

Olivier Gimenez

Bayesian analysis of capture-recapture data with hidden Markov models

Theory and case studies in R and NIMBLE



Contents

List of Figures	xi
List of Tables	xv
Welcome	xvii
Preface	xix
I Foundations	1
Introduction	3
1 Bayesian statistics & MCMC	5
1.1 Introduction	5
1.2 Bayes' theorem	5
1.3 What is the Bayesian approach?	7
1.4 Approximating posteriors via numerical integration	9
1.5 Markov chain Monte Carlo (MCMC)	15
1.5.1 Monte Carlo integration	16
1.5.2 Markov chains	17
1.5.3 Metropolis algorithm	18
1.6 Assessing convergence	26
1.6.1 Burn-in	26

1.6.2	Chain length	28
1.6.3	What if you have issues of convergence?	32
1.7	Summary	33
1.8	Suggested reading	33
2	NIMBLE tutorial	35
2.1	Introduction	35
2.2	What is NIMBLE?	35
2.3	Getting started	37
2.4	Programming	51
2.4.1	NIMBLE functions	51
2.4.2	Calling R/C++ functions	54
2.4.3	User-defined distributions	55
2.5	Under the hood	58
2.6	MCMC samplers	66
2.6.1	Default samplers	66
2.6.2	User-defined samplers	67
2.7	Tips and tricks	73
2.7.1	Precision vs standard deviation	73
2.7.2	Indexing	74
2.7.3	Faster compilation	74
2.7.4	Updating MCMC chains	75
2.7.5	Reproducibility	76
2.7.6	Parallelization	77
2.7.7	Incomplete and incorrect initialization	80
2.7.8	Vectorization	82
2.8	Summary	83
2.9	Suggested reading	83

3 Hidden Markov models	85
3.1 Introduction	85
3.2 Longitudinal data	85
3.3 A Markov model for longitudinal data	87
3.3.1 Assumptions	87
3.3.2 Transition matrix	88
3.3.3 Initial states	88
3.3.4 Likelihood	89
3.3.5 Example	91
3.4 Bayesian formulation	91
3.5 NIMBLE implementation	94
3.6 Hidden Markov models	99
3.6.1 Capture-recapture data	99
3.6.2 Observation matrix	101
3.6.3 Hidden Markov model	102
3.6.4 Likelihood	103
3.7 Fitting HMM with NIMBLE	104
3.8 Marginalization	110
3.8.1 Brute-force approach	110
3.8.2 Forward algorithm	113
3.8.3 NIMBLE implementation	116
3.9 Pooled encounter histories	122
3.10 Decoding after marginalization	127
3.10.1 Theory	128
3.10.2 Implementation	130
3.10.3 Compute first, average after	134
3.10.4 Average first, compute after	135

3.11	Summary	137
3.12	Suggested reading	137
II	Transitions	139
	Introduction	141
4	Alive and dead	143
4.1	Introduction	143
4.2	The Cormack-Jolly-Seber (CJS) model	143
4.3	Capture-recapture data	144
4.4	Fitting the CJS model to the dipper data with NIMBLE	147
4.5	CJS model derivatives	152
4.6	Model comparison with WAIC	159
4.7	Goodness of fit	161
4.7.1	Posterior predictive checks	161
4.7.2	Classical tests	161
4.7.3	Design considerations	164
4.8	Covariates	164
4.8.1	Temporal covariates	165
4.8.2	Individual covariates	179
4.8.3	Several covariates	184
4.8.4	Random effects	190
4.8.5	Individual time-varying covariates	194
4.9	Why Bayes? Incorporate prior information	200
4.9.1	Prior elicitation	200
4.9.2	Moment matching	201
4.10	Summary	203
4.11	Suggested reading	203

5 Sites and states	205
5.1 Introduction	205
5.2 The Arnason-Schwarz (AS) model	205
5.2.1 Theory	205
5.3 Fitting the AS model	208
5.3.1 Geese data	208
5.3.2 NIMBLE implementation	210
5.3.3 Goodness of fit	220
5.4 What if more than 2 sites?	221
5.4.1 Dirichlet prior	221
5.4.2 Multinomial logit	228
5.5 Sites may be states	231
5.5.1 Titis data	232
5.5.2 The AS model for states	234
5.5.3 NIMBLE implementation	235
5.6 Issue of local minima	240
5.7 Uncertainty	245
5.7.1 Breeding states	245
5.7.2 Disease states	252
5.8 Summary	256
5.9 Suggested reading	257
III Case studies	259
Introduction	261

6 Covariates	263
6.1 Covariate selection with reversible jump MCMC	263
6.2 Missing values	264
6.3 Sex uncertainty	264
6.4 Nonlinearities	264
6.5 Spatial	264
7 Lack of fit	267
7.1 Individual heterogeneity	267
7.2 Trap dep	273
7.3 Transience	273
7.4 Temporary emigration	273
7.5 Memory model	274
7.6 Posterior predictive check	276
8 Life history	277
8.1 Access to reproduction	277
8.2 Tradeoffs	278
8.3 Breeding dynamics	278
8.4 Using data on dead recoveries	278
8.4.1 Ring recovery simple model	278
8.4.2 Combination of live captures and dead recoveries	278
8.4.3 Cause-specific mortalities	279
8.5 Stopover duration	279
8.6 Actuarial senescence	279
8.7 Uncertainty in age	280
8.8 Uncertainty in age and size	280

<i>Contents</i>	ix
Conclusion	281



List of Figures

1.1	Cartoon of Thomas Bayes with Bayes' theorem in background. Source: James Kulich at < https://www.elmhurst.edu/blog/thomas-bayes/ >	6
1.2	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 around the mean (the smaller the variance). The expectation (or mean) of a beta(a,b) is $\frac{a}{a+b}$	13
1.3	MCMC article cover. Source: The Journal of Chemical Physics – https://aip.scitation.org/doi/10.1063/1.1699114	15
2.1	Logo of the NIMBLE R package designed by Luke Larson.	36
3.1	Graphical representation of the Viterbi algorithm with $\phi = 0.8$ and $p = 0.6$. States are alive $z = 1$ or dead $z = 2$ and observations are non-detected $y = 1$ or detected $y = 2$. To be done properly w/ tikz.	130
4.1	Animal individual marking. Top-left: rings (credit: Emmanuelle Cam and Jean-Yves Monat); Top-right: ear-tags (credit: Kelly Powell); Bottom left: coat patterns (credit: Fridolin Zimmermann); Bottom right: ADN feces (credit: Alexander Kopatz)	145

4.2	White-throated Dipper (<i>Cinclus cinclus</i>). Credit: Gilbert Marzolin.	146
5.1	Canada goose (<i>Branta canadensis</i>). Credit: Max McCarthy.	209
5.2	The Dirichlet distribution as a prior for $(\psi^{11}, \psi^{12}, \psi^{13})$ with vector of parameters α . Here all components of α are equal which makes the distribution symmetric, and its mean is $(1/3, 1/3, 1/3)$. When $\alpha = 1$, the prior for the ψ 's is uniform (middle panel), unimodal when $\alpha = 10$ (right panel) and concentrated in the corners (0 and 1) when $\alpha = 0.1$ (left panel).	223
5.3	The distribution $\text{gamma}(\alpha, \theta)$ for different values of α and θ . The shape argument α determines the overall shape while the scale parameter θ only affects the scale values (compare the values on X- and Y-axes between the bottom and upper panels). The exponential and chi-square distributions are particular cases of the gamma distribution. If the parameter shape is close to zero, the gamma is very similar to the exponential (bottom and upper left panels). If the parameter shape is large, then the gamma is similar to the chi-squared distribution (bottom and upper right panels).	227
5.4	Sooty shearwater (* <i>Ardenna grisea</i> *). Credit: John Harrison.	232
5.5	Influence of the choice of initial values on the convergence to the global minimum of the deviance illustrated with simulated data. The black curve is the profile deviance of the probability to move from site 2 to 1. If an initial value is picked in the red area, we end up in the local minimum while if it is picked in the green area, then we get the global minimum which corresponds to the maximum likelihood estimate.	242
5.6	A house finch with a heavy infection caused by conjunctivitis. Credit: Jim Mondok.	253

<i>List of Figures</i>	xiii
7.1 Dominance in wolves.	268



List of Tables



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 and NIMBLE*. Here, I write about three of my favorite research topics – capture-recapture, hidden Markov models and Bayesian statistics – let's enjoy this great cocktail together ×

I'm currently writing this book, and I welcome any feedback. You may raise an issue here¹, 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 me². Many thanks!

Olivier Gimenez. Written in Montpellier, France and Athens, Greece.
Last updated: June 23, 2025

License

The online version of this book is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License³.

The code is public domain, licensed under Creative Commons CC0 1.0 Universal (CC0 1.0)⁴.

¹<https://github.com/oliviergimenez/banana-book/issues>

²<mailto:olivier.gimenez@cefe.cnrs.fr>

³<http://creativecommons.org/licenses/by-nc-nd/4.0/>

⁴<https://creativecommons.org/publicdomain/zero/1.0/>



Preface

Why this book?

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. HMMs are increasingly used to analyse capture-recapture data and estimate key population parameters (e.g., survival, dispersal, or recruitment) with applications in all fields of ecology. I have assembled a searchable list at <https://oliviergimenez.github.io/curated-list-HMM-apps/> of HMM analyses of capture-recapture data to get inspiration. This list is not exhaustive, please get in touch with us if you'd like to add a reference. The first objective of this book is to illustrate the flexibility of HMM to decompose complex problems in smaller pieces that are easier to understand, model and analyse.

