix $\mathbf{\Omega}$. In rows we have the states alive $z = 1$ and dead $z = 2$, while in columns we have the observations non-detected $y = 1$ and detected $y = 2$ (previously coded 0 and 1 respectively): **if we go for 1 and 2, do wee need the comment between parentheses?**

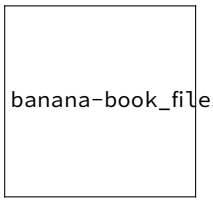
$$\mathbf{\Omega} = \begin{pmatrix} \omega_{1,1} & \omega_{1,2} \\ \omega_{2,1} & \omega_{2,2} \end{pmatrix} = \begin{pmatrix} 1-p & p \\ 1 & 0 \end{pmatrix}$$

Observation matrix:

$$\mathbf{\Omega} = \begin{array}{c} \begin{array}{cc} y_t = 1 & y_t = 2 \\ \text{(non-detected)} & \text{(detected)} \end{array} \\ \begin{pmatrix} 1-p & p \\ 1 & 0 \end{pmatrix} \end{array} \begin{array}{l} z_t = 1 \text{ (alive)} \\ z_t = 2 \text{ (dead)} \end{array}$$

3.6.3 Hidden Markov model

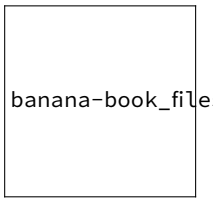
Our hidden Markov model can be represented this way:



States z are in gray. Observations y are in white. All individuals are first captured in the first winter $t = 1$, and are therefore all alive $z_1 = 1$ and detected $y_1 = 2$.

A hidden Markov model is just two time series running in parallel. One for the states with the Markovian property, and the other of for the observations generated from the states[6].

Have a look to the example below, in which an individual is detected at first sampling occasion, detected again, then missed for the rest of the study. While on occasion $t = 3$ that individual was alive $z_3 = 1$ and went undetected $y_3 = 1$, on occasions $t = 4$ and $t = 5$ it went undetected $y_4 = y_5 = 1$ because it was dead $z_4 = z_5 = 2$.



3.6.4 Likelihood

In the Bayesian framework, we usually work with the so-called complete likelihood, that is the probability of the observed data y and the latent states z given the parameters of our model, here the survival and detection probabilities ϕ and p. The complete likelihood for individual i is:

$$\Pr(\mathbf{y}_i, \mathbf{z}_i) = \Pr(y_{i,1}, y_{i,2}, \ldots, y_{i,T}, z_{i,1}, z_{i,2}, \ldots, z_{i,T})$$

Using the definition of a conditional probability, we have:

$$\begin{aligned}\Pr(\mathbf{y}_i) &= \Pr(y_{i,1}, y_{i,2}, \ldots, y_{i,T}) \\ &= \Pr(y_{i,1}, y_{i,2}, \ldots, y_{i,T} | z_{i,1}, z_{i,2}, \ldots, z_{i,T}) \Pr(z_{i,1}, z_{i,2}, \ldots, z_{i,T})\end{aligned}$$

Then by using the independence of the y conditional on the z, and the likelihood of a Markov chain, we get that:

$$\begin{aligned}\Pr(\mathbf{y}_i) &= \Pr(y_{i,1}, y_{i,2}, \ldots, y_{i,T}) \\ &= \Pr(y_{i,1}, y_{i,2}, \ldots, y_{i,T} | z_{i,1}, z_{i,2}, \ldots, z_{i,T}) \Pr(z_{i,1}, z_{i,2}, \ldots, z_{i,T}) \\ &= \left(\prod_{t=1}^{T} \Pr\left(y_{i,t} | z_{i,t}\right) \right) \left(\Pr(z_{i,1}) \prod_{t=2}^{T} \Pr\left(z_{i,t} | z_{i,t-1}\right) \right)\end{aligned}$$

Finally, by recognizing the observation and transition probabilities, we have that the complete likelihood for individual i is:

$$\begin{aligned}\Pr(\mathbf{y}_i) &= \Pr(y_{i,1}, y_{i,2}, \ldots, y_{i,T}) \\ &= \Pr(y_{i,1}, y_{i,2}, \ldots, y_{i,T} | z_{i,1}, z_{i,2}, \ldots, z_{i,T}) \Pr(z_{i,1}, z_{i,2}, \ldots, z_{i,T}) \\ &= \left(\prod_{t=1}^{T} \omega_{z_{i,t}, y_{i,t}} \right) \left(\Pr(z_{i,1}) \prod_{t=2}^{T} \gamma_{z_{i,t-1}, z_{i,t}} \right)\end{aligned}$$

To obtain the complete likelihood of the whole dataset, we need to multiply this individual likelihood for each animal $\prod_{i=1}^{N} \Pr(\mathbf{y}_i, \mathbf{z}_i)$. When several individuals have the same contribution, calculating their individual contribution only once can greatly reduce the computational burden, as illustrated in Section 3.9.

The Bayesian approach with MCMC methods allows treating the latent states $z_{i,t}$ as if they were parameters, and to be estimated as such. However, the likelihood is rather complex with a large number of latent states $z_{i,t}$, which comes with computational costs and slow mixing.

There are situations where the latent states are the focus of ecological inference and need to be estimated (see Suggested reading below). However, if not needed, you might want to get rid of the latent states and rely on the so-called marginal likelihood. By doing so, you can avoid sampling the latent states, focus on the ecological parameters, and often speeds up computations and improves mixing as shown in Section 3.8. Actually, you can even estimate the latent states afterwards, as illustrated in Section 3.10.

3.7 Fitting HMM with NIMBLE

Our model so far is written as follows:

$$
\begin{aligned}
z_{\text{first}} &\sim \text{Categorical}(1, \delta) && \text{[likelihood]} \\
z_t|z_{t-1} &\sim \text{Categorical}(1, \gamma_{z_{t-1},z_t}) && \text{[likelihood]} \\
y_t|z_t &\sim \text{Categorical}(1, \omega_{z_t}) && \text{[likelihood]} \\
\phi &\sim \text{Beta}(1, 1) && \text{[prior for } \phi\text{]} \\
p &\sim \text{Beta}(1, 1) && \text{[prior for } p\text{]}
\end{aligned}
$$

It has an observation layer for the y's, conditional on the z's. We also consider uniform priors for the detection and survival probabilities. How to implement this model in NIMBLE?

We start with priors for survival and detection probabilities:

```
hmm.survival <- nimbleCode({
  phi ~ dunif(0, 1) # prior survival
  p ~ dunif(0, 1) # prior detection
...
```

Then we define initial states, transition and observation matrices:

```
...
  # parameters
  delta[1] <- 1          # Pr(alive t = 1) = 1
  delta[2] <- 0          # Pr(dead t = 1) = 0
  gamma[1,1] <- phi      # Pr(alive t -> alive t+1)
  gamma[1,2] <- 1 - phi  # Pr(alive t -> dead t+1)
  gamma[2,1] <- 0        # Pr(dead t -> alive t+1)
  gamma[2,2] <- 1        # Pr(dead t -> dead t+1)
  omega[1,1] <- 1 - p    # Pr(alive t -> non-detected t)
  omega[1,2] <- p        # Pr(alive t -> detected t)
  omega[2,1] <- 1        # Pr(dead t -> non-detected t)
  omega[2,2] <- 0        # Pr(dead t -> detected t)
...
```

Then the likelihood:

```
...
    # likelihood
    for (i in 1:N){
    z[i,1] ~ dcat(delta[1:2])
    for (j in 2:T){
      z[i,j] ~ dcat(gamma[z[i,j-1], 1:2])
      y[i,j] ~ dcat(omega[z[i,j], 1:2])
    }
  }
})
```

Overall, the code looks like:

```
hmm.survival <- nimbleCode({
  phi ~ dunif(0, 1) # prior survival
  p ~ dunif(0, 1) # prior detection
  # likelihood
  delta[1] <- 1          # Pr(alive t = 1) = 1
  delta[2] <- 0          # Pr(dead t = 1) = 0
  gamma[1,1] <- phi      # Pr(alive t -> alive t+1)
```

```
  gamma[1,2] <- 1 - phi  # Pr(alive t -> dead t+1)
  gamma[2,1] <- 0        # Pr(dead t -> alive t+1)
  gamma[2,2] <- 1        # Pr(dead t -> dead t+1)
  omega[1,1] <- 1 - p    # Pr(alive t -> non-detected t)
  omega[1,2] <- p        # Pr(alive t -> detected t)
  omega[2,1] <- 1        # Pr(dead t -> non-detected t)
  omega[2,2] <- 0        # Pr(dead t -> detected t)
  for (i in 1:N){
    z[i,1] ~ dcat(delta[1:2])
    for (j in 2:T){
      z[i,j] ~ dcat(gamma[z[i,j-1], 1:2])
      y[i,j] ~ dcat(omega[z[i,j], 1:2])
    }
  }
})
```

Now we specify the constants:

```
my.constants <- list(N = nrow(y), T = 5)
my.constants
## $N
## [1] 57
##
## $T
## [1] 5
```

The data are made of 0's for non-detections and 1's for detections. To use the categorical distribution, we need to code 1's and 2's. We simply add 1 to get the correct format, that is $y = 1$ for non-detection and $y = 2$ for detection: **Using 1 and 2 would make my life easier… The 0/1 coding is a convention; Using the 1/2 coding would make clear that non-detections are actual data (while the use of 0s for non-detections is sometimes confusing). Do it, do it.**

```
my.data <- list(y = y + 1)
```

Now let's write a function for the initial values:

```
zinits <- y + 1 # non-detection -> alive
zinits[zinits == 2] <- 1 # dead -> alive
initial.values <- function() list(phi = runif(1,0,1),
                                  p = runif(1,0,1),
                                  z = zinits)
```

We specify the parameters we'd like to monitor:

```
parameters.to.save <- c("phi", "p")
parameters.to.save
## [1] "phi" "p"
```

We provide MCMC details:

```
n.iter <- 5000
n.burnin <- 1000
n.chains <- 2
```

At last, we're ready to run NIMBLE:

```
start_time <- Sys.time()
mcmc.output <- nimbleMCMC(code = hmm.survival,
                          constants = my.constants,
                          data = my.data,
                          inits = initial.values,
                          monitors = parameters.to.save,
                          niter = n.iter,
                          nburnin = n.burnin,
                          nchains = n.chains)
end_time <- Sys.time()
end_time - start_time
```

```
## Time difference of 32.39 secs
```

We can have a look to numerical summaries:

```
MCMCsummary(mcmc.output, round = 2)
##      mean   sd 2.5%  50% 97.5% Rhat n.eff
## p    0.61 0.06 0.50 0.61  0.72    1   740
## phi  0.75 0.04 0.67 0.75  0.83    1   805
```

The estimates for survival and detection are close to true survival $\phi = 0.8$ and detection $p = 0.6$ with which we simulated the data. The code I used is:

```
set.seed(2022) # for reproducibility
nocc <- 5 # nb of winters or sampling occasions
nind <- 57 # nb of animals
p <- 0.6 # detection prob
phi <- 0.8 # survival prob
# Vector of initial states probabilities
delta <- c(1,0) # all individuals are alive in first winter
# Transition matrix
Gamma <- matrix(NA, 2, 2)
Gamma[1,1] <- phi      # Pr(alive t -> alive t+1)
Gamma[1,2] <- 1 - phi  # Pr(alive t -> dead t+1)
Gamma[2,1] <- 0        # Pr(dead t -> alive t+1)
Gamma[2,2] <- 1        # Pr(dead t -> dead t+1)
# Observation matrix
Omega <- matrix(NA, 2, 2)
Omega[1,1] <- 1 - p      # Pr(alive t -> non-detected t)
Omega[1,2] <- p          # Pr(alive t -> detected t)
Omega[2,1] <- 1          # Pr(dead t -> non-detected t)
Omega[2,2] <- 0          # Pr(dead t -> detected t)
# Matrix of states
z <- matrix(NA, nrow = nind, ncol = nocc)
y <- z
y[,1] <- 2 # all individuals are detected in first winter
for (i in 1:nind){
```

```
  z[i,1] <- rcat(n = 1, prob = delta) # 1 for sure
  for (t in 2:nocc){
    # state at t given state at t-1
    z[i,t] <- rcat(n = 1, prob = Gamma[z[i,t-1],1:2])
    # observation at t given state at t
    y[i,t] <- rcat(n = 1, prob = Omega[z[i,t],1:2])
  }
}
y
y <- y - 1 # non-detection = 0, detection = 1
```

3.8 Marginalization

In some situations, you will not be interested in inferring the hidden states $z_{i,t}$, so why bother estimating them? The good news is that you can get rid of the states, so that the marginal likelihood is a function of survival and detection probabilities ϕ and p only.

3.8.1 Brute-force approach

Using the formula of total probability, we get the marginal likelihood by summing over all possible states in the complete likelihood:

$$\begin{aligned}\Pr(\mathbf{y}_i) &= \Pr(y_{i,1}, y_{i,2}, \ldots, y_{i,T}) \\ &= \sum_{\mathbf{z}_i} \Pr(\mathbf{y}_i, \mathbf{z}_i) \\ &= \sum_{z_{i,1}} \cdots \sum_{z_{i,T}} \Pr(y_{i,1}, y_{i,2}, \ldots, y_{i,T}, z_{i,1}, z_{i,2}, \ldots, z_{i,T})\end{aligned}$$

Going through the same steps as for deriving the complete likelihood, we obtain the marginal likelihood:

$$\Pr(\mathbf{y}_i) = \sum_{z_{i,1}} \cdots \sum_{z_{i,T}} \left(\prod_{t=1}^{T} \omega_{z_{i,t},y_{i,t}} \right) \left(\Pr(z_{i,1}) \prod_{t=2}^{T} \gamma_{z_{i,t-1},z_{i,t}} \right)$$

Let's go through an example. Let's imagine we have $T = 3$ winters, and we'd like to write the likelihood for an individual having the encounter history detected, detected then non-detected. Remember that non-detected is coded 1 and detected 2, while alive is coded 1 and dead 2. We need to calculate $\Pr(y_1 = 2, y_2 = 2, y_3 = 1)$ which, according to the formula above, is given by:

$$\Pr(y_1 = 2, y_2 = 2, y_3 = 1) = \sum_{i=1}^{2} \sum_{j=1}^{2} \sum_{k=1}^{2} [\Pr(y_1 = 2|z_1 = i) \Pr(y_2 = 2|z_2 = j) \Pr(y_3 = 1|z_3 = k)]$$
$$(\Pr(z_1 = i) \Pr(z_2 = j|z_1 = i) \Pr(z_3 = k|z_2 = j))$$