In parallel, Bayesian statistics is well established and fast growing in ecology and related disciplines, because it resonates with the 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) that allow practitioners to code their own analyses. The second objective of this book is to illustrate the use of the R package NIMBLE ([de Valpine et al. \[2017\]](#)) to analyse capture-recapture data with HMM in a Bayesian framework. NIMBLE 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 from published papers presented in a tutorial style to abide by the “learning by doing” philosophy. The third objective of this book is to provide reproducible analyses with code and data to teach yourself by example.

Who should read this book?

This book is aimed at beginners who're comfortable using R and write basic code, 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.

What will you learn?

The book is divided into three parts. The first part `Foundations` is aimed at getting you up-to-speed with Bayesian statistics, NIMBLE, and hidden Markov models. The second part `Transitions` will teach you all about capture-recapture models for open populations, with reproducible R code to ease the learning process. The third part `Case studies` 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. **Here say that the code is available on GitHub, and the data through a package `crdata`.**

What won't you learn?

I won't lie, there is some maths in this book. However, 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 exhaustively, 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 sug-

gested 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 [Williams et al., 2002], spatial capture-recapture models [Royle et al., 2013] and continuous models [Rushing, 2023] (**I might end up writing a chapter on continuous models.**). These models can be treated as HMMs, but for now the usual formulation is just fine. These developments will be the subject of new chapters in a second edition, hopefully.

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(  
  "bookdown", "coda", "forecast", "ggtern", "gtools",  
  "here", "janitor", "magick", "MCMCvis", "nimble",  
  "nimbleEcology", "patchwork", "pdftools",  
  "RColorBrewer", "sessioninfo", "tidyverse",  
  "wesanderson"  
)
```

How this book was written

I wrote this book in RStudio <http://www.rstudio.com/ide/> using bookdown <http://bookdown.org/>. The book website <https://oliviergimenez.github.io/banana-book> is hosted with GitHub Pages <https://pages.github.com/>, and automatically updated after every push by Github

Actions <https://github.com/features/actions>. The source is available from GitHub <https://github.com/oliviergimenez/banana-book>.

The version of the book you're reading was built with R version 4.5.0 (2025-04-11) and the following packages:

package	version	source
bookdown	0.43	CRAN (R 4.5.0)
coda	0.19-4.1	CRAN (R 4.5.0)
forecast	8.24.0	CRAN (R 4.5.0)
ggtern	3.5.0	CRAN (R 4.5.0)
gtools	3.9.5	CRAN (R 4.5.0)
here	1.0.1	CRAN (R 4.5.0)
janitor	NA	NA
magick	2.8.7	CRAN (R 4.5.0)
MCMCvis	0.16.3	CRAN (R 4.5.0)
nimble	1.3.0	CRAN (R 4.5.0)
nimbleEcology	0.5.0	CRAN (R 4.5.0)
patchwork	1.3.0	CRAN (R 4.5.0)
pdftools	3.5.0	CRAN (R 4.5.0)
RColorBrewer	1.1-3	CRAN (R 4.5.0)
sessioninfo	1.2.3	CRAN (R 4.5.0)
tidyverse	2.0.0	CRAN (R 4.5.0)
wesanderson	0.3.7	CRAN (R 4.5.0)

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; <https://www.cnrs.fr/en>) 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 carnivores. I would be nothing without the students and colleagues who are kind enough to bear with me.

You may find me on Twitter/X (<https://twitter.com/oaggimenez>), GitHub (<https://github.com/oliviergimenez>), or get in touch by email at olivier|dot|gimenez|at|cefe|dot|cnrs|dot|fr.

Acknowledgements

Writing a book is quite an adventure, and a lot of people contributed to make this book a reality. I wish to thank:

- Rob Calver, Sherry Thomas, Vaishali Singh and Kumar Shashi at Chapman and Hall/CRC.
- Marc Kéry, Rachel McCrea, Byron Morgan and Etienne Prévost for their positive reviews of the book proposal I sent to Chapman and Hall/CRC, and their constructive comments and suggestions.
- Marc Kéry for his precious pieces of advice on the process of writing.
- Perry de Valpine, Daniel Turek, Chris Paciorek and Ben Goldstein for the `NIMBLE` and `nimbleEcology` R packages.
- Colleagues who shared their data; See list at <https://github.com/oliviergimenez/banana-book#readme>

- People who commented, corrected, offered pieces of advice; See list at [`https://github.com/oliviergimenez/banana-book#readme`](https://github.com/oliviergimenez/banana-book#readme).
- Yihui Xie for the `bookdown` R package.
- Attendees of the workshops we run in relation to the content of this book (latest edition was in 2023, see [`https://oliviergimenez.github.io/bayesian-hmm-cr-workshop-valencia/`](https://oliviergimenez.github.io/bayesian-hmm-cr-workshop-valencia/))
- Perry de Valpine, Sarah Cubaynes, Chloé Nater, Maud Quéroué and Daniel Turek for their help with running workshops in relation to the content of this book (2021 edition: [`https://oliviergimenez.github.io/bayesian-cr-workshop/`](https://oliviergimenez.github.io/bayesian-cr-workshop/); 2022 edition: [`https://oliviergimenez.github.io/hmm-cr-nimble-isec2022-workshop/`](https://oliviergimenez.github.io/hmm-cr-nimble-isec2022-workshop/)).
- Ruth King, Steve Brooks and Byron Morgan for the workshop on Bayesian statistics for ecologists we taught in Cambridge, the book we wrote together [[King et al., 2009](#)], and their contribution to statistical ecology.
- Jean-Dominique Lebreton, Roger Pradel and Rémi Choquet for the workshops on modelling individual histories with state uncertainty we taught over the years, and sharing their science of capture-recapture with me.
- My family, including my mother, my parents-in-law for their kindness and hospitality, my amazing kids and wonderful wife for putting up with me while I was writing this book.

Part I

Foundations



Introduction

WORK IN PROGRESS

This first part `Foundations` is aimed at getting you up-to-speed with Bayesian statistics, NIMBLE, and hidden Markov models.



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 [[McGrayne, 2011](#)].

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 $\Pr(A | B)$, is the probability that A occurs, revised by considering the additional information that event B has occurred. 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

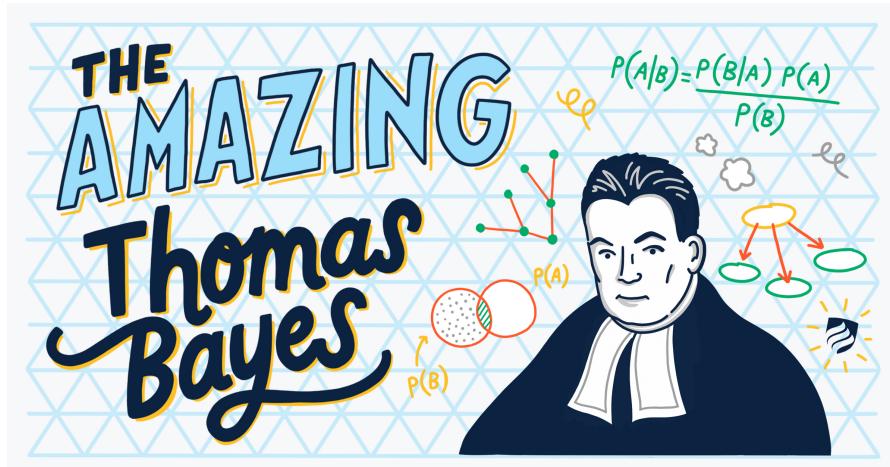


FIGURE 1.1: Cartoon of Thomas Bayes with Bayes' theorem in background. Source: James Kulich at <<https://www.elmhurst.edu/blog/thomas-bayes/>>

was even (event B) before you answer your friend's question. Because there are only three even numbers, one of which is six, you may revise your answer for the probability that a six was rolled from $1/6$ to $\Pr(A | B) = 1/3$. The order in which A and B appear is important, make sure you do not confuse $\Pr(A | B)$ and $\Pr(B | A)$.

Bayes' theorem gives you $\Pr(A | B)$ using marginal probabilities $\Pr(A)$ and $\Pr(B)$ and $\Pr(B | A)$:

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

Originally, Bayes' theorem was seen as a way to infer an unknown 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 unknown disease and B symptoms, the doctor knows $\Pr(\text{symptoms} | \text{disease})$ and wants to derive $\Pr(\text{disease} | \text{symptoms})$. This way of reversing $\Pr(B | A)$ into $\Pr(A | B)$ 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:

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

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 (see Chapter 4). 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? 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), 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. A probability distribution is a mathematical expression that gives the probability for a random variable to take particular values. It 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).

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 $B = \text{data}$, then Bayes' theorem gives you a way to estimate parameter θ given the data you have:

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

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 anything about θ . Usually however, you never start from scratch, and you'd like your prior to reflect the information you have (see Section 4.9 for how to accomplish that).

Last, we have **$\Pr(\text{data})$** which is sometimes called the average likeli-

hood because it is obtained by integrating the likelihood with respect to the prior $\text{Pr}(\text{data}) = \int L(\text{data} | \theta) \text{Pr}(\theta) d\theta$ 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 (we used a similar example in King et al. [2009]). We'd like to estimate winter survival θ . The data are:

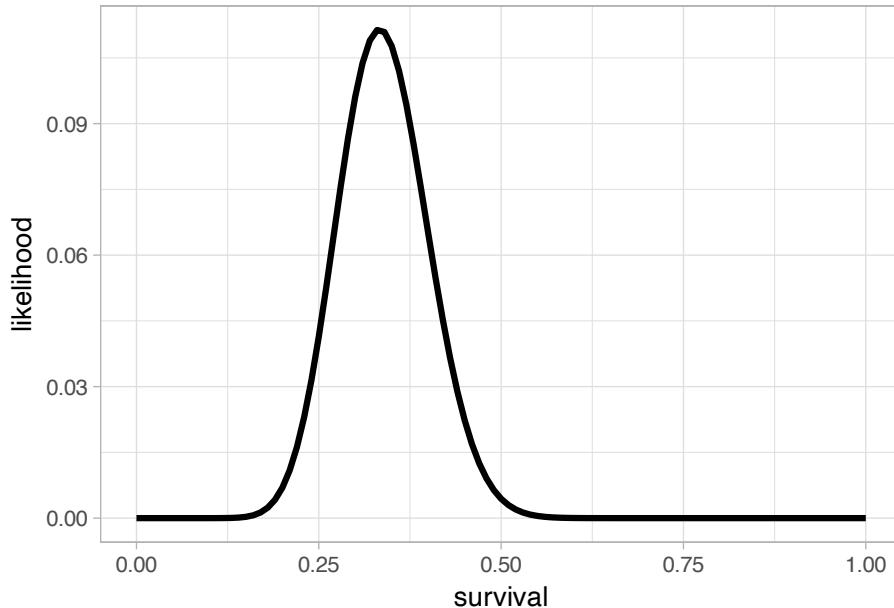
```
y <- 19 # nb of success
n <- 57 # nb of attempts
```

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 with n trials and θ the probability of success:

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

Note that I follow McElreath [2020] 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)
```



This is the 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.

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

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: $\Pr(\text{data} | \theta) \times \Pr(\theta)$

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

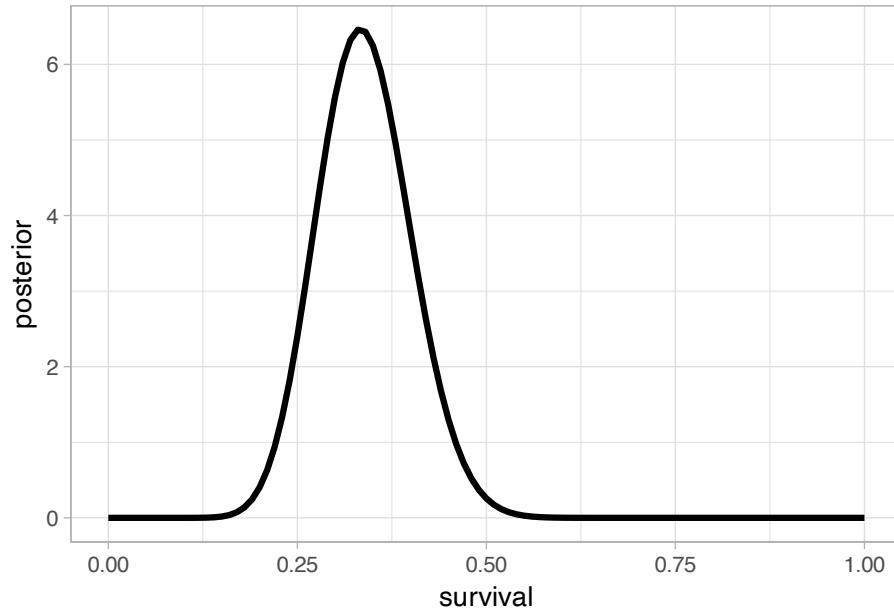
We write another function that calculates the denominator, the average likelihood: $\Pr(\text{data}) = \int L(\theta | \text{data}) \Pr(\theta) d\theta$

```
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 of winter survival 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)
```



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. We say that the beta distribution is the conjugate prior distribution for the binomial distribution. 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 1.2).

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 $b + n - y$. 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. Let's superimpose the exact posterior and the numerical approximation:

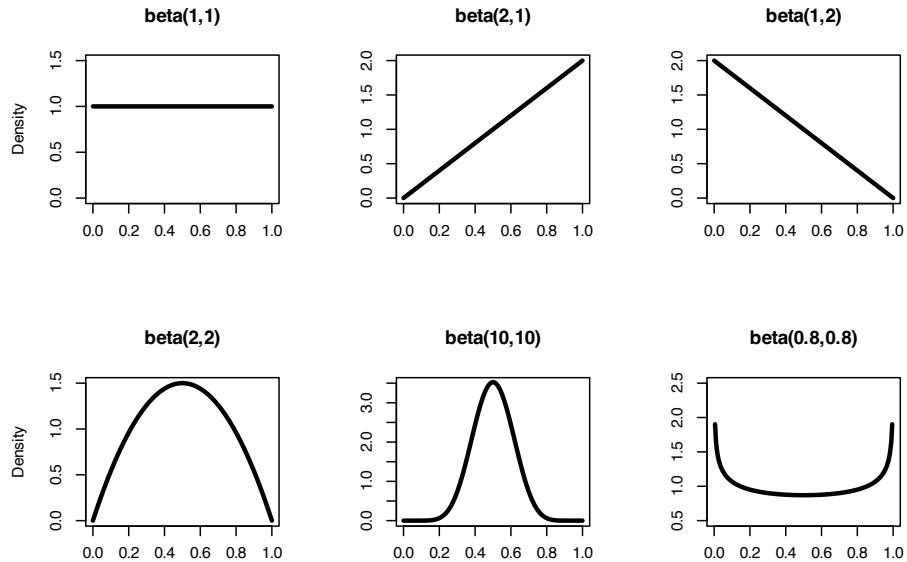
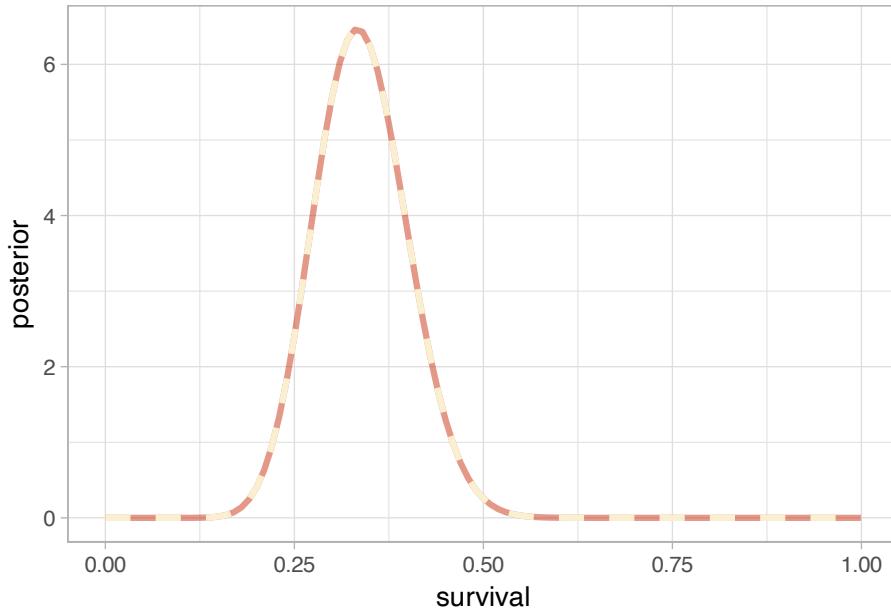


FIGURE 1.2: The distribution $\text{beta}(a,b)$ for different values of a and b . Note that for $a = b = 1$, we get the uniform distribution between 0 and 1 in the top left panel. When a and b are equal, the distribution is symmetric, and the bigger a and b , the more peaked the distribution around the mean (the smaller the variance). The expectation (or mean) of a $\text{beta}(a,b)$ is $\frac{a}{a+b}$.

```
explicit_posterior <- dbeta(grid, y + a, n - y + b)
dfexpposterior <- data.frame(survival = grid, explicit_posterior = explicit_posterior)
ggplot() +
  geom_line(data = numerical_posterior,
            aes(x = survival, y = posterior),
            size = 1.5,
            col = wesanderson::wes_palettes$Royal1[2],
            alpha = 0.5) +
  geom_line(data = dfexpposterior,
            aes(x = survival, y = explicit_posterior),
            size = 1.5,
            col = wesanderson::wes_palettes$Royal1[3],
            linetype = "dashed")
```



Clearly, the exact (dashed line) vs. numerical approximation (continuous line) of winter survival posterior distribution are indistinguishable, suggesting that the numerical approximation is more than fine.

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

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

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 1.3), 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.