Expliciting all the sums in $\Pr(y_1 = 2, y_2 = 2, y_3 = 1)$, we get the long and ugly expression:

$$
\begin{aligned}
\Pr(y_1 = 2, y_2 = 2, y_3 = 1) = \\
&\Pr(y_1 = 2|z_1 = 1)\Pr(y_2 = 2|z_2 = 1)\Pr(y_3 = 1|z_3 = 1)\times \\
&\quad \Pr(z_1 = 1)\Pr(z_2 = 1|z_1 = 1)\Pr(z_3 = 1|z_2 = 1) + \\
&\Pr(y_1 = 2|z_1 = 2)\Pr(y_2 = 2|z_2 = 1)\Pr(y_3 = 1|z_3 = 1)\times \\
&\quad \Pr(z_1 = 2)\Pr(z_2 = 1|z_1 = 2)\Pr(z_3 = 1|z_2 = 1) + \\
&\Pr(y_1 = 2|z_1 = 1)\Pr(y_2 = 2|z_2 = 2)\Pr(y_3 = 1|z_3 = 1)\times \\
&\quad \Pr(z_1 = 1)\Pr(z_2 = 2|z_1 = 1)\Pr(z_3 = 1|z_2 = 2) + \\
&\Pr(y_1 = 2|z_1 = 2)\Pr(y_2 = 2|z_2 = 2)\Pr(y_3 = 1|z_3 = 1)\times \\
&\quad \Pr(z_1 = 2)\Pr(z_2 = 2|z_1 = 2)\Pr(z_3 = 1|z_2 = 2) + \\
&\Pr(y_1 = 2|z_1 = 1)\Pr(y_2 = 2|z_2 = 1)\Pr(y_3 = 1|z_3 = 2)\times \\
&\quad \Pr(z_1 = 1)\Pr(z_2 = 1|z_1 = 1)\Pr(z_3 = 2|z_2 = 1) + \\
&\Pr(y_1 = 2|z_1 = 2)\Pr(y_2 = 2|z_2 = 1)\Pr(y_3 = 1|z_3 = 2)\times \\
&\quad \Pr(z_1 = 2)\Pr(z_2 = 1|z_1 = 2)\Pr(z_3 = 2|z_2 = 1) + \\
&\Pr(y_1 = 2|z_1 = 1)\Pr(y_2 = 2|z_2 = 2)\Pr(y_3 = 1|z_3 = 2)\times \\
&\quad \Pr(z_1 = 1)\Pr(z_2 = 2|z_1 = 1)\Pr(z_3 = 2|z_2 = 2) + \\
&\Pr(y_1 = 2|z_1 = 2)\Pr(y_2 = 2|z_2 = 2)\Pr(y_3 = 1|z_3 = 2)\times \\
&\quad \Pr(z_1 = 2)\Pr(z_2 = 2|z_1 = 2)\Pr(z_3 = 2|z_2 = 2)
\end{aligned}
$$

You can simplify this expression by noticing that i) all individuals are alive for sure when marked and released in first winter, or $\Pr(z_1 = 2) = 0$ and ii) dead individuals are non-detected for sure, or $\Pr(y_t = 2|z_t = 2) = 0$, which lead to:

$$
\begin{aligned}
\Pr(y_1 = 2, y_2 = 2, y_3 = 1) = \\
&\Pr(y_1 = 2|z_1 = 1)\Pr(y_2 = 2|z_2 = 1)\Pr(y_3 = 1|z_3 = 1)\times \\
&\quad \Pr(z_1 = 1)\Pr(z_2 = 1|z_1 = 1)\Pr(z_3 = 1|z_2 = 1) + \\
&\Pr(y_1 = 2|z_1 = 1)\Pr(y_2 = 2|z_2 = 1)\Pr(y_3 = 1|z_3 = 2)\times \\
&\quad \Pr(z_1 = 1)\Pr(z_2 = 1|z_1 = 1)\Pr(z_3 = 2|z_2 = 1)
\end{aligned}
$$

Because all individuals are captured in first winter, or $\Pr(y_1 = 2|z_1 = 1) = 1$, we get:

$$\begin{aligned}\Pr(y_1 = 2, y_2 = 2, y_3 = 1) = \\ & 1(1-p)\times \\ & \quad 1\phi\phi + \\ & 1p1\times \\ & \quad 1\phi(1-\phi)\end{aligned}$$

You end up with $\Pr(y_1 = 2, y_2 = 2, y_3 = 1) = \phi p(1 - p\phi)$.

The latent states are no longer involved in the likelihood for this individual. However, even on a rather simple example, the marginal likelihood is quite complex to evaluate because it involves many operations. If T is the length of our encounter histories and N is the number of hidden states (two for alive and dead, but this will be more in some chapters to come), then we need to calculate the sum of N^T terms (the sums in the formula above), each of which has two products of T factors (the products in the formula above), hence $2TN^T$ calculations in total. You can check that in the simple example above, we have $T^N = 2^3 = 8$ terms that are summed, each of which is a product of $2T = 2 \times 3 = 6$ terms. This means that the number of operations increases exponentially as the number of states increases. In most cases, this complexity precludes using this method to get rid of the states. Fortunately, we have another algorithm in the HMM toolbox that is useful to calculate the marginal likelihood efficiently.

3.8.2 Forward algorithm

In the brute-force approach, several products are computed several times to calculate the marginal likelihood. What if we could store these products and use them later while computing the probability of the observation sequence? This is precisely what the forward algorithm does.

We need to introduce $\alpha_t(j)$ the probability for the latent state z of being in state j at t after seeing the first j observations $y_1, \ldots, y_t$, that is $\alpha_t(j) = \Pr(y_1, \ldots, y_t, z_t = j)$.

Using the law of total probability, we can write the marginal

likelihood as a function of $\alpha_T(j)$, namely we have $\Pr(\mathbf{y}) = \sum_{j=1}^{N} \Pr(y_1, \ldots, y_t, z_t = j) = \sum_{j=1}^{N} \alpha_T(j)$.

How to calculate the the $\alpha_T(j)$s? This is where the magic of the forward algorithm happens. We use a recurrence relationship that saves us many computations.

The recurrence states that:

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i)\gamma_{i,j}\omega_{j,y_t}$$

How to obtain this recurrence? First, using the law of total probability with z_{t-1}, we have that:

$$\alpha_t(j) = \sum_{i=1}^{N} \Pr(y_1, \ldots, y_t, z_{t-1} = i, z_t = j)$$

Second, using conditional probabilities, we get:

$$\alpha_t(j) = \sum_{i=1}^{N} \Pr(y_t | z_{t-1} = i, z_t = j, y_1, \ldots, y_t) \Pr(z_{t-1} = i, z_t = j, y_1, \ldots, y_t)$$

Third, using conditional probabilities again, on the second term of the product, we get:

$$\begin{aligned}\alpha_t(j) = \sum_{i=1}^{N} &\Pr(y_t | z_{t-1} = i, z_t = j, y_1, \ldots, y_t) \times \\ &\Pr(z_t = j | z_{t-1} = i, y_1, \ldots, y_t) \Pr(z_{t-1} = i, y_1, \ldots, y_t)\end{aligned}$$

Which, using conditional independence, simplifies into:

$$\alpha_t(j) = \sum_{i=1}^{N} \Pr(y_t | z_t = j) \Pr(z_t = j | z_{t-1} = i) \Pr(z_{t-1} = i, y_1, \ldots, y_t)$$

Recognizing that $\Pr(y_t | z_t = j) = \omega_{j,y_t}$, $\Pr(z_t = j | z_{t-1} = i) = \gamma_{i,j}$ and $\Pr(z_{t-1} = i, y_1, \ldots, y_t) = \alpha_{t-1}(i)$, we obtain the recurrence.

In practice, the forward algorithm works as follows. First you initialize the procedure by calculating for all j: $\alpha_1(j) = Pr(z_1 = j)\omega_{j,y_1}$. Then you compute for all j the relationship $\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i)\gamma_{i,j}\omega_{j,y_t}$ for $t = 2, \ldots, T$. Finally, you compute $\Pr(\mathbf{y}) = \sum_{j=1}^{N} \alpha_T(j)$. At each time t, we need to calculate N values of $\alpha_t(j)$, and each $\alpha_t(j)$ is a sum of N products of α_{t-1}, $\gamma_{i,j}$ and ω_{j,y_t}, hence TN^2 computations in total, much less than $2TN^T$ in the brute-force approach.

Going back to our example, we wish to calculate $\Pr(y_1 = 2, y_2 = 2, y_3 = 1)$. First we initialize and compute $\alpha_1(1)$ and $\alpha_1(2)$. We have:

$$\begin{aligned}\alpha_1(1) &= \Pr(z_1 = 1)\omega_{1,y_1=2} \\ &= 1\end{aligned}$$

because all animals are alive and captured in first winter. We also have:

$$\begin{aligned}\alpha_1(2) &= \Pr(z_1 = 2)\omega_{2,y_1=2} \\ &= 0\end{aligned}$$

Then we compute $\alpha_2(1)$ and $\alpha_2(2)$. We have:

$$\begin{aligned}\alpha_2(1) &= \sum_{i=1}^{2} \alpha_1(i)\gamma_{i,1}\omega_{1,y_2=2} \\ &= \gamma_{1,1}\omega_{1,y_2=2} \\ &= \phi p\end{aligned}$$

because $\alpha_1(2) = 0$. Also, we have:

$$\begin{aligned}\alpha_2(2) &= \sum_{i=1}^{2} \alpha_1(i)\gamma_{i,2}\omega_{2,y_2=2} \\ &= \gamma_{1,2}\omega_{2,y_2=2} \\ &= (1 - \phi)0\end{aligned}$$

Finally we compute $\alpha_3(1)$ and $\alpha_3(2)$. We have:

$$\begin{aligned}\alpha_3(1) &= \sum_{i=1}^{2} \alpha_2(i)\gamma_{i,1}\omega_{1,y_3=1} \\ &= \alpha_2(1)\gamma_{1,1}\omega_{1,y_3=1} \\ &= \phi p \phi(1-p)\end{aligned}$$

We also have:

$$\begin{aligned}\alpha_3(2) &= \sum_{i=1}^{2} \alpha_2(i)\gamma_{i,2}\omega_{2,y_3=1} \\ &= \alpha_2(1)\gamma_{1,2}\omega_{2,y_3=1} \\ &= \phi p(1-\phi)1\end{aligned}$$

Eventually, we compute $\Pr(y_1 = 2, y_2 = 2, y_3 = 1)$:

$$\begin{aligned}\Pr(y_1 = 2, y_2 = 2, y_3 = 1) &= \alpha_3(1) + \alpha_3(2) \\ &= \phi p(\phi)(1-p) + \phi p(1-\phi) \\ &= \phi p(1-\phi p)\end{aligned}$$

You can check that we did in total $3 \times 2^2 = 12$ operations.

3.8.3 NIMBLE implementation

In NIMBLE, we use functions to implement the forward algorithm. The only differences with the theory above is that i) we work on the log scale for numerical stability and ii) we use a matrix formulation of the recurrence.

First we write the density function:

```
dHMM <- nimbleFunction(
  run = function(x = double(1),
                 probInit = double(1),
                 probObs = double(2),
```

```
                probTrans = double(2),
                len = double(0, default = 0),
                log = integer(0, default = 0)) {
    alpha <- probInit[1:2]
    for (t in 2:len) {
      alpha[1:2] <- (alpha[1:2] %*% probTrans[1:2,1:2]) * probObs[1:2,x[t]]
    }
    logL <- log(sum(alpha[1:2]))
    returnType(double(0))
    if (log) return(logL)
    return(exp(logL))
  }
)
```

In passing, this is the function you would maximize in a Frequentist approach. Then we write a function to simulate values from a HMM:

```
rHMM <- nimbleFunction(
  run = function(n = integer(),
                probInit = double(1),
                probObs = double(2),
                probTrans = double(2),
                len = double(0, default = 0)) {
    returnType(double(1))
    z <- numeric(len)
    z[1] <- rcat(n = 1, prob = probInit[1:2]) # all individuals alive at t = 0
    y <- z
    y[1] <- 2 # all individuals are detected at t = 0
    for (t in 2:len){
      # state at t given state at t-1
      z[t] <- rcat(n = 1, prob = probTrans[z[t-1],1:2])
      # observation at t given state at t
      y[t] <- rcat(n = 1, prob = probObs[z[t],1:2])
    }
    return(y)
  })
```

We assign these functions to the global R environment:

```
assign('dHMM', dHMM, .GlobalEnv)
assign('rHMM', rHMM, .GlobalEnv)
```

Now we resume our workflow:

```
# code
hmm.survival <- nimbleCode({
  phi ~ dunif(0, 1) # prior survival
  p ~ dunif(0, 1) # prior detection
  # likelihood
  delta[1] <- 1           # Pr(alive t = 1) = 1
  delta[2] <- 0           # Pr(dead t = 1) = 0
  gamma[1,1] <- phi       # Pr(alive t -> alive t+1)
  gamma[1,2] <- 1 - phi   # Pr(alive t -> dead t+1)
  gamma[2,1] <- 0         # Pr(dead t -> alive t+1)
  gamma[2,2] <- 1         # Pr(dead t -> dead t+1)
  omega[1,1] <- 1 - p     # Pr(alive t -> non-detected t)
  omega[1,2] <- p         # Pr(alive t -> detected t)
  omega[2,1] <- 1         # Pr(dead t -> non-detected t)
  omega[2,2] <- 0         # Pr(dead t -> detected t)
  for (i in 1:N){
    y[i,1:T] ~ dHMM(probInit = delta[1:2],
                    probObs = omega[1:2,1:2], # observation matrix
                    probTrans = gamma[1:2,1:2], # transition matrix
                    len = T) # nb of sampling occasions
  }
})
# constants
my.constants <- list(N = nrow(y), T = 5)
# data
my.data <- list(y = y + 1)
# initial values - no need to specify values for z anymore
```

```
initial.values <- function() list(phi = runif(1,0,1),
                                  p = runif(1,0,1))
# parameters to save
parameters.to.save <- c("phi", "p")
# MCMC details
n.iter <- 5000
n.burnin <- 1000
n.chains <- 2
```

And run NIMBLE:

```
start_time <- Sys.time()
mcmc.output <- nimbleMCMC(code = hmm.survival,
                          constants = my.constants,
                          data = my.data,
                          inits = initial.values,
                          monitors = parameters.to.save,
                          niter = n.iter,
                          nburnin = n.burnin,
                          nchains = n.chains)
## Registering the following user-provided distributions: dHMM
## NIMBLE has registered dHMM as a distribution based on its use in BUGS code. Note that if yo
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
end_time <- Sys.time()
end_time - start_time
## Time difference of 27.96 secs
```

The numerical summaries are similar to those we obtained with the complete likelihood, and effective samples sizes are larger denoting better mixing:

```
MCMCsummary(mcmc.output, round = 2)
##      mean   sd 2.5%  50% 97.5% Rhat n.eff
## p    0.61 0.06 0.49 0.61  0.72    1  1211
## phi  0.76 0.04 0.67 0.76  0.84    1  1483
```

3.9 Pooled encounter histories

We can go one step further to make convergence even faster. As mentionned earlier in Section 3.6.4, the likelihood of an HMM fitted to capture-recapture data often involves individuals that share the same encounter histories. Instead of repeating the same calculations several times, the likelihood contribution that is shared by say x individuals is raised to the power x in the likelihood of the whole dataset, hence making the same operations only once[7].

In this section, we amend the NIMBLE functions we wrote for marginalizing latent states in Section 3.8 to express the likelihood using pooled encounter histories. We use a vector `size` that contains the number of individuals with the same encounter history.

The density function is the function `dHMM` to which we add a `size` argument, and raise the individual likelihood to the power `size`, or multiply by `size` as we work on the log scale `log(sum(alpha[1:2])) * size`:

```
dHMMpooled <- nimbleFunction(
  run = function(x = double(1),
                 probInit = double(1),
                 probObs = double(2),
                 probTrans = double(2),
```

[7] This idea is used in routine in capture-recapture software like MARK or E-SURGE. For Bayesian software however, it is only recently that the trick was tested in NIMBLE (in Turek, de Valpine, and Paciorek (2016). Efficient Markov chain Monte Carlo sampling for hierarchical hidden Markov models. *Environmental and Ecological Statistics* 23: 549-564. Many thanks to Chloé Nater for showing me how to implement it.

```
                len = double(0),
                size = double(0),
                log = integer(0, default = 0)) {
    alpha <- probInit[1:2]
    for (t in 2:len) {
      alpha[1:2] <- (alpha[1:2] %*% probTrans[1:2,1:2]) * probObs[1:2,x[t]]
    }
    logL <- log(sum(alpha[1:2])) * size
    returnType(double(0))
    if (log) return(logL)
    return(exp(logL))
  }
)
```

The `rHMM` function is renamed `rHMMpooled` for compatibility but remains unchanged:

```
rHMMpooled <- nimbleFunction(
  run = function(n = integer(),
                probInit = double(1),
                probObs = double(2),
                probTrans = double(2),
                len = double(0),
                size = double(0)) {
    returnType(double(1))
    z <- numeric(len)
    z[1] <- rcat(n = 1, prob = probInit[1:2]) # all individuals alive at t = 0
    y <- z
    y[1] <- 2 # all individuals are detected at t = 0
    for (t in 2:len){
      # state at t given state at t-1
      z[t] <- rcat(n = 1, prob = probTrans[z[t-1],1:2])
      # observation at t given state at t
      y[t] <- rcat(n = 1, prob = probObs[z[t],1:2])
    }
```

```
    return(y)
  })
```

We assign these two function to the global R environment so that we can use them:

```
assign('dHMMpooled', dHMMpooled, .GlobalEnv)
assign('rHMMpooled', rHMMpooled, .GlobalEnv)
```

You can now plug your pooled HMM density function in your NIMBLE code:

```
hmm.survival <- nimbleCode({
  phi ~ dunif(0, 1) # prior survival
  p ~ dunif(0, 1) # prior detection
  # likelihood
  delta[1] <- 1           # Pr(alive t = 1) = 1
  delta[2] <- 0           # Pr(dead t = 1) = 0
  gamma[1,1] <- phi       # Pr(alive t -> alive t+1)
  gamma[1,2] <- 1 - phi   # Pr(alive t -> dead t+1)
  gamma[2,1] <- 0         # Pr(dead t -> alive t+1)
  gamma[2,2] <- 1         # Pr(dead t -> dead t+1)
  omega[1,1] <- 1 - p     # Pr(alive t -> non-detected t)
  omega[1,2] <- p         # Pr(alive t -> detected t)
  omega[2,1] <- 1         # Pr(dead t -> non-detected t)
  omega[2,2] <- 0         # Pr(dead t -> detected t)
  for (i in 1:N){
    y[i,1:T] ~ dHMMpooled(probInit = delta[1:2],
                          probObs = omega[1:2,1:2], # observation matrix
                          probTrans = gamma[1:2,1:2], # transition matrix
                          len = T, # nb of sampling occasions
                          size = size[i]) # number of individuals with encounter history i
  }
})
```

Before running NIMBLE, we need to actually pool individuals with the same encounter history together:

```
y_pooled <- y %>%
  as_tibble() %>%
  group_by_all() %>% # group
  summarise(size = n()) %>% # count
  relocate(size) %>% # put size in front
  arrange(-size) %>% # sort along size
  as.matrix()
y_pooled
##       size winter 1 winter 2 winter 3 winter 4 winter 5
##  [1,]   21        1        0        0        0        0
##  [2,]    8        1        1        0        0        0
##  [3,]    8        1        1        1        1        0
##  [4,]    4        1        1        0        0        1
##  [5,]    4        1        1        1        0        0
##  [6,]    2        1        0        0        1        0
##  [7,]    2        1        0        1        1        0
##  [8,]    2        1        1        0        1        0
##  [9,]    1        1        0        1        0        0
## [10,]    1        1        0        1        0        1
## [11,]    1        1        0        1        1        1
## [12,]    1        1        1        0        1        1
## [13,]    1        1        1        1        0        1
## [14,]    1        1        1        1        1        1
```

For example, we have 21 individuals with encounter history (1, 0, 0, 0, 0).

Now you can resume the NIMBLE workflow:

```
my.constants <- list(N = nrow(y_pooled), T = 5, size = y_pooled[,'size'])
my.data <- list(y = y_pooled[,-1] + 1) # delete size from dataset
initial.values <- function() list(phi = runif(1,0,1),
                                  p = runif(1,0,1))
parameters.to.save <- c("phi", "p")
```

```
n.iter <- 5000
n.burnin <- 1000
n.chains <- 2
start_time <- Sys.time()
mcmc.output <- nimbleMCMC(code = hmm.survival,
                          constants = my.constants,
                          data = my.data,
                          inits = initial.values,
                          monitors = parameters.to.save,
                          niter = n.iter,
                          nburnin = n.burnin,
                          nchains = n.chains)
## Registering the following user-provided distributions: dHMMpooled
## NIMBLE has registered dHMMpooled as a distribution based on its use in BUGS code. Note that
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
end_time <- Sys.time()
end_time - start_time
## Time difference of 28.69 secs
MCMCsummary(mcmc.output, round = 2)
##     mean   sd 2.5%  50% 97.5% Rhat n.eff
## p   0.61 0.06 0.49 0.61  0.72    1  1453
## phi 0.76 0.04 0.67 0.76  0.84    1  1359
```

The results are the same as those obtained previously. The gain in computation times will be bigger for more complex models as we will see in the next chapters.

3.10 Decoding after marginalization

If you need to infer the latent states, and you cannot afford the computation times of the complete likelihood of Section 3.6.4, you can

still use the marginal likelihood with the forward algorithm of Section 3.8.2. You will need an extra step to decode the latent states with the Viterbi algorithm. The Viterbi algorithm allows you to compute the sequence of states that is most likely to have generated the sequence of observations.

3.10.1 Theory

On peut estimer, ou bien décoder avec Viterbi, voir la différence dans https://luisdamiano.github.io/BayesHMM/articles/introduction.html. Briefly speaking, this algorithm does this and that. Use colors to explain, as for the forward algorithm. See also https://en.wikipedia.org/wiki/Forward_algorithm.

3.10.2 Implementation

Let's write a R function to implement the Viterbi algorithm. As parameters, our function will take the transition and observation matrices, the vector of initial state probabilities and the observed sequence of detections and non-detections for which you aim to compute the sequence of states from which it was most likely generated:

```
# getViterbi() returns sequence of states that most likely generated sequence of observations
# adapted from https://github.com/vbehnam/viterbi
getViterbi <- function(Omega, Gamma, delta, y) {
# Omega: transition matrix
# Gamma: observation matrix
# delta: vector of initial state probabilities
# y: observed sequence of detections and non-detections

# get number of states and sampling occasions
N <- nrow(Gamma)
T <- length(y)

# stateSeq contains the most likely states up until this point
# probSeq is the corresponding likelihood
probSeq <- matrix(0, nrow = N, ncol = T)
```

```
stateSeq <- matrix(0, nrow = N, ncol = T)
firstObs <- y[1]

# fill in first columns of both matrices
#probSeq[,1] <- initial * emission[,firstObs]
#stateSeq[,1] <- 0
probSeq[,1] <- c(1,0) # initial = (1, 0) * emission[,firstObs] = (1, 0)
stateSeq[,1] <- 1 # alive at first occasion

for (i in 2:T) {
    for (j in 1:N) {
      obs <- y[i]
      # initialize to -1, then overwritten by for loop coz all possible values are >= 0
      probSeq[j,i] <- -1
      # loop to find max and argmax for k
      for (k in 1:N) {
        value <- probSeq[k,i-1] * Gamma[k,j] * Omega[j,obs]
        if (value > probSeq[j,i]) {
          # maximizing for k
          probSeq[j,i] <- value
          # argmaximizing for k
          stateSeq[j,i] <- k
        }
      }
    }
  }
  # mlp = most likely path
  mlp <- numeric(T)
  # argmax for stateSeq[,T]
  am <- which.max(probSeq[,T])
  mlp[T] <- stateSeq[am,T]

  # backtrace using backpointers
  for (i in T:2) {
    zm <- which.max(probSeq[,i])
    mlp[i-1] <- stateSeq[zm,i]
```

```
  }
  return(mlp)
}
```

Note that instead of writing your own R function, you could use a built-in function from an existing R package to implement the Viterbi algorithm[8], and call it from NIMBLE as we have seen in Section 2.4.2. The difficulty is that in HMM for capture-recapture data, we have to deal with detection at first encounter, which is not estimated but is one because an individual has to be captured to be marked and released for the first time.

Let's test the `getViterbi()` function with an example. In our simulated dataset, animal #15 has the encounter history (2, 1, 1, 1, 1) which was generated from the sequence of states (1, 1, 2, 2, 2) with survival probability $\phi = 0.8$ and detection probability $p = 0.6$. Applying our function to that animal encounter history, we get:

```
delta # Vector of initial states probabilities
## [1] 1 0
Gamma # Transition matrix
##      [,1] [,2]
## [1,]  0.8  0.2
## [2,]  0.0  1.0
Omega # Observation matrix
##      [,1] [,2]
## [1,]  0.4  0.6
## [2,]  1.0  0.0
getViterbi(Omega = Omega,
           Gamma = Gamma,
           delta = delta,
           y = y[15,] + 1)
## [1] 1 2 2 2 2
```

The Viterbi algorithm does pretty well at recovering the latent states,

[8] For example, the `viterbi()` function from the HMM and depmixS4 packages.

despite falsely decoding a death in the second winter while individual #15 only dies in the third winter.

Now that we have a function that implements the Viterbi algorithm, we can use it with our MCMC outputs. You have two options, either you apply Viterbi to each MCMC iteration then you compute the posterior median or mode path for each individual, or you compute the posterior mean or median of the transition and observation matrices then you apply Viterbi to each individual encounter history.

For both options, we will need the values from the posterior distributions of survival and detection probabilities:

```
phi <- c(mcmc.output$chain1[,'phi'], mcmc.output$chain2[,'phi'])
p <- c(mcmc.output$chain1[,'p'], mcmc.output$chain2[,'p'])
```

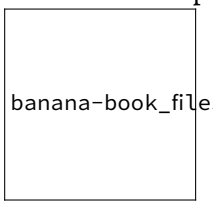
3.10.3 Compute first, average after

First option is to apply Viterbi to each MCMC sample, then to compute median of the MCMC Viterbi paths for each observed sequence:

```
niter <- length(p)
T <- 5
res <- matrix(NA, nrow = nrow(y), ncol = T)
for (i in 1:nrow(y)){
  res_mcmc <- matrix(NA, nrow = niter, ncol = T)
  for (j in 1:niter){
    # Initial states
    delta <- c(1, 0)
    # Transition matrix
    transition <- matrix(NA, 2, 2)
    transition[1,1] <- phi[j]      # Pr(alive t -> alive t+1)
    transition[1,2] <- 1 - phi[j]  # Pr(alive t -> dead t+1)
    transition[2,1] <- 0        # Pr(dead t -> alive t+1)
    transition[2,2] <- 1        # Pr(dead t -> dead t+1)
    # Observation matrix
    emission <- matrix(NA, 2, 2)
```

```
    emission[1,1] <- 1 - p[j]      # Pr(alive t -> non-detected t)
    emission[1,2] <- p[j]          # Pr(alive t -> detected t)
    emission[2,1] <- 1           # Pr(dead t -> non-detected t)
    emission[2,2] <- 0           # Pr(dead t -> detected t)
    res_mcmc[j,1:T] <- getViterbi(emission, transition, delta, y[i,] + 1)
  }
  res[i, 1:length(y[1,])] <- apply(res_mcmc, 2, median)
}
```

You can compare the Viterbi decoding to the actual states z:



Decoding is on overall correct except that the alive actual state is often decoded as the dead state by the Viterbi algorithm. Note that here we compute the Viterbi paths after we run NIMBLE. You could write a NIMBLE function inspired by the R function `getViterbi()` and plug it in your model code to apply Viterbi. This does not make any difference except perhaps to increase MCMC computation times.

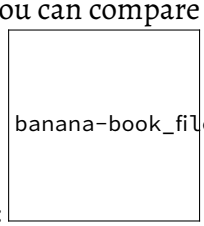
3.10.4 Average first, compute after

Second option is to compute the posterior mean of the observation and transition matrices, then to apply Viterbi:

```
# Initial states
delta <- c(1, 0)
# Transition matrix
transition <- matrix(NA, 2, 2)
transition[1,1] <- mean(phi)      # Pr(alive t -> alive t+1)
transition[1,2] <- 1 - mean(phi)  # Pr(alive t -> dead t+1)
transition[2,1] <- 0              # Pr(dead t -> alive t+1)
transition[2,2] <- 1              # Pr(dead t -> dead t+1)
```

```
# Observation matrix
emission <- matrix(NA, 2, 2)
emission[1,1] <- 1 - mean(p)      # Pr(alive t -> non-detected t)
emission[1,2] <- mean(p)          # Pr(alive t -> detected t)
emission[2,1] <- 1                # Pr(dead t -> non-detected t)
emission[2,2] <- 0                # Pr(dead t -> detected t)
res <- matrix(NA, nrow = nrow(y), ncol = T)
for (i in 1:nrow(y)){
  res[i, 1:length(y[1,]) ] <- getViterbi(emission, transition, delta, y[i,] + 1)
}
```

Again, you can compare the result of the Viterbi decoding to the actual states z:



The results are very similar to those we obtained in Section 3.10.3.

3.11 Summary

- A HMM is a model that consists of two parts: i) an unobserved sequence of discrete random variables - the states - satisfying the Markovian property (future states depends on current states only and not on past states) and ii) an observed sequence of discrete random variables - the observations - depending only on the current state.
- The Bayesian approach together with MCMC simulations allow estimating survival and detection probabilities as well as individual latent states alive or dead with the complete likelihood. If you can afford the computation times, then using the complete likelihood is the easiest path for model fitting.

- If you do not need to infer the latent states, you can use the marginal likelihood via the forward algorithm. By avoiding to sample the latent states, you usually get better mixing and faster convergence.
- If you do need to infer the latent states, and you cannot afford the computation times of the complete likelihood, then you can go for the marginal likelihood in conjunction with the Viterbi algorithm to decode the latent states.
- If the computational burden is still an issue, and you have individuals that share the same encounter history, you can use a pooled likelihood to speed up the marginal likelihood evaluation and MCMC convergence.

3.12 Suggested reading

- Jurafsky D. and Martin J.H. (2021) Hidden Markov models[9]. In Speech and Language Processing. 3rd edition. Pearson Education UK, 2021.
- McClintock B.T., Langrock R., Gimenez O., Cam E., Borchers D.L., Glennie R. and Patterson T.A. (2020), Uncovering ecological state dynamics with hidden Markov models[10]. Ecology Letters, 23: 1878-1903.
- Rabiner L.R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition[11]. Proceedings of the IEEE, 77:257-286.
- Zucchini W., MacDonald I.L. and Langrock R. (2016) Hidden Markov Models for Time Series: An Introduction Using R (2nd ed)[12]. Chapman and Hall/CRC.