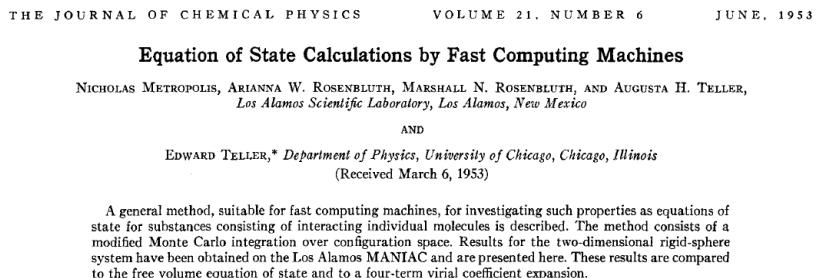


FIGURE 1.3: MCMC article cover. Source: The Journal of Chemical Physics – <https://aip.scitation.org/doi/10.1063/1.1699114>

These simulation algorithms are called Markov chain Monte Carlo (MCMC), and they definitely gave a boost to Bayesian statistics. There 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, \dots, 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(20, 39)
mean(sample_from_posterior) # compute mean with Monte Carlo integration
## [1] 0.3383
```

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

```
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 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%
## 0.2266 0.4590
```

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 \times :

$$\times = \begin{pmatrix} & \text{sunny tomorrow} & \text{rainy tomorrow} \\ \text{sunny today} & 0.8 & 0.2 \\ \text{rainy today} & 0.9 & 0.1 \end{pmatrix}$$

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, 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
```

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. You might have heard about the Metropolis-Hastings or the Gibbs sampler. Have a look to <https://chi-feng.github.io/mcmc-demo/> for an interactive gallery of MCMC algorithms. Here I illustrate the Metropolis algorithm and how to implement it in practice.

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)
}
```

```
# 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, or a starting location.
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 $R = \frac{\text{Pr}(\text{candidate}|\text{data})}{\text{Pr}(\text{current}|\text{data})}$. This is where the magic of MCMC happens, in that $\text{Pr}(\text{data})$, 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 $\text{Pr}(\text{candidate}|\text{data})$ is higher than at the current location $\text{Pr}(\text{current}|\text{data})$, 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:

```
move <- function(x, away = 1){ # by default, standard deviation of the proposal distribution is 1
  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 current logit
  candidate <- exp(logit_candidate) / (1 + exp(logit_candidate)) # inverse logit transform
  return(candidate)
}
```

```

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

```

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.

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
  }
}

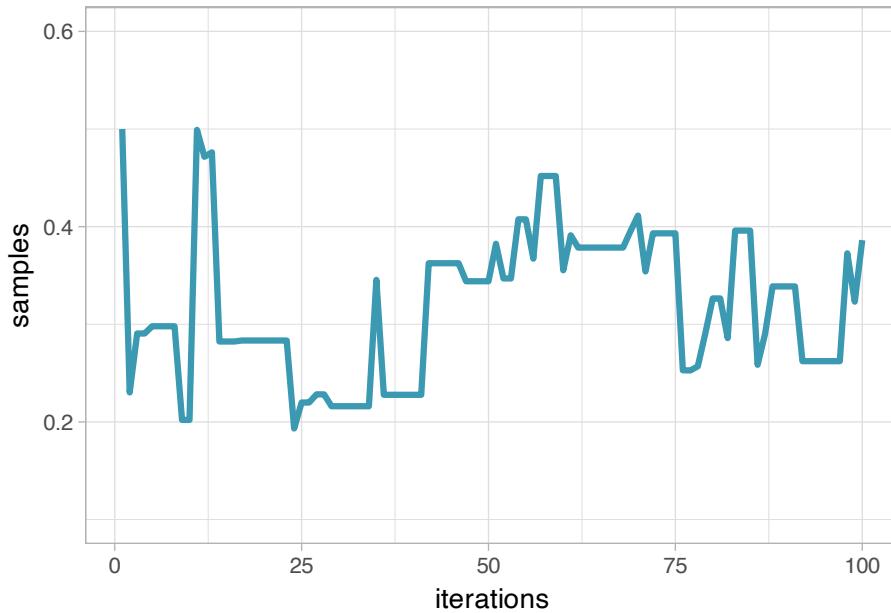
```

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:

```
df <- data.frame(x = 1:steps, y = theta.post)
df %>%
  ggplot() +
  geom_line(aes(x = x, y = y), size = 1.5, color = wesanderson::wes_palettes$Zissou1[1]) +
  labs(x = "iterations", y = "samples") +
  ylim(0.1, 0.6)
```



In this visualisation, remember that our Markov chain starts at value 0.5. The steps or iterations are 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.

To make our life easier and avoid repeating the same lines of code again and again, let's make a function out of the code we have written so far:

```
metropolis <- function(steps = 100, inits = 0.5, away = 1){

  # pre-alloc memory
  theta.post <- rep(NA, steps)

  # start
  theta.post[1] <- inits

  for (t in 2:steps){

    # propose candidate value for prob of success
    theta_star <- move(theta.post[t-1], away = away)

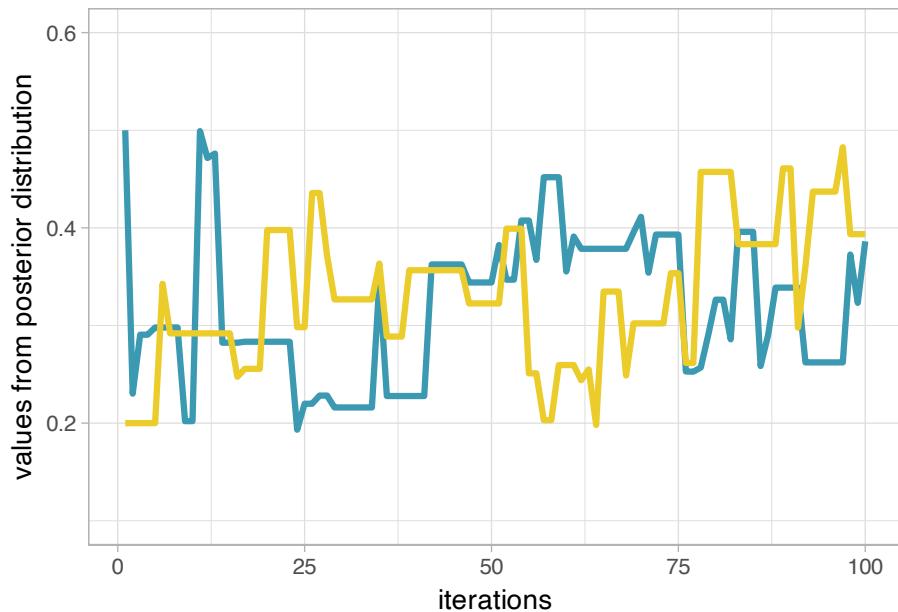
    # calculate ratio R
    pstar <- posterior(survived, p = theta_star)
    pprev <- posterior(survived, p = theta.post[t-1])
    logR <- pstar - pprev
    R <- exp(logR)

    # 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
    }
    else{
      theta.post[t] <- theta.post[t-1]
    }
  }
  theta.post
}
```

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 with trace plot of survival for two chains starting at 0.2 (yellow) and 0.5 (blue) run for 100 steps:

```
theta.post2 <- metropolis(steps = 100, inits = 0.2)
df2 <- data.frame(x = 1:steps, y = theta.post2)
ggplot() +
  geom_line(data = df, aes(x = x, y = y), size = 1.5, color = wesanderson::wes_palettes$Zissou)
  geom_line(data = df2, aes(x = x, y = y), size = 1.5, color = wesanderson::wes_palettes$Zissou)
  labs(x = "iterations", y = "values from posterior distribution") +
  ylim(0.1, 0.6)
```

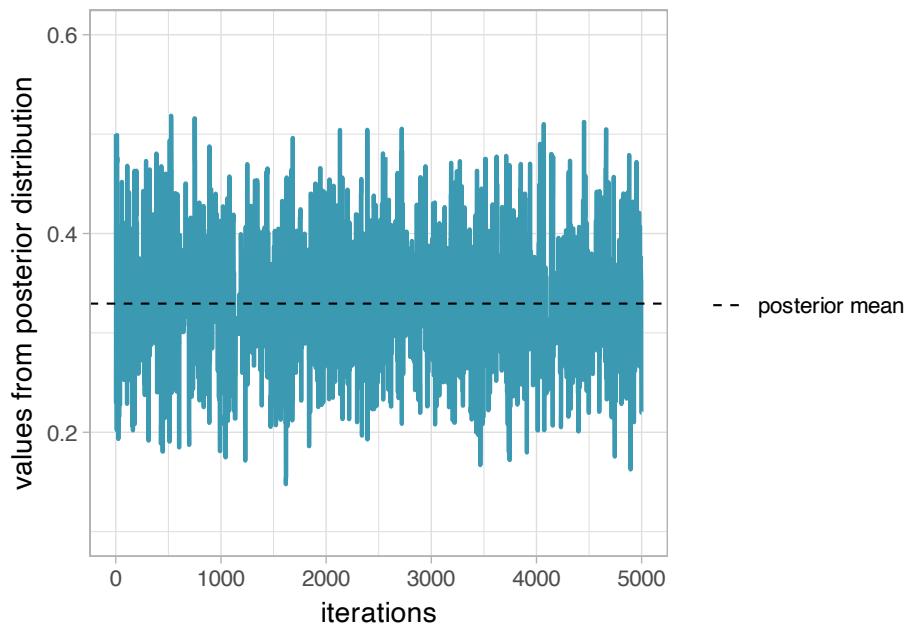


Notice that we do not get the exact same results because the algorithm is stochastic. The question is to know whether we have reached the stationary distribution. Let's increase the number of steps, start at 0.5 and run a chain with 5000 iterations:

```

steps <- 5000
set.seed(1234)
theta.post <- metropolis(steps = steps, inits = 0.5)
df <- data.frame(x = 1:steps, y = theta.post)
df %>%
  ggplot() +
  geom_line(aes(x = x, y = y), size = 1, color = wesanderson::wes_palettes$Zissou1[1]) +
  labs(x = "iterations", y = "values from posterior distribution") +
  ylim(0.1, 0.6) +
  geom_hline(aes(yintercept = mean(theta.post), linetype = "posterior mean")) +
  scale_linetype_manual(name = "", values = c(2,2))

```



This is what we're after, a trace plot that looks like a beautiful lawn, see Section 1.6.

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.

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).

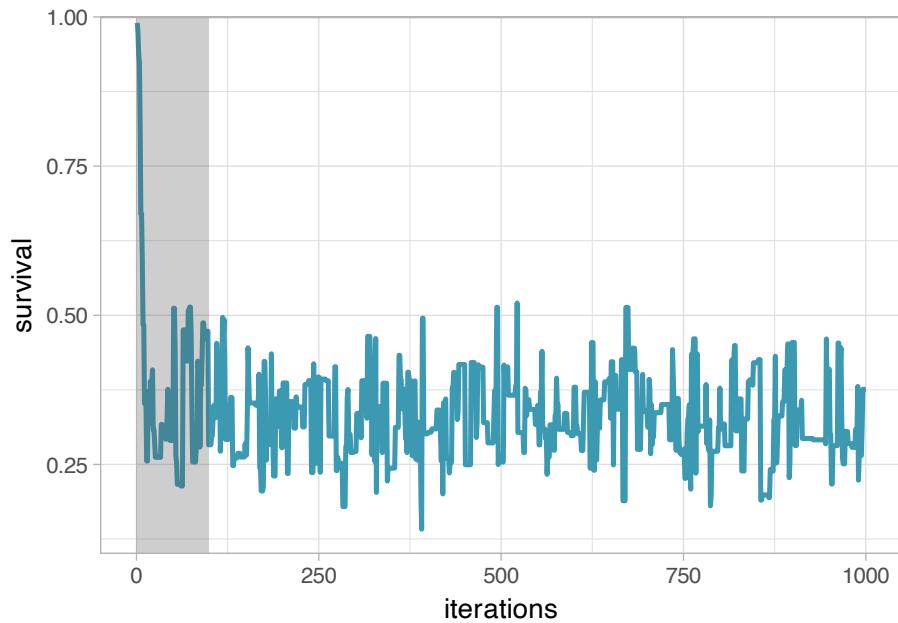
1.6.1 Burn-in

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, let's have a look to a trace plot of a chain that starts at value 0.99.

```
# set up the scene
steps <- 1000
theta.post <- metropolis(steps = steps, inits = 0.99)
df <- data.frame(x = 1:steps, y = theta.post)
df %>%
  ggplot() +
  geom_line(aes(x = x, y = y), size = 1.2, color = wesanderson::wes_palettes$Zissou1[1]) +
  labs(x = "iterations", y = "survival") +
  theme_light(base_size = 14) +
  annotate("rect",
           xmin = 0,
           xmax = 100,
           ymin = 0.1,
           ymax = 1,
```

```
alpha = .3) +
scale_y_continuous(expand = c(0,0))
```



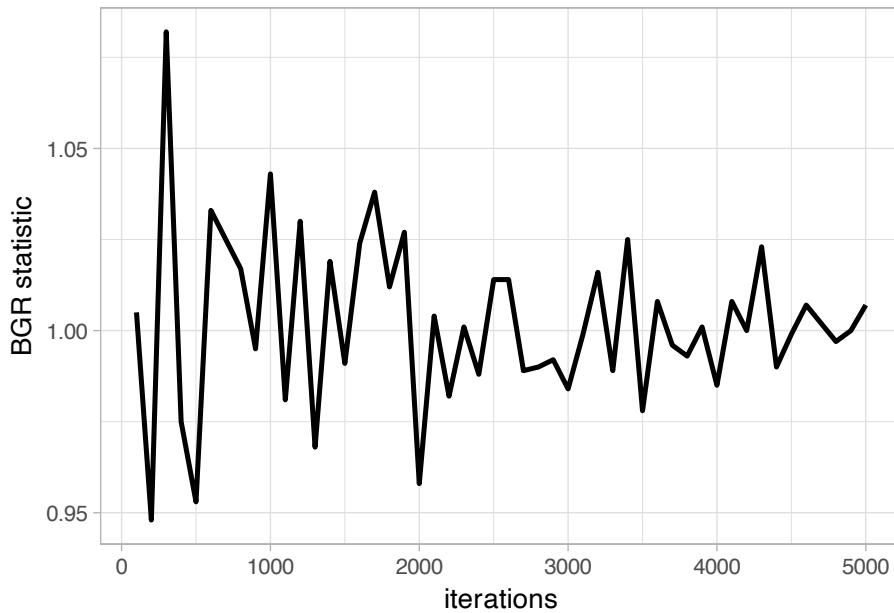
The chain starts at value 0.99 and rapidly stabilises, with values bouncing back and forth around 0.3 from the 100th iteration onwards. You may choose the shaded area as the burn-in, and discard the first 100th values.

We see from the trace plot below 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 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 (code not shown):



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.

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.

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. The figure below 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):

```
# inspired from https://bookdown.org/content/3686/markov-chain-monte-carlo.html

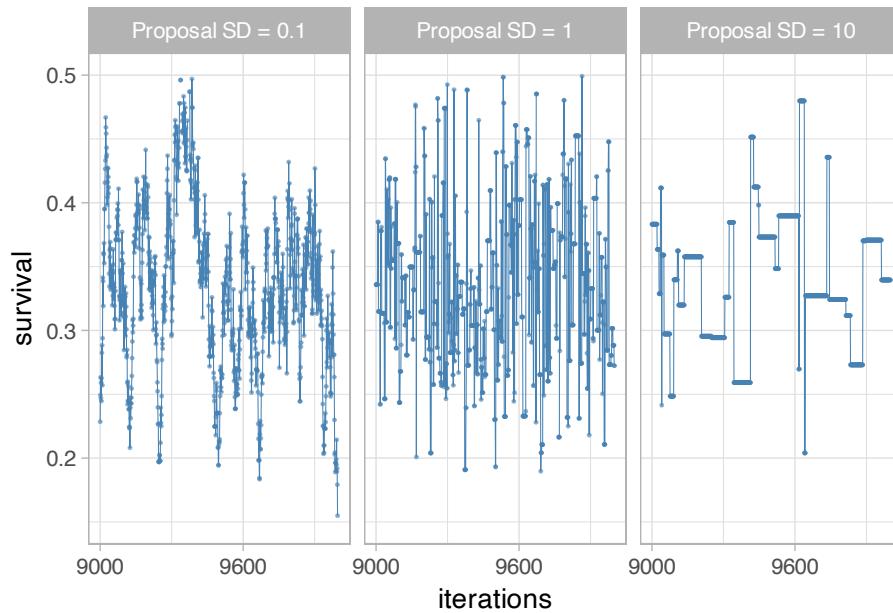
n_steps <- 10000

d <-
  tibble(away = c(0.1, 1, 10)) %>%
  mutate(accepted_traj = map(away, metropolis, steps = n_steps, inits = 0.1)) %>%
  unnest(accepted_traj)

d <-
  d %>%
  mutate(proposal_sd = str_c("Proposal SD = ", away),
         iter       = rep(1:n_steps, times = 3))

trace <- d %>%
  ggplot(aes(y = accepted_traj, x = iter)) +
  geom_path(size = 1/4, color = "steelblue") +
  geom_point(size = 1/2, alpha = 1/2, color = "steelblue") +
  scale_y_continuous("survival", breaks = 0:5 * 0.1, limits = c(0.15, 0.5)) +
  scale_x_continuous("iterations",
                     breaks = seq(n_steps - n_steps * 10/100, n_steps, by = 600),
                     limits = c(n_steps - n_steps * 10/100, n_steps)) +
  facet_wrap(~proposal_sd, ncol = 3) +
  theme_light(base_size = 14)

trace
```

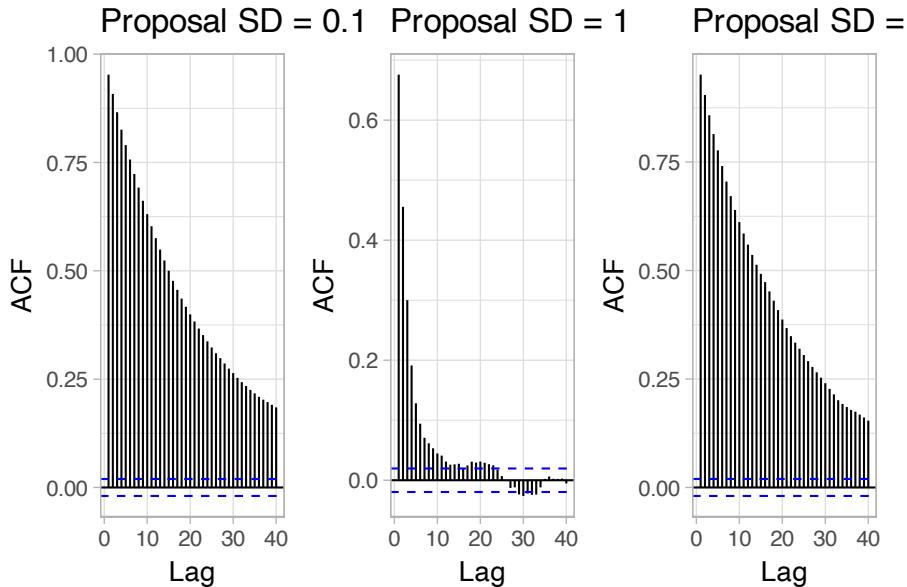


Small and big moves in the left and right panels provide high correlations between successive observations of the Markov chain, whereas a standard deviation of 1 in the center panel allows efficient exploration of the parameter space. The movement around the parameter space is referred to as *mixing*. Mixing is bad when the chain makes small and big moves, and good otherwise.

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*. We obtain the autocorrelation function plots for different values of the standard deviation of the proposal distribution with the R `forecast::ggAcf()` function:

```
library(forecast)
plot1 <- ggAcf(x = d$accepted_traj[d$proposal_sd=="Proposal SD = 0.1"]) + ggtitle("Proposal SD = 0.1")
plot2 <- ggAcf(x = d$accepted_traj[d$proposal_sd=="Proposal SD = 1"]) + ggtitle("Proposal SD = 1")
plot3 <- ggAcf(x = d$accepted_traj[d$proposal_sd=="Proposal SD = 10"]) + ggtitle("Proposal SD = 10")
```

```
library(patchwork)
(plot1 + plot2 + plot3)
```



In the left and right panels, autocorrelation is strong, decreases slowly with increasing lag and mixing is bad. In the center panel, autocorrelation is weak, decreases rapidly with increasing lag and mixing is good.

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. You should check the `n.eff` of every parameter of interest, and of any interesting parameter combinations. In general, we need $n.eff \geq 1000$ 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:

```
neff1 <- coda::effectiveSize(d$accepted_traj[d$proposal_sd=="Proposal SD = 0.1"])
neff2 <- coda::effectiveSize(d$accepted_traj[d$proposal_sd=="Proposal SD = 1"])
neff3 <- coda::effectiveSize(d$accepted_traj[d$proposal_sd=="Proposal SD = 10"])
```

```
df <- tibble("Proposal SD" = c(0.1, 1, 10),
             "n.eff" = round(c(neff1, neff2, neff3)))
df
## # A tibble: 3 x 2
##   `Proposal SD` `n.eff`
##       <dbl>    <dbl>
## 1      0.1     224
## 2      1       1934
## 3     10      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 is the mixing good and 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 (also known as the folk theorem of statistical computing as stated by Andrew Gelman in 2008, see https://statmodeling.stat.columbia.edu/2008/05/13/the_folk_theorem/). 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 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 understand how it works, what it does not do, and the data you need to get reasonable parameter estimates.

1.7 Summary

- With the Bayes' theorem, you update your beliefs (prior) with new data (likelihood) to get posterior beliefs (posterior): $\text{posterior} \propto \text{likelihood} \times \text{prior}$.
- 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.

1.8 Suggested reading

- [McCarthy \[2007\]](#) is an excellent introduction to Bayesian statistics for ecologists.
- For deeper insights, I recommend [Gelman and Hill \[2006\]](#) which

analyse data using the frequentist and Bayesian approaches side-by-side. The book by [McElreath \[2020\]](#) is also an excellent read. The presentation of the Metropolis algorithm in Section 1.5.3 was inspired by [Albert and Hu \[2019\]](#). If you'd like to know more about Monte Carlo methods, the book [Robert and Casella \[2004\]](#) is a must (see also its R counterpart [Robert and Casella \[2010\]](#)).

- I also recommend [Gelman et al. \[2020\]](#) in which the authors offer a workflow for Bayesian analyses. They discuss model building, model comparison, model checking, model validation, model understanding and troubleshooting of computational problems.

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. Should you wish to dive deeper into the mechanics, NIMBLE also got you covered and allows you to write samples, use custom functions, etc. We will illustrate NIMBLE’s main features with a simple example, but the ideas hold for more complex 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 makes this easy by using a syntax very similar to the R syntax, which should make your life easier. It is also a direct extension of the 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), Monte Carlo Expectation Maximization (for maximum likelihood), Hamiltonian Monte Carlo (like in program Stan), and Laplace approximation (like in program TMB). 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. The NIMBLE users google group is an open and inclusive space where everyone can receive help from the community: <https://groups.google.com/g/nimble-users>.



FIGURE 2.1: Logo of the NIMBLE R package designed by Luke Larson.

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 info and help trouble-shooting installation issues can be found here: <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)
## }

```

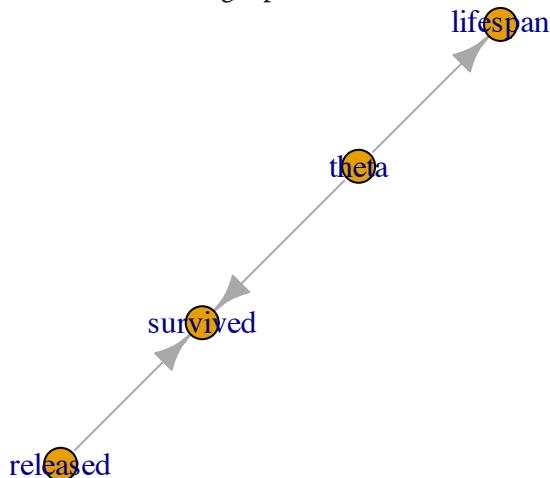
In the code above, `survived` and `released` are known, only `theta` needs to be estimated. The line `survived ~ dbinom(theta, released)` states that the number of successes or animals that have survived over winter, `survived`, is distributed as (that's the `~`) a binomial with `released` trials and probability of success or survival `theta`. Then the line `theta ~ dunif(0, 1)` assigns a uniform distribution between 0 and 1 as a prior 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. If you'd like to know more about the calculation of life expectancy, check out [Cook et al. \[1967\]](#).

A few comments:

- The most common distributions are readily 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 distributions 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 arrows from parents to children. Such graphs are called directed acyclic graph or DAG.



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 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. It will suggest you to move some data into constants to improve efficiency. 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. As a bare minimum, you need initial values for all nodes that only appear on the left side of a `~` in your code and are not given as data. 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 (here we specify for three chains) 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 an R function that generates random initial values:

```
initial.values <- function() list(theta = runif(1,0,1))
initial.values()
## $theta
## [1] 0.8356
```

If you are using a function to generate random initial values, it's always a good idea to set the seed in your code before you draw the initial values. For example like this:

```
my.seed <- 666
set.seed(my.seed)
```

Setting the seed makes your code reproducible, which really helps if you need to trouble-shoot it later. Initialization problems are not uncommon when working with NIMBLE, and being able to reproduce the same initial values again is very useful for solving them.

Fifth 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:

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

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. Thinning is fixed to 1 by default in NIMBLE so that all simulations are used to summarise posterior distributions. [Link and Eaton \[2012\]](#) offer a discussion of the pros and cons of thinning.

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

```
mcmc.output <- nimbleMCMC(code = model,
                             data = my.data,
                             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, one is `setSeed`. Just like with sampling initial values above, setting the seed within the MCMC call allows you to run the **same** chains (again), thus making your analyses reproducible and problems easier to debug (see Section [2.7.5](#)). You can also get a summary of the outputs by specifying `summary = TRUE`. Conversely, if you

would rather just get the MCMC samples back (in `coda mcmc` format) you can set `samplesAsCodaMCMC = TRUE`. Finally, you can suppress the progress bar if you find it too depressing when running long simulations with `progressBar = FALSE`. 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] 0.907 0.907 0.907 0.907 0.853 ...
##   ..- attr(*, "dimnames")=List of 2
##     ... .:$ : NULL
##     ... .:$ : chr [1:2] "lifespan" "theta"
## $ chain2: num [1:4000, 1:2] 0.787 0.894 1.291 1.388 1.388 ...
##   ..- attr(*, "dimnames")=List of 2
##     ... .:$ : NULL
##     ... .:$ : chr [1:2] "lifespan" "theta"
## $ chain3: num [1:4000, 1:2] 0.745 0.745 0.745 0.886 1.136 ...
##   ..- 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,] 0.9069 0.3320
## [2,] 0.9069 0.3320
## [3,] 0.9069 0.3320
## [4,] 0.9069 0.3320
## [5,] 0.8526 0.3095
## [6,] 0.7987 0.2859
```

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`:

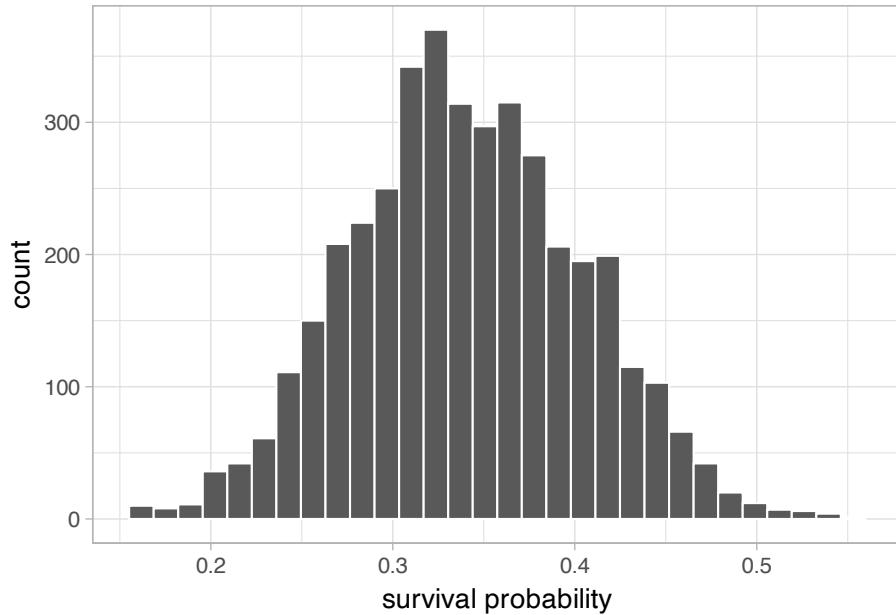
```
mean(mcmc.output$chain1[, 'theta'])  
## [1] 0.3407
```

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%  
## 0.2219 0.4620
```