[9] https://web.stanford.edu/~jurafsky/slp3/A.pdf

[10] https://onlinelibrary.wiley.com/doi/full/10.1111/ele.13610

[11] https://web.ece.ucsb.edu/Faculty/Rabiner/ece259/Reprints/tutorial%20on%20hmm%20and%20applications.pdf

[12] https://www.routledge.com/Hidden-Markov-Models-for-Time-Series-An-Introduction-Using-R-Second-Edition/Zucchini-MacDonald-Langrock/p/book/9781482253832

Part II

II. Transitions

Introduction

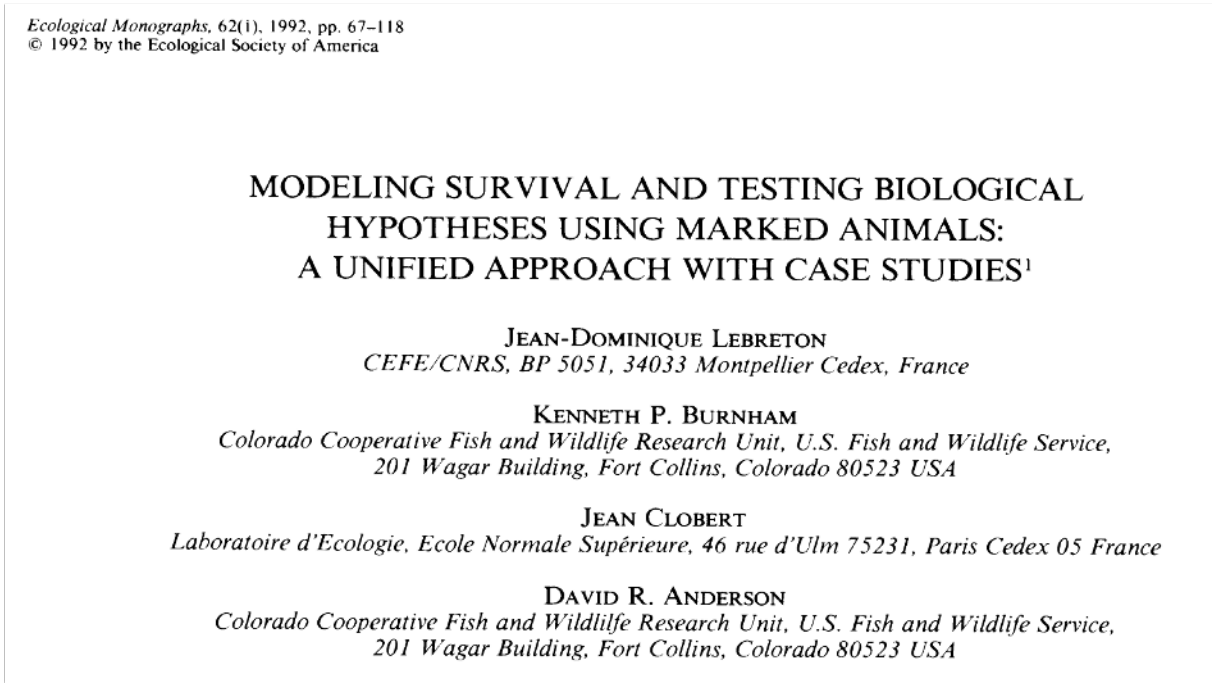
4

Survival

4.1 Introduction

Blabla.

```
knitr::include_graphics("images/lebreton.png")
```



Ecological Monographs, 62(1), 1992, pp. 67–118

MODELING SURVIVAL AND TESTING BIOLOGICAL HYPOTHESES USING MARKED ANIMALS: A UNIFIED APPROACH WITH CASE STUDIES[1]

JEAN-DOMINIQUE LEBRETON
CEFE/CNRS, BP 5051, 34033 Montpellier Cedex, France

KENNETH P. BURNHAM
Colorado Cooperative Fish and Wildlife Research Unit, U.S. Fish and Wildlife Service, 201 Wagar Building, Fort Collins, Colorado 80523 USA

JEAN CLOBERT
Laboratoire d'Ecologie, Ecole Normale Supérieure, 46 rue d'Ulm 75231, Paris Cedex 05 France

DAVID R. ANDERSON
Colorado Cooperative Fish and Wildlilfe Research Unit, U.S. Fish and Wildlife Service, 201 Wagar Building, Fort Collins, Colorado 80523 USA

4.2 History of the Cormack-Jolly-Seber (CJS) model

S.T. Buckland (2016). A Conversation with Richard M. Cormack. Statistical Science 31: 142-150.

Buckland: George Jolly was a colleague of yours in the 1960s. Could you describe your interactions with him?

Cormack: George was in the ARC Unit of Statistics in the same corridor as I was. His main job was designing and conducting agricultural surveys in Scotland. There wasn't a practice of giving seminars in the department to talk to colleagues about what one was doing, and David Finney's appointees had been chosen to cover all the varied areas of statistics rather than build a research group in a particular area. So despite the fact that I met George every day at coffee, and, indeed, we caused David a lot of angst as, on many mornings, we played kriegspiel (a version of chess where you don't see the other person's board and a referee judges—very good for developing inference), we never mentioned work and mark-recapture. I don't remember George noticing my Biometrika paper in 1964 (Cormack, 1964), or indeed the practical paper in British Birds in 1963 (Dunnet, Anderson and Cormack, 1963). It was completely unknown to the two of us that we were working in the same area.

Buckland: What interactions did you have with George Seber?

Cormack: Before the 1965 papers (Jolly, 1965; Seber, 1965), George Seber and I had no contact whatsoever. After the papers, yes, we did. We got into deep communication after the first papers, and he was all for sending me drafts of everything he did. He produced stuff at a colossal rate and his encyclopaedic knowledge was unbelievable. I'm not sure he ever actually worked closely with biologists, but, when he was writing his book, he asked if I would comment on the draft chapters on the bits I knew about. But you have to realise that communication between opposite corners of the world took time. At one point, I received a plaintive handwritten letter saying "The University has cut down on postage and I'm not allowed to post the draft chapter airmail and you will have to wait for it to come by surface mail from New Zealand." By the time it arrived, I already had another airmail letter from him saying, "I'm sorry you haven't been able to comment on the chapter—I've had to submit it!" To some extent, the opposite is true now: response is too quick.

Interface Focus

Interface Focus (2012) **2**, 190–204
doi:10.1098/rsfs.2011.0078
Published online 25 January 2012

Review

A review of Bayesian state-space* modelling of capture–recapture–recovery data

Ruth King*

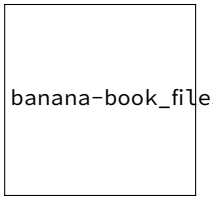
School of Mathematics and Statistics and Centre for Research into Ecological and Environmental Modelling, University of St Andrews, St Andrews, Fife KY16 9LZ, UK

Traditionally, state-space models are fitted to data where there is uncertainty in the observation or measurement of the system. State-space models are partitioned into an underlying system process describing the transitions of the true states of the system over time and the observation process linking the observations of the system to the true states. Open population capture–recapture–recovery data can be modelled in this framework by regarding the system process as the state of each individual observed within the study in terms of being alive or dead, and the observation process the recapture and/or recovery process. The traditional observation error of a state-space model is incorporated via the recapture/recovery probabilities being less than unity. The models can be fitted using a Bayesian data augmentation approach and in standard BUGS packages. Applying this state-space framework to such data permits additional complexities including individual heterogeneity to be fitted to the data at very little additional programming effort. We consider the efficiency of the state-space model fitting approach by considering a random effects model for capture–recapture data relating to dippers and compare different Bayesian model-fitting algorithms within WinBUGS.

* State–space models may include continuous latent states

Bayesian uptake

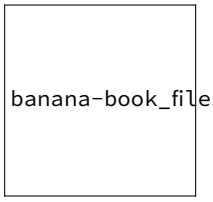
4.3 What we've seen so far



For states (in gray), $z = 1$ is alive, $z = 2$ is dead.

For observations (in white), $y = 1$ is non-detected, $y = 2$ is detected

4.4 In the CJS model, survival and recapture are time-varying



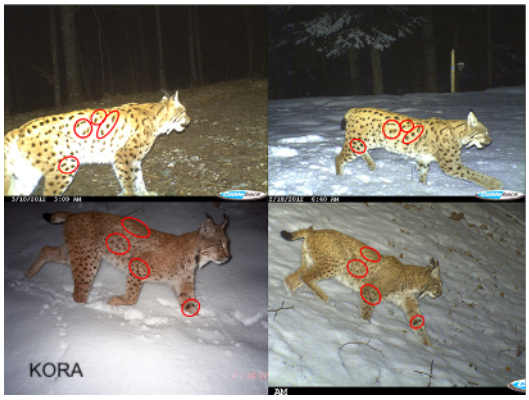
Survival probability is $\phi_t = \Pr(z_{t+1} = 1 | z_t = 1)$.

Recapture (detection) probability is $p_t = \Pr(y_t = 1 | z_t = 1)$.

Accounts for variation in e.g. environmental conditions (survival) or sampling effort (detection).

4.5 Capture, mark and recapture

4.6 Capture, mark and recapture



Artificial marks

Natural marks

4.7 The famous Dipper example

FIGURE 4.1: White-throated Dipper (Cinclus cinclus)

FIGURE 4.2: Gilbert Marzolin

4.8 294 dippers captured and recaptured between 1981 and 1987 with known sex and wing length

year_1981	year_1982	year_1983	year_1984	year_1985	year_1986	year_1987	sex	wing_le
1	1	1	1	1	1	0	M	
1	1	1	1	1	0	0	F	
1	1	1	1	0	0	0	M	
1	1	1	1	0	0	0	F	
1	1	0	1	1	1	0	F	
1	1	0	0	0	0	0	M	
1	1	0	0	0	0	0	M	
1	1	0	0	0	0	0	M	
1	1	0	0	0	0	0	M	
1	1	0	0	0	0	0	F	
1	1	0	0	0	0	0	F	
1	0	1	0	0	0	0	M	
1	0	1	0	0	0	0	F	
1	0	0	0	0	0	0	M	
1	0	0	0	0	0	0	M	
1	0	0	0	0	0	0	M	
1	0	0	0	0	0	0	M	
1	0	0	0	0	0	0	M	
1	0	0	0	0	0	0	F	
1	0	0	0	0	0	0	F	
1	0	0	0	0	0	0	F	
1	0	0	0	0	0	0	F	
0	1	1	1	1	1	1	F	
0	1	1	1	1	1	1	F	
0	1	1	1	1	1	0	F	
0	1	1	1	1	0	0	M	
0	1	1	1	1	0	0	F	
0	1	1	1	1	0	0	F	
0	1	1	1	0	0	0	M	
0	1	1	1	0	0	0	F	
0	1	1	0	1	1	0	F	
0	1	1	0	0	0	0	M	
0	1	1	0	0	0	0	M	
0	1	1	0	0	0	0	M	
0	1	1	0	0	0	0	M	
0	1	1	0	0	0	0	M	
0	1	1	0	0	0	0	M	
0	1	1	0	0	0	0	M	

4.9 Back to Nimble.

4.9.1 Our model so far (ϕ, p)

```
hmm.phip <- nimbleCode({
  phi ~ dunif(0, 1) # prior survival
  p ~ dunif(0, 1) # prior detection
  # likelihood
  gamma[1,1] <- phi      # Pr(alive t -> alive t+1)
  gamma[1,2] <- 1 - phi  # Pr(alive t -> dead t+1)
  gamma[2,1] <- 0        # Pr(dead t -> alive t+1)
  gamma[2,2] <- 1        # Pr(dead t -> dead t+1)
  delta[1] <- 1          # Pr(alive t = 1) = 1
  delta[2] <- 0          # Pr(dead t = 1) = 0
  omega[1,1] <- 1 - p    # Pr(alive t -> non-detected t)
  omega[1,2] <- p        # Pr(alive t -> detected t)
  omega[2,1] <- 1        # Pr(dead t -> non-detected t)
  omega[2,2] <- 0        # Pr(dead t -> detected t)
  for (i in 1:N){
    z[i,first[i]] ~ dcat(delta[1:2])
    for (j in (first[i]+1):T){
      z[i,j] ~ dcat(gamma[z[i,j-1], 1:2])
      y[i,j] ~ dcat(omega[z[i,j], 1:2])
    }
  }
})
```

4.9.2 Our model so far (ϕ, p)

```
##     mean   sd 2.5%  50% 97.5% Rhat n.eff
## phi 0.56 0.03 0.52 0.56  0.62 1.00   500
## p   0.89 0.03 0.83 0.89  0.94 1.13   273
```

4.9.3 The CJS model (ϕ_t, p_t)

```
hmm.phitpt <- nimbleCode({
  delta[1] <- 1          # Pr(alive t = 1) = 1
  delta[2] <- 0          # Pr(dead t = 1) = 0
  for (t in 1:(T-1)){
    phi[t] ~ dunif(0, 1) # prior survival #<<
    gamma[1,1,t] <- phi[t]      # Pr(alive t -> alive t+1)
    gamma[1,2,t] <- 1 - phi[t]  # Pr(alive t -> dead t+1)
    gamma[2,1,t] <- 0        # Pr(dead t -> alive t+1)
    gamma[2,2,t] <- 1        # Pr(dead t -> dead t+1)
    p[t] ~ dunif(0, 1) # prior detection #<<
    omega[1,1,t] <- 1 - p[t]    # Pr(alive t -> non-detected t)
    omega[1,2,t] <- p[t]        # Pr(alive t -> detected t)
    omega[2,1,t] <- 1        # Pr(dead t -> non-detected t)
    omega[2,2,t] <- 0        # Pr(dead t -> detected t)
  }
  # likelihood
  for (i in 1:N){
    z[i,first[i]] ~ dcat(delta[1:2])
    for (j in (first[i]+1):T){
      z[i,j] ~ dcat(gamma[z[i,j-1], 1:2, j-1])
      y[i,j] ~ dcat(omega[z[i,j], 1:2, j-1])
    }
  }
})
```

4.9.4 The CJS model (ϕ_t, p_t)

```
hmm.phitpt <- nimbleCode({
  delta[1] <- 1          # Pr(alive t = 1) = 1
  delta[2] <- 0          # Pr(dead t = 1) = 0
  for (t in 1:(T-1)){ #<<
    phi[t] ~ dunif(0, 1) # prior survival
    gamma[1,1,t] <- phi[t]      # Pr(alive t -> alive t+1)
```

```
    gamma[1,2,t] <- 1 - phi[t]  # Pr(alive t -> dead t+1)
    gamma[2,1,t] <- 0        # Pr(dead t -> alive t+1)
    gamma[2,2,t] <- 1        # Pr(dead t -> dead t+1)
    p[t] ~ dunif(0, 1) # prior detection
    omega[1,1,t] <- 1 - p[t]    # Pr(alive t -> non-detected t)
    omega[1,2,t] <- p[t]        # Pr(alive t -> detected t)
    omega[2,1,t] <- 1        # Pr(dead t -> non-detected t)
    omega[2,2,t] <- 0        # Pr(dead t -> detected t)
  } #<<
  # likelihood
  for (i in 1:N){
    z[i,first[i]] ~ dcat(delta[1:2])
    for (j in (first[i]+1):T){
      z[i,j] ~ dcat(gamma[z[i,j-1], 1:2, j-1])
      y[i,j] ~ dcat(omega[z[i,j], 1:2, j-1])
    }
  }
})
```

4.9.5 The CJS model (ϕ_t, p_t)

```
hmm.phitpt <- nimbleCode({
  delta[1] <- 1          # Pr(alive t = 1) = 1
  delta[2] <- 0          # Pr(dead t = 1) = 0
  for (t in 1:(T-1)){
    phi[t] ~ dunif(0, 1) # prior survival
    gamma[1,1,t] <- phi[t]      # Pr(alive t -> alive t+1) #<<
    gamma[1,2,t] <- 1 - phi[t]  # Pr(alive t -> dead t+1) #<<
    gamma[2,1,t] <- 0        # Pr(dead t -> alive t+1) #<<
    gamma[2,2,t] <- 1        # Pr(dead t -> dead t+1) #<<
    p[t] ~ dunif(0, 1) # prior detection
    omega[1,1,t] <- 1 - p[t]    # Pr(alive t -> non-detected t)
    omega[1,2,t] <- p[t]        # Pr(alive t -> detected t)
    omega[2,1,t] <- 1        # Pr(dead t -> non-detected t)
    omega[2,2,t] <- 0        # Pr(dead t -> detected t)
```

```
  }
  # likelihood
  for (i in 1:N){
    z[i,first[i]] ~ dcat(delta[1:2])
    for (j in (first[i]+1):T){
      z[i,j] ~ dcat(gamma[z[i,j-1], 1:2, j-1])
      y[i,j] ~ dcat(omega[z[i,j], 1:2, j-1])
    }
  }
})
```

4.9.6 The CJS model (ϕ_t, p_t)

```
hmm.phitpt <- nimbleCode({
  delta[1] <- 1           # Pr(alive t = 1) = 1
  delta[2] <- 0           # Pr(dead t = 1) = 0
  for (t in 1:(T-1)){
    phi[t] ~ dunif(0, 1) # prior survival
    gamma[1,1,t] <- phi[t]      # Pr(alive t -> alive t+1)
    gamma[1,2,t] <- 1 - phi[t]  # Pr(alive t -> dead t+1)
    gamma[2,1,t] <- 0        # Pr(dead t -> alive t+1)
    gamma[2,2,t] <- 1        # Pr(dead t -> dead t+1)
    p[t] ~ dunif(0, 1) # prior detection
    omega[1,1,t] <- 1 - p[t]    # Pr(alive t -> non-detected t) #<<
    omega[1,2,t] <- p[t]        # Pr(alive t -> detected t) #<<
    omega[2,1,t] <- 1        # Pr(dead t -> non-detected t) #<<
    omega[2,2,t] <- 0        # Pr(dead t -> detected t) #<<
  }
  # likelihood
  for (i in 1:N){
    z[i,first[i]] ~ dcat(delta[1:2])
    for (j in (first[i]+1):T){
      z[i,j] ~ dcat(gamma[z[i,j-1], 1:2, j-1])
      y[i,j] ~ dcat(omega[z[i,j], 1:2, j-1])
    }
```

```
  }
})
```

4.9.7 The CJS model (ϕ_t, p_t)

```
hmm.phitpt <- nimbleCode({
  delta[1] <- 1          # Pr(alive t = 1) = 1
  delta[2] <- 0          # Pr(dead t = 1) = 0
  for (t in 1:(T-1)){
    phi[t] ~ dunif(0, 1) # prior survival
    gamma[1,1,t] <- phi[t]      # Pr(alive t -> alive t+1)
    gamma[1,2,t] <- 1 - phi[t]  # Pr(alive t -> dead t+1)
    gamma[2,1,t] <- 0        # Pr(dead t -> alive t+1)
    gamma[2,2,t] <- 1        # Pr(dead t -> dead t+1)
    p[t] ~ dunif(0, 1) # prior detection
    omega[1,1,t] <- 1 - p[t]    # Pr(alive t -> non-detected t)
    omega[1,2,t] <- p[t]        # Pr(alive t -> detected t)
    omega[2,1,t] <- 1        # Pr(dead t -> non-detected t)
    omega[2,2,t] <- 0        # Pr(dead t -> detected t)
  }
  # likelihood
  for (i in 1:N){
    z[i,first[i]] ~ dcat(delta[1:2])
    for (j in (first[i]+1):T){
      z[i,j] ~ dcat(gamma[z[i,j-1], 1:2, j-1]) #<<
      y[i,j] ~ dcat(omega[z[i,j], 1:2, j-1]) #<<
    }
  }
})
```

4.9.8 The CJS model (ϕ_t, p_t)

```
##         mean   sd 2.5%  50% 97.5% Rhat n.eff
## phi[1] 0.73 0.14 0.46 0.72  0.99 1.02   199
## phi[2] 0.45 0.07 0.32 0.44  0.59 1.02   410
```

```
## phi[3] 0.48 0.06 0.35 0.48  0.59 1.01   506
## phi[4] 0.63 0.06 0.52 0.63  0.75 1.03   415
## phi[5] 0.60 0.06 0.49 0.60  0.72 1.01   365
## phi[6] 0.74 0.13 0.51 0.74  0.97 1.10    38
## p[1]   0.66 0.14 0.38 0.67  0.89 1.01   344
## p[2]   0.87 0.08 0.68 0.89  0.98 1.02   249
## p[3]   0.88 0.07 0.73 0.89  0.97 1.02   307
## p[4]   0.87 0.06 0.74 0.88  0.96 1.05   333
## p[5]   0.90 0.05 0.77 0.91  0.98 1.01   224
## p[6]   0.72 0.13 0.50 0.72  0.97 1.08    37
```

4.9.9 Time-varying survival (ϕ_t, p)

```
hmm.phitp <- nimbleCode({
  for (t in 1:(T-1)){
    phi[t] ~ dunif(0, 1) # prior survival
    gamma[1,1,t] <- phi[t]      # Pr(alive t -> alive t+1)
    gamma[1,2,t] <- 1 - phi[t]  # Pr(alive t -> dead t+1)
    gamma[2,1,t] <- 0        # Pr(dead t -> alive t+1)
    gamma[2,2,t] <- 1        # Pr(dead t -> dead t+1)
  }
  p ~ dunif(0, 1) # prior detection
  delta[1] <- 1          # Pr(alive t = 1) = 1
  delta[2] <- 0          # Pr(dead t = 1) = 0
  omega[1,1] <- 1 - p    # Pr(alive t -> non-detected t)
  omega[1,2] <- p        # Pr(alive t -> detected t)
  omega[2,1] <- 1        # Pr(dead t -> non-detected t)
  omega[2,2] <- 0        # Pr(dead t -> detected t)
  # likelihood
  for (i in 1:N){
    z[i,first[i]] ~ dcat(delta[1:2])
    for (j in (first[i]+1):T){
      z[i,j] ~ dcat(gamma[z[i,j-1], 1:2, j-1])
      y[i,j] ~ dcat(omega[z[i,j], 1:2])
    }
```

```
  }
})
```

]

4.9.10 Time-varying survival (ϕ_t, p)

```
##         mean   sd 2.5%  50% 97.5% Rhat n.eff
## phi[1] 0.63 0.10 0.42 0.63  0.82 1.04   564
## phi[2] 0.46 0.06 0.35 0.46  0.59 1.01   629
## phi[3] 0.48 0.05 0.37 0.48  0.59 1.00   610
## phi[4] 0.62 0.06 0.51 0.62  0.73 1.00   553
## phi[5] 0.61 0.05 0.50 0.61  0.72 1.00   568
## phi[6] 0.59 0.05 0.48 0.59  0.69 1.03   463
## p      0.89 0.03 0.82 0.89  0.95 1.04   211
```

4.9.11 Time-varying detection (ϕ, p_t)

```
hmm.phipt <- nimbleCode({
  phi ~ dunif(0, 1) # prior survival
  gamma[1,1] <- phi      # Pr(alive t -> alive t+1)
  gamma[1,2] <- 1 - phi  # Pr(alive t -> dead t+1)
  gamma[2,1] <- 0        # Pr(dead t -> alive t+1)
  gamma[2,2] <- 1        # Pr(dead t -> dead t+1)
  delta[1] <- 1          # Pr(alive t = 1) = 1
  delta[2] <- 0          # Pr(dead t = 1) = 0
  for (t in 1:(T-1)){
    p[t] ~ dunif(0, 1) # prior detection
    omega[1,1,t] <- 1 - p[t]    # Pr(alive t -> non-detected t)
    omega[1,2,t] <- p[t]        # Pr(alive t -> detected t)
    omega[2,1,t] <- 1        # Pr(dead t -> non-detected t)
    omega[2,2,t] <- 0        # Pr(dead t -> detected t)
  }
  # likelihood
  for (i in 1:N){
```

```
    z[i,first[i]] ~ dcat(delta[1:2])
    for (j in (first[i]+1):T){
      z[i,j] ~ dcat(gamma[z[i,j-1], 1:2])
      y[i,j] ~ dcat(omega[z[i,j], 1:2, j-1])
    }
  }
})
```

4.9.12 Time-varying detection (ϕ, p_t)

```
##      mean   sd 2.5%  50% 97.5% Rhat n.eff
## phi  0.56 0.03 0.52 0.56  0.61 1.02   381
## p[1] 0.75 0.12 0.48 0.77  0.93 1.03   452
## p[2] 0.85 0.08 0.68 0.86  0.97 1.02   359
## p[3] 0.85 0.07 0.69 0.85  0.96 1.00   316
## p[4] 0.89 0.05 0.77 0.89  0.97 1.00   412
## p[5] 0.91 0.04 0.82 0.92  0.98 1.00   376
## p[6] 0.90 0.07 0.73 0.91  1.00 1.07   111
```

4.10 Why Bayes? Incorporate prior information.

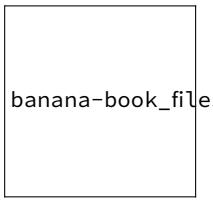
4.11 Vague prior

So far, we have assumed a vague prior:

$$\phi_{prior} \sim \text{Beta}(1, 1) = \text{Uniform}(0, 1)$$

With a vague prior, mean posterior survival is $\phi_{posterior} = 0.56$

With credible interval $[0.52, 0.62]$



Posterior distribution of survival in color (two chains), prior in gray dashed line.

4.12 How to incorporate prior information?

Using information on body mass and annual survival of 27 European passerines, we can predict survival of European dippers using only body mass.

For dippers, body mass is 59.8g, therefore $\phi = 0.57$ with sd $= 0.073$.

Assuming an informative prior $\phi_{prior} \sim$ Normal$(0.57, 0.073^2)$.

Mean posterior $\phi_{posterior} = 0.56$ with credible interval $[0.52, 0.61]$.

No increase of precision in posterior inference.

4.13 How to incorporate prior information?

Now if you had only the three first years of data, what would have happened?

Width of credible interval is 0.53 (vague prior) vs. 0.24 (informative prior).

Huge increase of precision in posterior inference, a 120% gain!

4.13.1 Compare survival posterior with and without informative prior

4.14 Prior elicitation via moment matching

The prior $\phi_{prior} \sim \text{Normal}(0.57, 0.073^2)$ is not entirely satisfying

Remember the Beta distribution

Recall that the Beta distribution is a continuous distribution with values between 0 and 1. Useful for modelling survival or detection probabilities.

If $X \sim Beta(\alpha, \beta)$, then the first and second moments of X are:

$$\mu = \text{E}(X) = \frac{\alpha}{\alpha + \beta}$$

$$\sigma^2 = \text{Var}(X) = \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}$$

4.15 Moment matching

In the capture-recapture example, we know a priori that the mean of the probability we're interested in is $\mu = 0.57$ and its variance is $\sigma^2 = 0.073^2$. Parameters μ and σ^2 are seen as the moments of a $Beta(\alpha, \beta)$ distribution. Now we look for values of α and β that match the observed moments of the Beta distribution μ and σ^2. We need another set of equations:

$$\alpha = \left(\frac{1-\mu}{\sigma^2} - \frac{1}{\mu} \right) \mu^2$$

$$\beta = \alpha \left(\frac{1}{\mu} - 1 \right)$$

For our model, that means:

```
(alpha <- ( (1 - 0.57)/(0.073*0.073) - (1/0.57) )*0.57^2)
## [1] 25.65
(beta <- alpha * ( (1/0.57) - 1))
## [1] 19.35
```

Now use $\phi_{prior} \sim \text{Beta}(\alpha = 25.6, \beta = 19.3)$ instead of $\phi_{prior} \sim \text{Normal}(0.57, 0.073^2)$

4.16 Prior predictive checks

4.16.1 Linear regression

Unreasonable prior $\beta \sim N(0, 1000^2)$

Reasonable prior $\beta \sim N(2, 0.5^2)$

4.16.2 Logistic regression

Unreasonable prior $\text{logit}(\phi) = \beta \sim N(0, 10^2)$

Reasonable prior $\text{logit}(\phi) = \beta \sim N(0, 1.5^2)$

4.17 Capture-recapture models rely on assumptions

Design: No mark lost, Identity of individuals recorded without error (no false positives), Captured individuals are a random sample

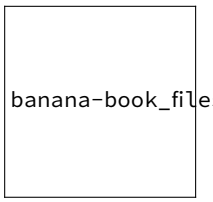
Model: Homogeneity of survival and recapture probabilities, Independence between individuals (overdispersion)

Test validity of assumptions: These assumptions should be valid, whatever inferential framework, Use goodness-of-fit tests — Pradel et al. (2005), R implementation with package R2ucare[1], Posterior predictive checks can also be used (not covered; Gelman et al. 2020[2]). Forward reference to chapter with gof and model selection.

[1] https://besjournals.onlinelibrary.wiley.com/doi/full/10.1111/2041-210X.13014

[2] https://arxiv.org/pdf/2011.01808.pdf

4.17.1 Parameter-redundancy issue



Last survival and recapture probabilities cannot be estimated separately.

Poor mixing of the chains.

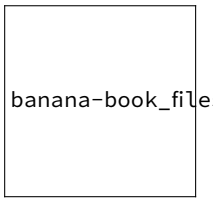
4.18 Parameter redundancy

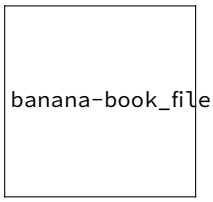
Two issues

Intrinsic redundancy: Likelihood can be expressed by a smaller number of parameters; Feature of the model

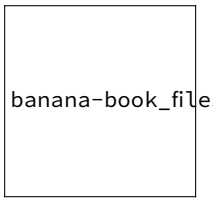
Extrinsic redundancy: Model structure is fine, But lack of data makes a parameter non-estimable, Feature of the data.

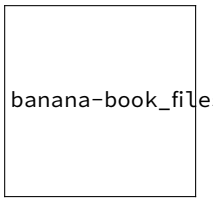
4.19 Prior-posterior overlap for ϕ_4 and ϕ_6





4.20 Prior-posterior overlap for p_3 and p_7





4.21 What does survival actually mean in capture-recapture ?

Survival refers to the study area.

Mortality and permanent emigration are confounded.

Therefore we estimate apparent survival, not true survival.

Apparent survival probability = true survival × study area fidelity.

Consequently, apparent survival < true survival unless study area fidelity = 1.

Use caution with interpretation. If possible, combine with ring-recovery data, or go spatial to get closer to true survival.

4.22 Summary

- Blabla.
- Blabla.

4.23 Suggested reading

- CJS state-space formulation Gimenez et al. (2007)[3] and Royle (2008)[4].
- Work on missing values by Bonner et al. (2006)[5] and Langrock and King (2013)[6] and Worthington et al. (2015)[7].
- The example on how to incorporate prior information is in McCarthy and Masters (2005)[8].
- Combine live recapture w/ dead recoveries by Lebreton et al. (1999)[9] and go spatial to account for emigration Gilroy et al. (2012)[10] and Schaub & Royle (2014)[11].

[3] https://oliviergimenez.github.io/pubs/Gimenezetal2007EcologicalModelling.pdf

[4] https://onlinelibrary.wiley.com/doi/10.1111/j.1541-0420.2007.00891.x

[5] https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1541-0420.2005.00399.x

[6] https://projecteuclid.org/journals/annals-of-applied-statistics/volume-7/issue-3/Maximum-likelihood-estimation-of-markrecapturerecovery-models-in-the-presence-of/10.1214/13-AOAS644.full

[7] https://link.springer.com/article/10.1007/s13253-014-0184-z

[8] https://besjournals.onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-2664.2005.01101.x

[9] https://www.tandfonline.com/doi/pdf/10.1080/00063659909477230

[10] https://esajournals.onlinelibrary.wiley.com/doi/full/10.1890/12-0124.1

[11] https://besjournals.onlinelibrary.wiley.com/doi/full/10.1111/2041-210X.12134

- Non-identifiability in a Bayesian framework, see Gimenez et al. (2009)[12] and book by Cole (2020)[13].

[12] https://oliviergimenez.github.io/pubs/Gimenezetal2009-weakidentifiability.pdf

[13] https://www.routledge.com/Parameter-Redundancy-and-Identifiability/Cole/p/book/9781498720878

5

Dispersal

5.1 Introduction

Blabla.

```
knitr::include_graphics("images/arnason1973.png")
```

Res. Popul. Ecol. (1973) 15, 1—8.

THE ESTIMATION OF POPULATION SIZE, MIGRATION RATES AND SURVIVAL IN A STRATIFIED POPULATION

A. Neil Arnason

Computer Science Department, University of Manitoba, Winnipeg, Canada

Introduction

Chapman and Junge (1956, hereafter referred to as C & J) developed estimates of stratum size and migration rates for a population divided into $n \geq 2$ areas (strata) when animals were free to migrate from area to area. The method was based on data from sampling and marking observations on two occasions. The method was extended by Darroch (1961) to allow sampling in different numbers of strata at the two sampling times, and to show how to treat some special problems that arise when using the method. These problems arise when a particular data matrix (which must be inverted) is singular or ill-conditioned. The same problems could occur with the estimates which will be given in this paper.

In order to account for deaths or losses from the areas due to permanent emigration out of the areas being sampled, it is necessary to sample on at least three occasions. I developed estimates for the three sample experiment on two areas (Arnason 1972a) and later (Arnason 1972b) gave asymptotic variance formulae for the estimates and suggested methods of predicting population sizes or time to extinction, using data from this experiment. The purpose of the present paper is to give the estimates for the general situation of sampling on three occasions in $n \geq 2$ areas. Modified estimates to account for losses on capture (as occur, for instance, in bird-banding studies) are also given.

```
knitr::include_graphics("images/schwarz1993.png")
```

BIOMETRICS 49, 177–193
March 1993

Estimating Migration Rates Using Tag-Recovery Data

Carl J. Schwarz

Department of Statistics, University of Manitoba,
Winnipeg, Manitoba R3T 2N2, Canada

Jake F. Schweigert

Biological Sciences Branch, Pacific Biological Station,
Department of Fisheries and Oceans, Nanaimo, British Columbia V9R 5K6, Canada

and

A. Neil Arnason

Department of Computer Science, University of Manitoba,
Winnipeg, Manitoba R3T 2N2, Canada

SUMMARY

Tag-recovery data are used to estimate migration rates among a set of strata. The model formulation is a simple matrix extension of the formulation of a tag-recovery experiment discussed by Brownie et al. (1985, *Statistical Inference from Band-Recovery Data—A Handbook*, 2nd edition, Washington, D.C.: U.S. Department of the Interior). Estimation is more difficult because of the convolution of parameters between release and recovery and this convolution may cause estimates of the survival/migration parameters to have low precision. Derived parameters of emigration, immigration, harvest derivation, and overall net survival are also estimated. The models are applied to estimate the migration of Pacific herring among spawning grounds off the west coast of Canada. If animals can be re-released after being recaptured, the model corresponds, in its migration/survival components, to that of Arnason (1972, *Researches in Population Ecology* **13,** 97–113). This correspondence is developed, leading to more efficient estimators of these parameters.

```
knitr::include_graphics("images/deadpool.gif")
```

Thank you Canada!

```
knitr::include_graphics("images/nichols.png")
```

Ecology, 73(1), 1992, pp. 306–312

ESTIMATING TRANSITION PROBABILITIES FOR
BASED POPULATION PROJECTION MATRICES
CAPTURE–RECAPTURE DATA[1]

JAMES D. NICHOLS AND JOHN R. SAUER
United States Fish and Wildlife Service, Patuxent Wildlife Research Center, Laurel,

KENNETH H. POLLOCK
Institute of Statistics, North Carolina State University, Box 8203, Raleigh, North Caro

JAY B. HESTBECK[2]
United States Fish and Wildlife Service, Patuxent Wildlife Research Center, Laurel,

Abstract. In stage-based demography, animals are often categorized i
classes, and size-based probabilities of surviving and changing mass clas
mated before demographic analyses can be conducted. In this paper,
procedures for the estimation of mass transition probabilities from captur
The first approach uses a multistate capture–recapture model that is paran
with the transition probabilities of interest. Maximum likelihood estima
tained numerically using program SURVIV. The second approach involv
of Pollock's robust design. Estimation proceeds by conditioning on ani
particular class at time i, and then using closed models to estimate the
that are alive in other classes at $i + 1$. Both methods are illustrated b
meadow vole, *Microtus pennsylvanicus*, capture–recapture data. The two m
reasonable estimates that were similar. Advantages of these two approa
directness of estimation, the absence of need for restrictive assumptions
pendence of survival and growth, the testability of assumptions, and
related hypotheses of ecological interest (e.g., the hypothesis of tempo
transition probabilities).

Key words: capture–recapture models; Microtus pennsylvanicus; *multistate
estimation; Pollock's robust design; stage-based population projection matrices; sta
abilities.*

5.2 Wintering site fidelity in Canada Geese

5.2.1 3 sites Carolinas, Chesapeake, Mid-Atlantic,

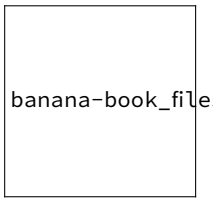
with 21277 banded geese, data kindly provided by Jay Hestbeck (Hestbeck et al. 1991[1])

[1] https://esajournals.onlinelibrary.wiley.com/doi/10.2307/2937193

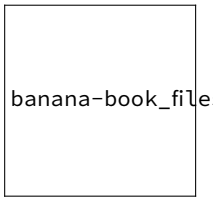
year_1984	year_1985	year_1986	year_1987	year_1988	year_1989
0	2	2	0	0	0
0	0	0	0	0	2
0	0	0	1	0	0
0	0	2	0	0	0
0	3	0	0	3	2
0	0	0	2	0	0
2	2	0	2	3	2
0	0	0	0	2	2
0	0	0	1	0	0
0	2	0	0	0	0
0	2	0	0	3	2
0	0	3	0	0	0
0	0	2	0	0	0
2	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
3	0	0	0	0	0
0	0	0	0	3	1
0	0	2	0	2	0
0	0	0	2	0	0
0	0	0	0	2	2
0	1	0	0	0	0
0	3	0	0	0	0
0	0	0	0	2	0
0	0	0	0	2	0
0	0	0	2	0	0
0	0	1	0	0	0
0	0	0	2	0	3
0	0	3	0	0	0
0	3	3	0	3	3
0	0	0	1	0	0
0	0	0	1	1	0
0	0	2	2	0	2
0	0	0	3	3	0
0	0	0	2	0	0
2	0	0	0	0	0
0	1	0	0	0	0
0	0	0	1	0	0
0	0	2	2	0	0
0	2	0	0	0	0
3	0	0	0	0	0
0	2	0	0	0	0
0	0	3	2	0	0

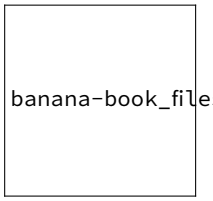
(large areas along East coast of US)

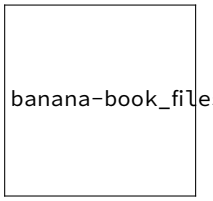
5.2.2 Biological inference



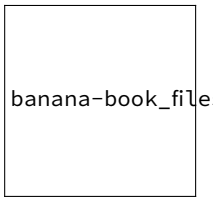
Observations and states are closely related, but not entirely.







5.2.3 The model construction: How we should think.



Generative model. States generate observations.

5.2.4 The model construction: How we should think.

5.2.5 The model construction: How we should think.

5.2.6 The model construction: How we should think.

5.2.7 HMM model for dispersal with 2 sites (drop Carolinas)

Transition matrix

$$\mathbf{\Gamma} = \begin{array}{c} \begin{array}{ccc} z_t = A & z_t = B & z_t = D \end{array} \\ \begin{pmatrix} \phi_A(1-\psi_{AB}) & \phi_A\psi_{AB} & 1-\phi_A \\ \phi_B\psi_{BA} & \phi_B(1-\psi_{BA}) & 1-\phi_B \\ 0 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{l} z_{t-1} = A \\ z_{t-1} = B \\ z_{t-1} = D \end{array}$$

5.2.8 HMM model for dispersal with 2 sites (drop Carolinas)

Observation matrix

$$\mathbf{\Omega} = \begin{array}{c} \begin{matrix} y_t = 0 & y_t = 1 & y_t = 2 \end{matrix} \\ \begin{pmatrix} 1 - p_A & p_A & 0 \\ 1 - p_B & 0 & p_B \\ 1 & 0 & 0 \end{pmatrix} \end{array} \begin{matrix} \\ z_t = A \\ z_t = B \\ z_t = D \end{matrix}$$

5.2.9 HMM model for dispersal with 2 sites (drop Carolinas)

Observation matrix

$$\mathbf{\Omega} = \begin{array}{c} \begin{matrix} y_t = 0 & y_t = 1 & y_t = 2 \end{matrix} \\ \begin{pmatrix} 1 - p_A & p_A & 0 \\ 1 - p_B & 0 & p_B \\ 1 & 0 & 0 \end{pmatrix} \end{array} \begin{matrix} \\ z_t = A \\ z_t = B \\ z_t = D \end{matrix}$$

Note: You may code non-detections as $y_t = 2$, and the first column in the observation matrix should go last.

Quick answer about the -1 and the important issue of coding states and obs. I did this on purpose, to have folks think about the difference between observations and states (non-detection obs should not be confused with state for dead). This becomes even more crucial when we get to multievent models where several observations may be generated by a single state. I get the intuition argument perfectly, but I'd like them to fight against it at first, then once they're comfortable with the difference, they may code obs/states as they see fit. Let's see how it goes. I agree that we should mention that during the multistate lecture, in the spirit of « you're free to code states and jobs the way you like ». I'll add something.

5.2.10 Our model $(\phi_A, \phi_B, \psi_{AB}, \psi_{BA}, p_A, p_B)$

```
multisite <- nimbleCode({
  # -----------------------------------------------
  # Parameters:
  # phiA: survival probability site A
  # phiB: survival probability site B
  # psiAB: movement probability from site A to site B
```

```
  # psiBA: movement probability from site B to site A
  # pA: recapture probability site A
  # pB: recapture probability site B
  # ------------------------------------------------
  # States (z):
  # 1 alive at A
  # 2 alive at B
  # 3 dead
  # Observations (y):
  # 1 not seen
  # 2 seen at A
  # 3 seen at B
  # ------------------------------------------------
...
```

5.2.11 Our model $(\phi_A, \phi_B, \psi_{AB}, \psi_{BA}, p_A, p_B)$

```
multisite <- nimbleCode({
...
  # Priors
  phiA ~ dunif(0, 1)
  phiB ~ dunif(0, 1)
  psiAB ~ dunif(0, 1)
  psiBA ~ dunif(0, 1)
  pA ~ dunif(0, 1)
  pB ~ dunif(0, 1)
...
```

5.2.12 Our model $(\phi_A, \phi_B, \psi_{AB}, \psi_{BA}, p_A, p_B)$

```
multisite <- nimbleCode({
...
  # initial state probabilities
```

```
  delta[1] <- piA          # Pr(alive in A t = 1)
  delta[2] <- 1 - piA      # Pr(alive in B t = 1)
  delta[3] <- 0            # Pr(dead t = 1) = 0
...
```

Actually, initial state is known exactly. It is alive at site of initial capture, and π_A is just the proportion of individuals first captured in site A, no need to estimate it.

Instead of `z[i,first[i]] ~ dcat(delta[1:3])`, use `z[i,first[i]] <- y[i,first[i]]-1` instead in the likelihood.

Same trick applies to CJS models.

5.2.13 Our model $(\phi_A, \phi_B, \psi_{AB}, \psi_{BA}, p_A, p_B)$

```
multisite <- nimbleCode({
...
  # probabilities of state z(t+1) given z(t)
  # (read as gamma[z(t),z(t+1)] = gamma[fromState,toState])

  gamma[1,1] <- phiA * (1 - psiAB)
  gamma[1,2] <- phiA * psiAB
  gamma[1,3] <- 1 - phiA
  gamma[2,1] <- phiB * psiBA
  gamma[2,2] <- phiB * (1 - psiBA)
  gamma[2,3] <- 1 - phiB
  gamma[3,1] <- 0
  gamma[3,2] <- 0
  gamma[3,3] <- 1
...
```

5.2.14 Our model $(\phi_A, \phi_B, \psi_{AB}, \psi_{BA}, p_A, p_B)$

```
multisite <- nimbleCode({
...
  # probabilities of y(t) given z(t)
  # (read as omega[y(t),z(t)] = omega[Observation,State])

  omega[1,1] <- 1 - pA     # Pr(alive A t -> non-detected t)
  omega[1,2] <- pA         # Pr(alive A t -> detected A t)
  omega[1,3] <- 0          # Pr(alive A t -> detected B t)
  omega[2,1] <- 1 - pB     # Pr(alive B t -> non-detected t)
  omega[2,2] <- 0          # Pr(alive B t -> detected A t)
  omega[2,3] <- pB         # Pr(alive B t -> detected B t)
  omega[3,1] <- 1          # Pr(dead t -> non-detected t)
  omega[3,2] <- 0          # Pr(dead t -> detected A t)
  omega[3,3] <- 0          # Pr(dead t -> detected B t)
...
```

5.2.15 Our model $(\phi_A, \phi_B, \psi_{AB}, \psi_{BA}, p_A, p_B)$

```
multisite <- nimbleCode({
...
  # likelihood
  for (i in 1:N){
    # latent state at first capture
    z[i,first[i]] <- y[i,first[i]] - 1
    for (t in (first[i]+1):K){
      # z(t) given z(t-1)
      z[i,t] ~ dcat(gamma[z[i,t-1],1:3])
      # y(t) given z(t)
      y[i,t] ~ dcat(omega[z[i,t],1:3])
    }
  }
})
```

```
##       mean   sd 2.5%  50% 97.5% Rhat n.eff
## pA    0.53 0.09 0.36 0.52  0.73 1.04   122
## pB    0.40 0.04 0.32 0.40  0.48 1.07   165
## phiA  0.60 0.05 0.50 0.60  0.71 1.01   195
## phiB  0.69 0.04 0.62 0.69  0.76 1.04   199
## psiAB 0.27 0.06 0.16 0.26  0.40 1.04   244
## psiBA 0.07 0.02 0.04 0.07  0.12 1.03   360
```

5.3 What if there are three sites?

The transition probabilities still need to be between 0 and 1.

Another constraint is that the sum of three probabilities of departure from a given site should be one.

Two methods to fulfill both constraints. Dirichlet prior and multinomial logit link.

Dirichlet prior with parameter alpha

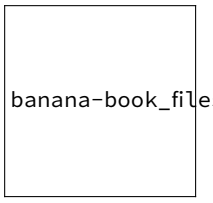


FIGURE 5.1: Dirichlet prior with parameter alpha

5.3.1 Nimble implementation of the Dirichlet prior

```
multisite <- nimbleCode({
...
  # transitions: Dirichlet priors
  psiA[1:3] ~ ddirch(alpha[1:3]) # psiAA, psiAB, psiAC
  psiB[1:3] ~ ddirch(alpha[1:3]) # psiBA, psiBB, psiCC
  psiC[1:3] ~ ddirch(alpha[1:3]) # psiCA, psiCB, psiCC
...
```

5.3.2 Nimble implementation of the Dirichlet prior

```
multisite <- nimbleCode({
...
  # probabilities of state z(t+1) given z(t)
  gamma[1,1] <- phiA * psiA[1]
  gamma[1,2] <- phiA * psiA[2]
  gamma[1,3] <- phiA * psiA[3]
  gamma[1,4] <- 1 - phiA
  gamma[2,1] <- phiB * psiB[1]
  gamma[2,2] <- phiB * psiB[2]
  gamma[2,3] <- phiB * psiB[3]
  gamma[2,4] <- 1 - phiB
  gamma[3,1] <- phiC * psiC[1]
  gamma[3,2] <- phiC * psiC[2]
  gamma[3,3] <- phiC * psiC[3]
  gamma[3,4] <- 1 - phiC
  gamma[4,1] <- 0
  gamma[4,2] <- 0
  gamma[4,3] <- 0
  gamma[4,4] <- 1
...
```

```
##         mean   sd 2.5%  50% 97.5% Rhat n.eff
## pA      0.50 0.09 0.34 0.50  0.70 1.00   153
```

```
## pB      0.47 0.05 0.38 0.46  0.58 1.01   152
## pC      0.24 0.06 0.14 0.23  0.37 1.01   117
## phiA    0.61 0.05 0.50 0.61  0.71 1.00   230
## phiB    0.70 0.04 0.62 0.70  0.77 1.04   183
## phiC    0.77 0.07 0.64 0.77  0.92 1.07   104
## psiA[1] 0.75 0.05 0.63 0.75  0.84 1.01   463
## psiA[2] 0.23 0.05 0.14 0.22  0.34 1.01   441
## psiA[3] 0.02 0.02 0.00 0.02  0.08 1.03   201
## psiB[1] 0.07 0.02 0.04 0.07  0.12 1.00   275
## psiB[2] 0.83 0.04 0.72 0.83  0.90 1.04   129
## psiB[3] 0.10 0.04 0.04 0.09  0.18 1.06   129
## psiC[1] 0.02 0.01 0.00 0.02  0.06 1.00   624
## psiC[2] 0.21 0.05 0.12 0.21  0.33 1.02   420
## psiC[3] 0.77 0.06 0.64 0.77  0.86 1.02   419
```

5.3.3 Multinomial logit

Say we have P sites or states.

Specify a normal prior distribution for $P-1$ transition parameters α_j. These probabilities are on the multinomial logit scale, possibly function of covariates.

To back-transform these parameters, we use:

$$\beta_j = \frac{\exp(\alpha_j)}{1 + \sum_{p=1}^{P} \exp(\alpha_p)}, j = 1, \ldots, P-1$$

This ensures that all β_j are between 0 and 1, and their sum is 1.

Last parameter is calculated as the complement $\beta_P = 1 - \sum_{j=1}^{P-1} \exp(\beta_j)$

5.3.4 Nimble implementation of the Dirichlet prior

```
multisite <- nimbleCode({
...
  # transitions: multinomial logit
  # normal priors on logit of all but one transition probs
  for (i in 1:2){
    lpsiA[i] ~ dnorm(0, sd = 1000)
    lpsiB[i] ~ dnorm(0, sd = 1000)
    lpsiC[i] ~ dnorm(0, sd = 1000)
  }
  # constrain the transitions such that their sum is < 1
  for (i in 1:2){
    psiA[i] <- exp(lpsiA[i]) / (1 + exp(lpsiA[1]) + exp(lpsiA[2]))
    psiB[i] <- exp(lpsiB[i]) / (1 + exp(lpsiB[1]) + exp(lpsiB[2]))
    psiC[i] <- exp(lpsiC[i]) / (1 + exp(lpsiC[1]) + exp(lpsiC[2]))
  }
  # last transition probability
  psiA[3] <- 1 - psiA[1] - psiA[2]
  psiB[3] <- 1 - psiB[1] - psiB[2]
  psiC[3] <- 1 - psiC[1] - psiC[2]
...
```

5.3.5 Nimble implementation of the Dirichlet prior

```
multisite <- nimbleCode({
...
  # probabilities of state z(t+1) given z(t)
  gamma[1,1] <- phiA * psiA[1]
  gamma[1,2] <- phiA * psiA[2]
  gamma[1,3] <- phiA * psiA[3]
  gamma[1,4] <- 1 - phiA
  gamma[2,1] <- phiB * psiB[1]
  gamma[2,2] <- phiB * psiB[2]
  gamma[2,3] <- phiB * psiB[3]
```

```
  gamma[2,4] <- 1 - phiB
  gamma[3,1] <- phiC * psiC[1]
  gamma[3,2] <- phiC * psiC[2]
  gamma[3,3] <- phiC * psiC[3]
  gamma[3,4] <- 1 - phiC
  gamma[4,1] <- 0
  gamma[4,2] <- 0
  gamma[4,3] <- 0
  gamma[4,4] <- 1
...
```

```
##          mean   sd 2.5%  50% 97.5% Rhat n.eff
## pA       0.52 0.08 0.36 0.52  0.69 1.02   154
## pB       0.45 0.05 0.35 0.44  0.55 1.10   129
## pC       0.26 0.06 0.15 0.25  0.39 1.01    94
## phiA     0.60 0.05 0.50 0.60  0.71 1.01   244
## phiB     0.70 0.04 0.63 0.70  0.77 1.11   168
## phiC     0.76 0.07 0.63 0.76  0.88 1.03   126
## psiA[1] 0.76 0.05 0.64 0.76  0.85 1.02   477
## psiA[2] 0.24 0.05 0.15 0.24  0.36 1.01   486
## psiA[3] 0.00 0.00 0.00 0.00  0.00 1.35    47
## psiB[1] 0.07 0.02 0.04 0.06  0.11 1.03   394
## psiB[2] 0.85 0.04 0.77 0.86  0.91 1.04   133
## psiB[3] 0.08 0.03 0.04 0.08  0.16 1.01    79
## psiC[1] 0.01 0.01 0.00 0.01  0.04 1.00   514
## psiC[2] 0.21 0.05 0.12 0.21  0.33 1.00   299
## psiC[3] 0.78 0.06 0.65 0.78  0.88 1.00   270
```

5.4 Sites may be states.

5.5 Examples of multistate models

- *Epidemiological or disease states*: sick/healthy, uninfected/infected/recovered.
- *Morphological states*: small/medium/big, light/medium/heavy.
- *Breeding states*: e.g. breeder/non-breeder, failed breeder, first-time breeder.
- *Developmental or life-history states*: e.g. juvenile/subadult/adult.
- *Social states*: e.g. solitary/group-living, subordinate/dominant.
- *Death states*: e.g. alive, dead from harvest, dead from natural causes.

States = individual, time-specific categorical covariates.

```
knitr::include_graphics("images/sooty.jpg")
```


David Boyle

5.5.1 Sooty shearwater (David Boyle)

5.6 Sooty shearwaters and life-history tradeoffs

We consider data collected between 1940 and 1957 by Lance Richdale on Sooty shearwaters (aka titis).

These data were reanalyzed with multistate models by Scofield et al. (2001)[2] who kindly provided us with the data.

Following the way the data were collected, four states were originally considered: Alive breeder; Accompanied by another bird in a burrow; Alone in a burrow; On the surface; Dead.

5.7 Sooty shearwaters and life-history tradeoffs

Because of numerical issues, we pooled all alive states but breeder together in a non-breeder state (NB) that includes:

- failed breeders (birds that had bred previously – skip reproduction or divorce) and pre-breeders (birds that had yet to breed).
- Note that because burrows were not checked before hatching, some birds in the category NB might have already failed.
- We therefore regard those birds in the B state as successful breeders, and those in the NB state as nonbreeders plus prebreeders and failed breeders.

Observations are non-detections, and detections as breeder and non-breeder

Does breeding affect survival? Does breeding in current year affect breeding next year?

[2] https://link.springer.com/article/10.1198/108571101750524607

year_1942	year_1943	year_1944	year_1949	year_1952	year_1953	year_1956
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	0	1

5.7.1 HMM model for transition between states

Transition matrix

$$\mathbf{\Gamma} = \begin{array}{c} \\ z_{t-1} = B \\ z_{t-1} = NB \\ z_{t-1} = D \end{array} \begin{array}{c} \begin{array}{ccc} z_t = B & z_t = NB & z_t = D \end{array} \\ \begin{pmatrix} \phi_B(1-\psi_{BNB}) & \phi_B\psi_{BNB} & 1-\phi_B \\ \phi_{NB}\psi_{NBB} & \phi_{NB}(1-\psi_{NBB}) & 1-\phi_{NB} \\ 0 & 0 & 1 \end{pmatrix} \end{array}$$

- Costs or reproduction would reflect in future reproduction $\psi_{BB} = 1 - \psi_{BNB} < \psi_{NBB}$ or survival $\phi_B < \phi_{NB}$.

5.7.2 HMM model for transition between states

Observation matrix

$$\mathbf{\Omega} = \begin{array}{c} \\ z_t = B \\ z_t = NB \\ z_t = D \end{array} \begin{array}{c} \begin{array}{ccc} y_t = 0 & y_t = 1 & y_t = 2 \end{array} \\ \begin{pmatrix} 1-p_B & p_B & 0 \\ 1-p_{NB} & 0 & p_{NB} \\ 1 & 0 & 0 \end{pmatrix} \end{array}$$

5.7.3 Our model $(\phi_{NB}, \phi_B, \psi_{NBB}, \psi_{BNB}, p_{NB}, p_B)$

```
multistate <- nimbleCode({
  # ------------------------------------------------
  # Parameters:
  # phiB: survival probability state B
  # phiNB: survival probability state NB
  # psiBNB: transition probability from B to NB
  # psiNBB: transition probability from NB to B
  # pB: recapture probability B
  # pNB: recapture probability NB
  # ------------------------------------------------
  # States (z):
  # 1 alive B
  # 2 alive NB
  # 3 dead
  # Observations (y):
```

```
  # 1 not seen
  # 2 seen as B
  # 3 seen as NB
  # ------------------------------------------------
...
```

5.7.4 Our model $(\phi_{NB}, \phi_B, \psi_{NBB}, \psi_{BNB}, p_{NB}, p_B)$

```
multistate <- nimbleCode({
...
  # Priors
  phiB ~ dunif(0, 1)
  phiNB ~ dunif(0, 1)
  psiBNB ~ dunif(0, 1)
  psiNBB ~ dunif(0, 1)
  pB ~ dunif(0, 1)
  pNB ~ dunif(0, 1)
...
```

5.7.5 Our model $(\phi_{NB}, \phi_B, \psi_{NBB}, \psi_{BNB}, p_{NB}, p_B)$

```
multistate <- nimbleCode({
...
  # probabilities of state z(t+1) given z(t)
  gamma[1,1] <- phiB * (1 - psiBNB)
  gamma[1,2] <- phiB * psiBNB
  gamma[1,3] <- 1 - phiB
  gamma[2,1] <- phiNB * psiNBB
  gamma[2,2] <- phiNB * (1 - psiNBB)
  gamma[2,3] <- 1 - phiNB
  gamma[3,1] <- 0
  gamma[3,2] <- 0
```

```
  gamma[3,3] <- 1
...
```

5.7.6 Our model $(\phi_{NB}, \phi_B, \psi_{NBB}, \psi_{BNB}, p_{NB}, p_B)$

```
multistate <- nimbleCode({
...
  # probabilities of y(t) given z(t)
  omega[1,1] <- 1 - pB    # Pr(alive B t -> non-detected t)
  omega[1,2] <- pB        # Pr(alive B t -> detected B t)
  omega[1,3] <- 0         # Pr(alive B t -> detected NB t)
  omega[2,1] <- 1 - pNB   # Pr(alive NB t -> non-detected t)
  omega[2,2] <- 0         # Pr(alive NB t -> detected B t)
  omega[2,3] <- pNB       # Pr(alive NB t -> detected NB t)
  omega[3,1] <- 1         # Pr(dead t -> non-detected t)
  omega[3,2] <- 0         # Pr(dead t -> detected N t)
  omega[3,3] <- 0         # Pr(dead t -> detected NB t)
...
```

5.7.7 Our model $(\phi_{NB}, \phi_B, \psi_{NBB}, \psi_{BNB}, p_{NB}, p_B)$

```
multistate <- nimbleCode({
...
  # likelihood
  for (i in 1:N){
    # latent state at first capture
    z[i,first[i]] <- y[i,first[i]] - 1
    for (t in (first[i]+1):K){
      # z(t) given z(t-1)
      z[i,t] ~ dcat(gamma[z[i,t-1],1:3])
      # y(t) given z(t)
      y[i,t] ~ dcat(omega[z[i,t],1:3])
    }
```

```
  }
})
```

```
##         mean   sd 2.5%  50% 97.5% Rhat n.eff
## pB      0.60 0.03 0.54 0.59  0.66 1.00   202
## pNB     0.57 0.03 0.51 0.57  0.62 1.01   281
## phiB    0.80 0.02 0.77 0.80  0.83 1.01   313
## phiNB   0.85 0.02 0.82 0.85  0.88 1.00   404
## psiBNB 0.25 0.02 0.21 0.25  0.30 1.00   434
## psiNBB 0.24 0.02 0.20 0.24  0.29 1.03   478
```

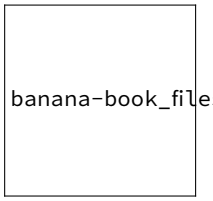


FIGURE 5.2: Dirichlet prior with parameter alpha

5.8 Multistate models are very flexible

Access to reproduction

Temporary emigration

Combination of life and dead encounters

5.8.1 Access to reproduction

Transition matrix:

$$\mathbf{\Gamma} = \begin{array}{c} \begin{array}{ccccc} z_t = J & z_t = 1yNB & z_t = 2yNB & z_t = B & z_t = D \end{array} \\ \begin{pmatrix} 0 & \phi_1(1-\alpha_1) & 0 & \phi_1\alpha_1 & 1-\phi_1 \\ 0 & 0 & \phi_2(1-\alpha_2) & \phi_2\alpha_2 & 1-\phi_2 \\ 0 & 0 & 0 & \phi_3 & 1-\phi_3 \\ 0 & 0 & 0 & \phi_B & 1-\phi_B \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{c} \\ z_{t-1} = J \\ z_{t-1} = 1yNB \\ z_{t-1} = 2yNB \\ z_{t-1} = B \\ z_{t-1} = D \end{array}$$

First-year and second-year individuals breed with probabilities α_1 and α_2.

Then, everybody breeds from age 3.

5.8.2 Access to reproduction

Observation matrix:

$$\mathbf{\Omega} = \begin{array}{c} \begin{array}{cccc} y_t = 0 & y_t = 1 & y_t = 2 & y_t = 3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1-p_1 & p_1 & 0 & 0 \\ 1-p_2 & 0 & p_2 & 0 \\ 1-p_3 & 0 & 0 & p_3 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{c} \\ z_t = J \\ z_t = 1yNB \\ z_t = 2yNB \\ z_t = B \\ z_t = D \end{array}$$

Juveniles are never detected.

5.9 Temporary emigration

Transition matrix:

$$\mathbf{\Gamma} = \begin{array}{c} \begin{array}{ccc} z_t = \text{in} & z_t = \text{out} & z_t = \text{D} \end{array} \\ \begin{pmatrix} \phi(1-\psi_{\text{in}\to\text{out}}) & \phi\psi_{\text{in}\to\text{out}} & 1-\phi \\ \phi\psi_{\text{out}\to\text{in}} & \phi(1-\psi_{\text{out}\to\text{in}}) & 1-\phi \\ 0 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{c} \\ z_{t-1} = \text{in} \\ z_{t-1} = \text{out} \\ z_{t-1} = \text{D} \end{array}$$

Observation matrix:

$$\mathbf{\Omega} = \begin{array}{c} \begin{matrix} y_t = 0 & y_t = 1 \end{matrix} \\ \begin{pmatrix} 1-p & p \\ 1 & 0 \\ 1 & 0 \end{pmatrix} \end{array} \begin{array}{l} z_t = \text{in} \\ z_t = \text{out} \\ z_t = \text{D} \end{array}$$

5.9.1 Combination of life and dead encounters

Transition matrix

$$\mathbf{\Gamma} = \begin{array}{c} \begin{matrix} z_t = A & z_t = JD & z_t = D \end{matrix} \\ \begin{pmatrix} s & 1-s & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{c} z_{t-1} = \text{alive} \\ z_{t-1} = \text{just dead} \\ z_{t-1} = \text{dead for good} \end{array}$$

Observation matrix

$$\mathbf{\Omega} = \begin{array}{c} \begin{matrix} y_t = 0 & y_t = 1 & y_t = 2 \end{matrix} \\ \begin{pmatrix} 1-p & 0 & p \\ 1-r & r & 0 \\ 1 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{l} z_t = A \\ z_t = JD \\ z_t = D \end{array}$$

5.10 Issue of local minima

Simulated data: 2 sites or states, and 7 occasions, Survival $\phi = 1$, detection $p = 0.6$, Transition $\psi_{12} = 0.6$, Transition $\psi_{21} = 0.85$.

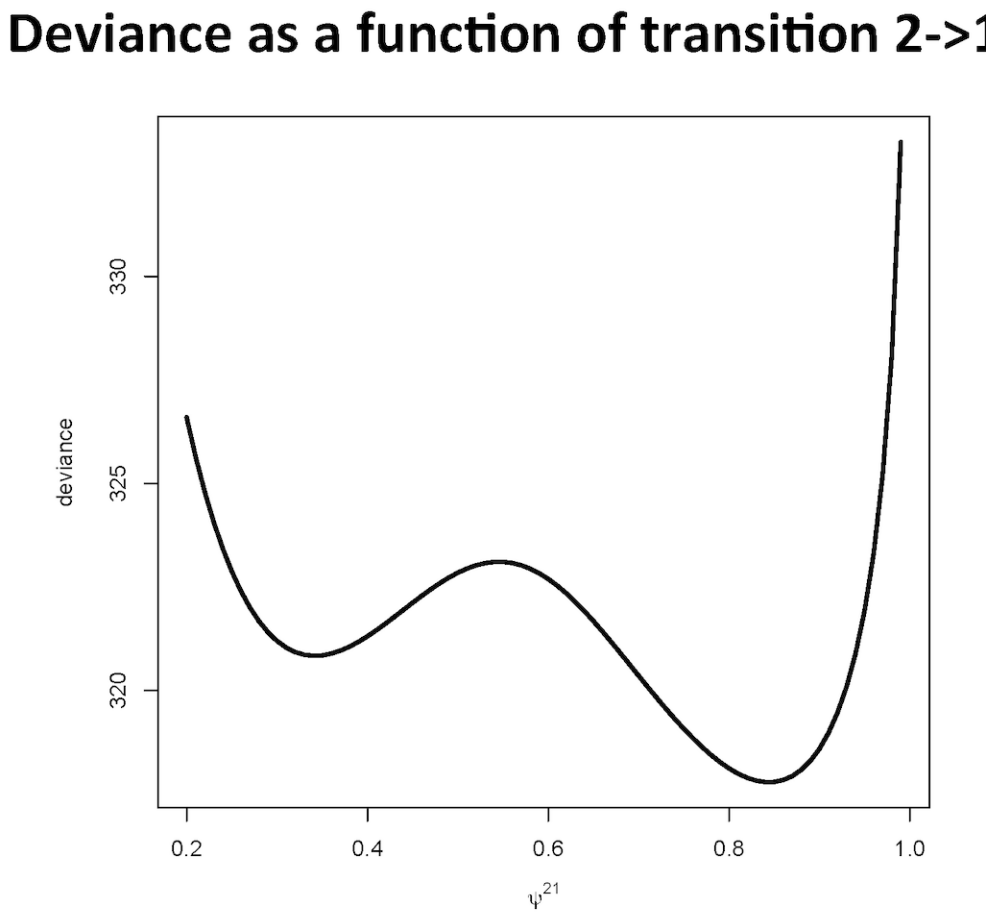
Courtesy of Jérôme Dupuis, used in Gimenez et al. (2005)[3].

[3] https://oliviergimenez.github.io/pubs/Gimenezetal2005JABES.pdf

5.10.1 Data

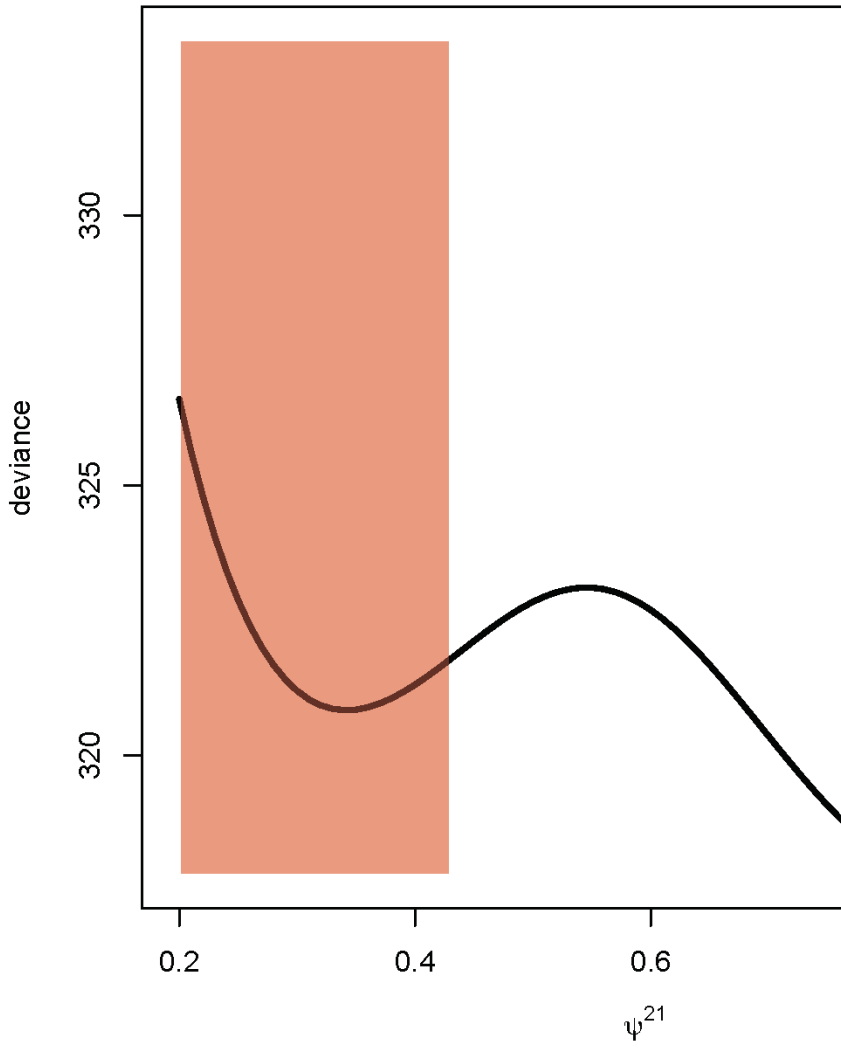
V1	V2	V3	V4	V5	V6	V7
2	0	2	1	2	0	2
2	0	2	1	2	0	2
2	0	2	1	2	0	2
2	0	2	1	2	0	2
1	1	1	0	1	0	1
1	1	1	0	1	0	1
1	1	1	0	1	0	1
1	1	1	0	1	0	1
2	0	2	0	2	0	1
2	0	2	0	2	0	1
2	0	2	0	2	0	1
2	0	2	0	2	0	1
1	0	1	0	1	0	1
1	0	1	0	1	0	1
1	0	1	0	1	0	1
1	0	1	0	1	0	1
2	0	2	0	2	0	2
2	0	2	0	2	0	2
2	0	2	0	2	0	2
2	0	2	0	2	0	2
1	0	1	0	1	0	2
1	0	1	0	1	0	2
1	0	1	0	1	0	2
1	0	1	0	1	0	2
2	2	0	1	0	2	1
2	2	0	1	0	2	1
2	2	0	1	0	2	1
2	2	0	1	0	2	1
2	1	0	2	0	1	1
2	1	0	2	0	1	1
2	1	0	2	0	1	1
2	1	0	2	0	1	1

```
knitr::include_graphics("images/multistate_local_minimav2_Page_05.png")
```



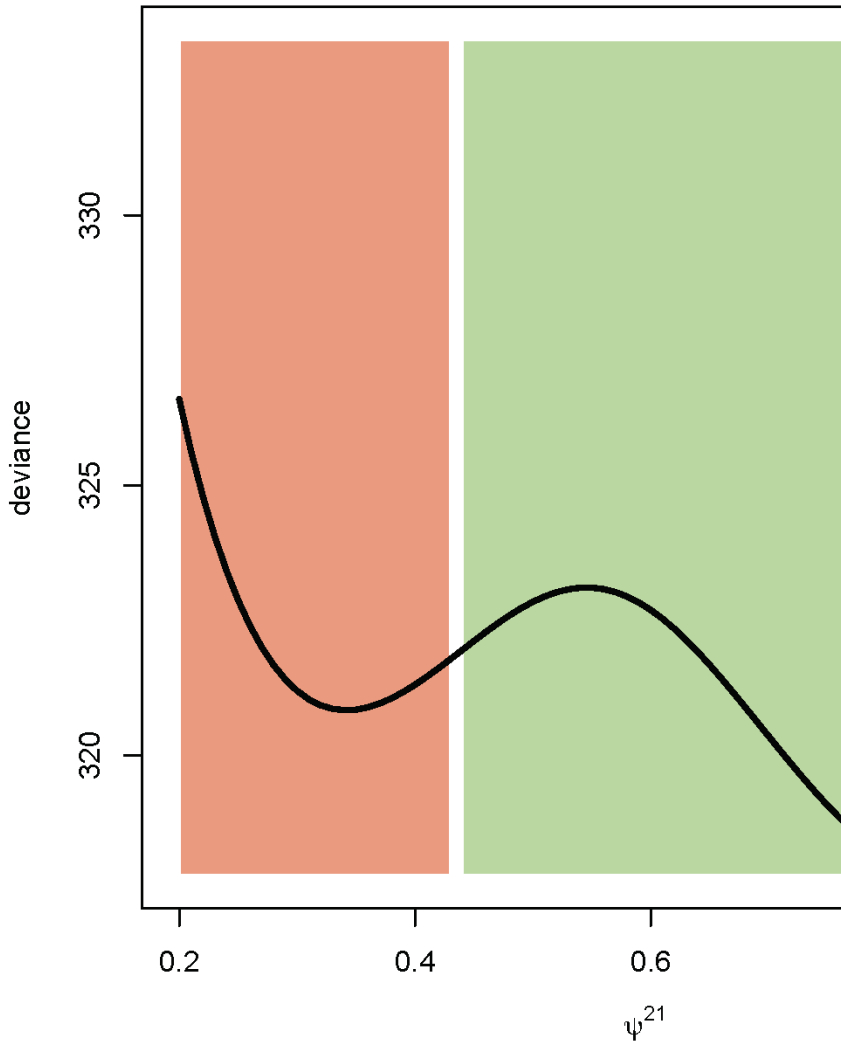

```
knitr::include_graphics("images/multistate_local_minimav2_Page_06.png")
```

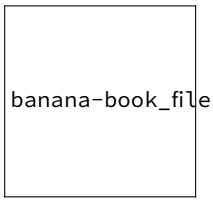
Initial values lead to *local*

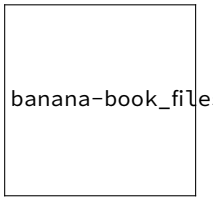


```
knitr::include_graphics("images/multistate_local_minimav2_Page_07.png")
```

Initial values lead to *globa*







5.11 Summary

- Blabla.
- Blabla.

5.12 Suggested reading

- Lebreton, J.-D., J. D. Nichols, R. J. Barker, R. Pradel and J. A. Spendelow (2009). Modeling Individual Animal Histories with Multistate Capture–Recapture Models[4]. Advances in Ecological Research, 41:87-173.

[4] https://multievent.sciencesconf.org/conference/multievent/pages/Lebretonetal2009AER.pdf

6

Covariates

7

Model selection and validation

Part III

III. States

Introduction

8

State uncertainty

9

Hidden semi-Markov models

Part IV

IV. Case studies

Introduction

10

Life history theory

10.1 Tradeoffs

?, **?**, and **?**

10.2 Breeding dynamics

?, **?**, **?**, and **?**

10.3 Actuarial senescence

?, **?**

10.4 Cause-specific mortalities

? and **?**

10.5 Disease dynamics

? and **?**

10.6 Sex uncertainty

? and **?**

11

Abundance

11.1 Horvitz-Thompson

?

Infering the latent states z can be useful to estimate prevalence, e.g. in animal epidemiology with prevalence of a disease[1], in evolutionary ecology with sex ratio[2] or in conservation biology with prevalence of hybrids[3].

11.2 Jolly-Seber

11.3 Robust design

?, **?**, **?**, and **?**

[1] https://veterinaryresearch.biomedcentral.com/articles/10.1186/1297-9716-45-39

[2] https://onlinelibrary.wiley.com/doi/abs/10.1002/cjs.5550360105

[3] https://onlinelibrary.wiley.com/doi/full/10.1002/ece3.4819?af=R

12

Stopover duration

?

13

Individual dependence

13.1 Dependence among individuals

? and **?**

13.2 Individual heterogeneity

?, **?**, and **?**

Part V

V. Conclusion

Take-home messages

–> –>

–>

–>

–>

FAQ