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

```
mcmc.output$chain1[, "theta"] %>%  
  as_tibble() %>%  
  ggplot() +  
  geom_histogram(aes(x = value), 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` to summarise and visualize MCMC outputs, but there are other perfectly valid options out there like `ggmcmc`, `bayesplot` and `basicMCMCplots`.

Let's load the package `MCMCvis`:

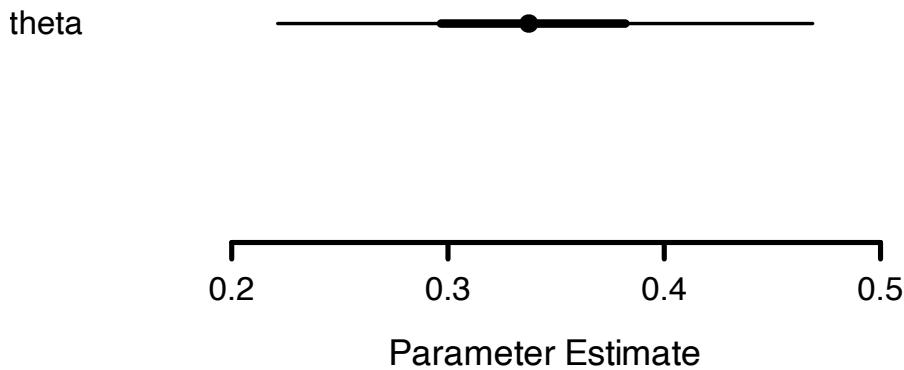
```
library(MCMCvis)
```

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  0.17 0.66 0.92  1.32     1 2513
## theta    0.34  0.06 0.22 0.34  0.47     1 2533
```

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

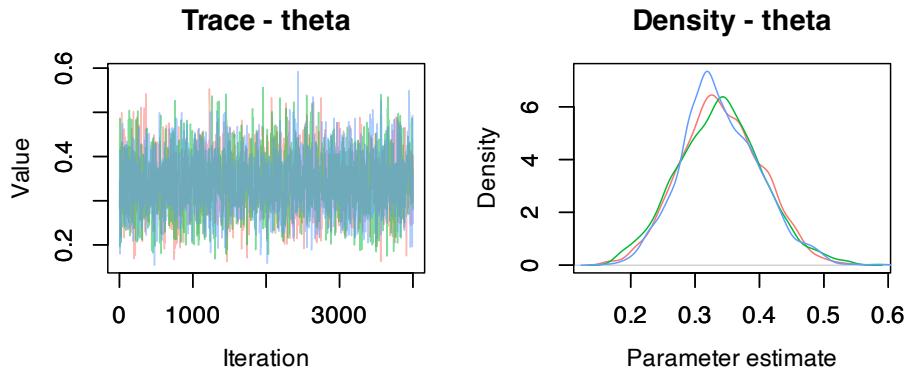
```
MCMCplot(object = mcmc.output,
          params = 'theta')
```



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

Visualization of a MCMC chain itself, i.e. the values of posterior samples plotted against iteration number, is called a trace. The trace and posterior density of θ can be obtained with `MCMCtrace()`:

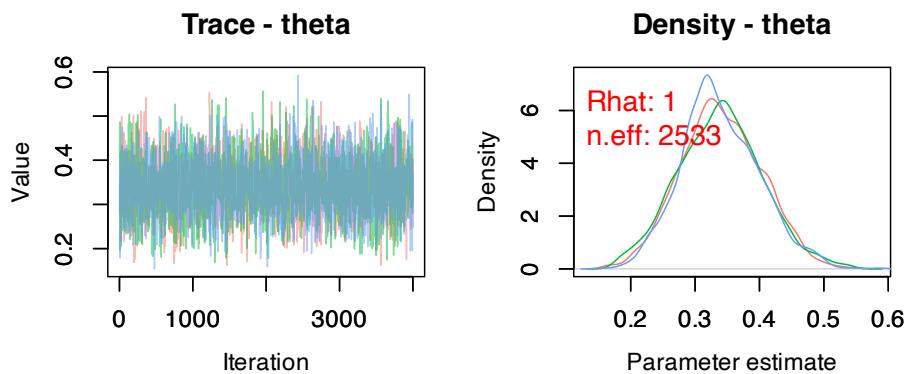
```
MCMCtrace(object = mcmc.output,
           pdf = FALSE, # no export to PDF
           ind = TRUE, # separate density lines per chain
           params = "theta")
```



We use the trace and density plots for assessing convergence and get an idea of whether there may be any estimation issues (see Section 1.6).

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 a function of your model parameters by applying this function to samples from the posterior distribution of these parameters. Especially when working with big models/data, it is recommended to keep any calculations that can be made “post-hoc” using the posterior samples outside of NIMBLE as this lessens memory load. 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:

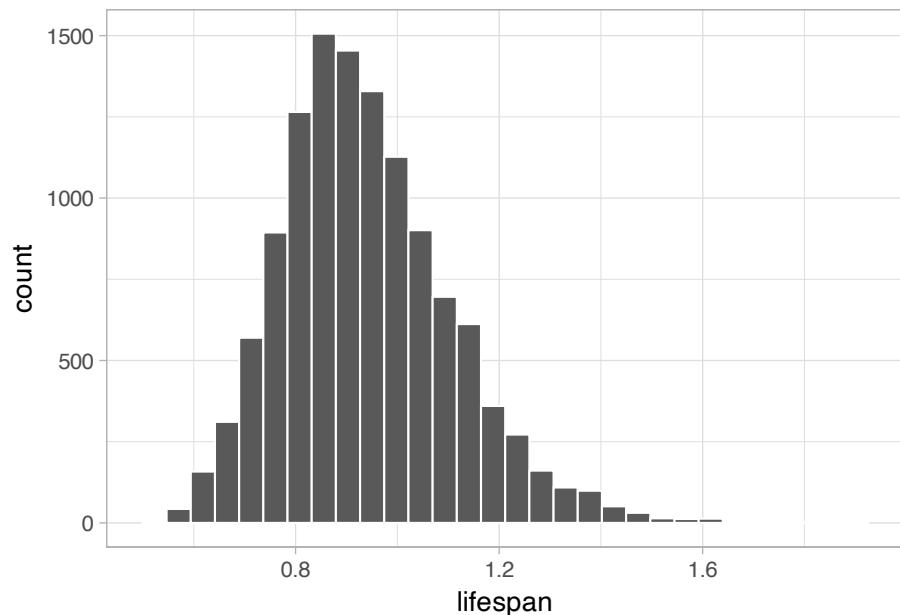
```
lifespan <- -1/log(theta_samples)
```

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

```
mean(lifespan)
## [1] 0.9398
quantile(lifespan, probs = c(2.5, 97.5)/100)
##    2.5%   97.5%
## 0.6629 1.3194
```

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 `nimbleCMC()` 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 and often faster runtime.

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 0.94 0.16 0.66 0.92  1.31     1  2593
## theta    0.34 0.06 0.22 0.34  0.47     1  2652

```

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 0.68 0.92  1.29    1 2597
## theta    0.34 0.06 0.23 0.34  0.46    1 2643
```

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

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 = size! / [x! (n-x)!] and take log
    lchoose <- lfactorial(size) - lfactorial(x) - lfactorial(size - x)
    # binomial density function = size! / [x! (n-x)!] * prob^x * (1-prob)^(size-x) and take log
    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)
# note: the n argument is required by NIMBLE but is not used, default is 1
rmybinom <- nimbleFunction(
  run = function(n = integer(0, default = 1),
                 size = double(0),
                 prob = 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
      }
    }
    returnType(double(0))
    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 single random value ($n = 1$ by default) from a binomial distribution with size 5 and probability 0.1:

```
rmybinom(size = 5, prob = 0.1)
## [1] 0
```

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)
## |-----|-----|-----|-----|
## |-----|
## |-----|-----|-----|-----|
## |-----|-----|-----|-----|
```

```
## |-----|-----|-----|-----|
## |-----|-----|-----|-----|
MCMCsummary(mcmc.output)
##      mean     sd   2.5%   50% 97.5% Rhat n.eff
## theta 0.34 0.05976 0.2286 0.3378 0.4598    1 2970
```

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 perfectly fine for most applications. However, in some situations you need to customize the MCMC samplers to improve or speed up 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. For example, if we provide a negative initial value for `theta`, `survival$calculate()` returns NA:

```

survival <- nimbleModel(code = model,
                         data = my.data,
                         inits = list(theta = -0.5))
survival$calculate()
## [1] NaN

```

As another example, if we convey in the data the information that more animals survived than were released, we'll get an infinity value for the log-likelihood:

```

my.data <- list(survived = 61, released = 57)
initial.values <- list(theta = 0.5)
survival <- nimbleModel(code = model,
                        data = my.data,
                        inits = initial.values)
survival$calculate()
## [1] -Inf

```

As a check that the model is correctly initialized and that your code is without bugs, the call to `model$calculate()` should return a number and not NA or -Inf:

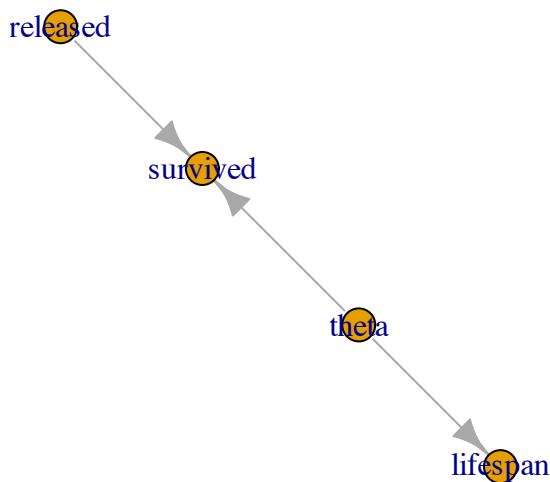
```

my.data <- list(survived = 19, released = 57)
initial.values <- list(theta = 0.5)
survival <- nimbleModel(code = model,
                        data = my.data,
                        inits = initial.values)
survival$calculate()
## [1] -5.422

```

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: lifespan, theta
survivalConf
## ===== Monitors =====
## thin = 1: lifespan, theta
## ===== 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 = CsuvivalMCMC,
                     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,] 0.9093 0.3330
## [2,] 0.9093 0.3330
## [3,] 0.9093 0.3330
## [4,] 1.2095 0.4374
## [5,] 1.2095 0.4374
## [6,] 1.1835 0.4296
```

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

```
samplesSummary(samples)
##           Mean Median St.Dev. 95%CI_low 95%CI_upp
## lifespan 0.9357 0.9194 0.16117    0.6831    1.2969
## theta    0.3386 0.3370 0.06128    0.2313    0.4625
```

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,
```

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 0.9357 0.9231 0.16002   0.6645   1.2826
## theta    0.3387 0.3385 0.06098   0.2221   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.

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 = CsuvivalMCMC,
                     niter = 5000,
                     nburnin = 1000)
## |-----|-----|-----|-----|
## |-----|
samplesSummary(samples)
##      Mean Median St.Dev. 95%CI_low 95%CI_upp
## theta 0.339 0.3377 0.05592   0.2374   0.4528

```

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

```

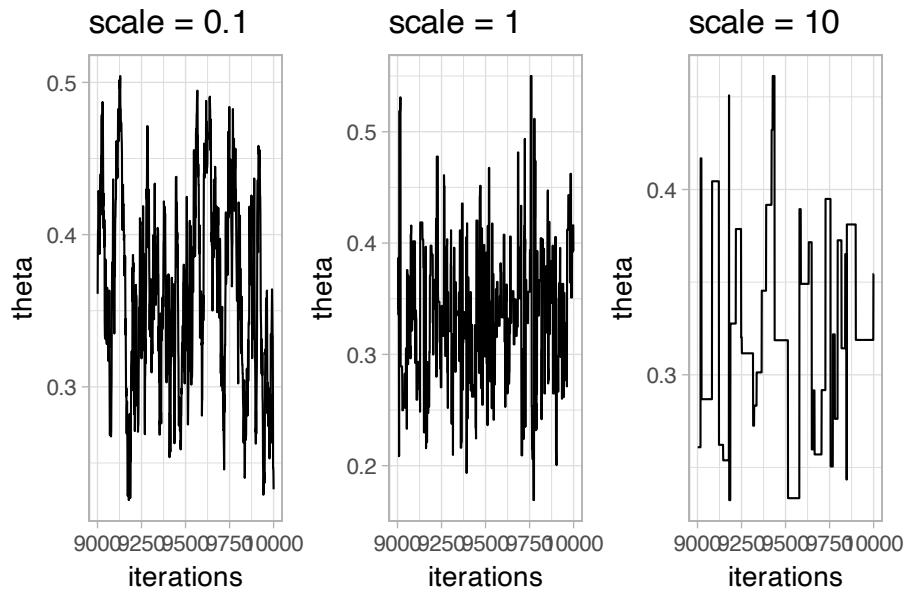
# standard deviation of proposal is 0.1
scale <- 0.1
Rmodel <- nimbleModel(code = model, data = my.data, inits = initial.values())
conf <- configureMCMC(Rmodel, monitors = c('theta'), print = FALSE)
conf$removeSamplers(c('theta'))
conf$addSampler(target = 'theta', type = 'my_metropolis', control = list(scale = scale))
Rmcmc <- buildMCMC(conf)
out <- compileNimble(list(model = Rmodel, mcmc = Rmcmc))
Cmcmc <- out$mcmc
samples_sd01 <- runMCMC(Cmcmc, niter = 10000, nburnin = 9000, progressBar = FALSE)
# standard deviation of proposal is 1
scale <- 1
Rmodel <- nimbleModel(code = model, data = my.data, inits = initial.values())
conf <- configureMCMC(Rmodel, monitors = c('theta'), print = FALSE)
conf$removeSamplers(c('theta'))
conf$addSampler(target = 'theta', type = 'my_metropolis', control = list(scale = scale))
Rmcmc <- buildMCMC(conf)
out <- compileNimble(list(model = Rmodel, mcmc = Rmcmc))

```

```

Cmcmc <- out$mcmc
samples_sd1 <- runMCMC(Cmcmc, niter = 10000, nburnin = 9000, progressBar = FALSE)
# standard deviation of proposal is 10
scale <- 10
Rmodel <- nimbleModel(code = model, data = my.data, inits = initial.values())
conf <- configureMCMC(Rmodel, monitors = c('theta'), print = FALSE)
conf$removeSamplers(c('theta'))
conf$addSampler(target = 'theta', type = 'my_metropolis', control = list(scale = scale))
Rmcmc <- buildMCMC(conf)
out <- compileNimble(list(model = Rmodel, mcmc = Rmcmc))
Cmcmc <- out$mcmc
samples_sd10 <- runMCMC(Cmcmc, niter = 10000, nburnin = 9000, progressBar = FALSE)
# trace plot for scenario with standard deviation 0.1
plot01 <- samples_sd01 %>%
  as_tibble() %>%
  ggplot() +
  aes(x = 9001:10000, y = theta) +
  geom_line() +
  labs(x = "iterations", title = "scale = 0.1")
# trace plot for scenario with standard deviation 1
plot1 <- samples_sd1 %>%
  as_tibble() %>%
  ggplot() +
  aes(x = 9001:10000, y = theta) +
  geom_line() +
  labs(x = "iterations", title = "scale = 1")
# trace plot for scenario with standard deviation 10
plot10 <- samples_sd10 %>%
  as_tibble() %>%
  ggplot() +
  aes(x = 9001:10000, y = theta) +
  geom_line() +
  labs(x = "iterations", title = "scale = 10")
# Assemble all three trace plots
library(patchwork)
plot01 + plot1 + plot10

```



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.

2.7.1 Precision vs standard deviation

In other software 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 pa-

parameterisation as well as the more natural parameterisation $\epsilon \sim dnorm(\mu, sd = 100)$ 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 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. The downside of not doing the `calculate()`, is that you might not be able to identify issues that could help you save time in the long run. You can also use `useConjugacy = FALSE` in `configureMCMC()` to disable the search for conjugate samplers. With the animal survival example, you would do:

```

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 = CsuvrivalMCMC,
                     niter = 5000,
                     nburnin = 1000)
## |-----|-----|-----|
## |-----|
samplesSummary(samples)
##           Mean Median St.Dev. 95%CI_low 95%CI_upp
## theta 0.3402 0.3391 0.06029    0.2258    0.4616

```

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
CsuvrivalMCMC$run(niter_ad, reset = FALSE)
## |-----|-----|-----|
## |-----|

```

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 0.3402 0.3382 0.05975    0.2281    0.4632
```

You can check that `more_samples` contains 10000 samples, 4000 from the call to `runMCMC()` plus 6000 additional samples. Note that this only works if you are using the detailed NIMBLE workflow. It does not work with the `nimbleMCMC()` wrapper function.

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. Note that we need to specify a seed for each chain, hence a vector of three components here:

```
# first call to nimbleMCMC()
mcmc.output1 <- nimbleMCMC(code = model,
                           data = my.data,
                           inits = initial.values,
                           niter = 5000,
                           nburnin = 1000,
                           nchains = 3,
                           summary = TRUE,
                           setSeed = c(2024, 2025, 2026))

## |-----|-----|-----|-----|
## |-----|
## |-----|-----|-----|
## |-----|
## |-----|-----|-----|
## |-----|-----|-----|
```

```

# second call to nimbleMCMC()
mcmc.output2 <- nimbleMCMC(code = model,
                             data = my.data,
                             inits = initial.values,
                             niter = 5000,
                             nburnin = 1000,
                             nchains = 3,
                             summary = TRUE,
                             setSeed = c(2024, 2025, 2026))

## |-----|-----|-----|-----|
## |-----|-----|-----|-----|
## |-----|-----|-----|-----|
## |-----|-----|-----|-----|
## |-----|-----|-----|-----|
## |-----|-----|-----|-----|
## |-----|-----|-----|-----|
# outputs from both calls are the same
mcmc.output1$summary$all.chains
##      Mean Median St.Dev. 95%CI_low 95%CI_upp
## theta 0.339 0.3363 0.06141    0.2245    0.4618
mcmc.output2$summary$all.chains
##      Mean Median St.Dev. 95%CI_low 95%CI_upp
## theta 0.339 0.3363 0.06141    0.2245    0.4618

```

Note that to make your workflow reproducible, you need to set the seed not only within the `nimbleMCMC()` call, but also before setting your initial values if you are using a randomized function for that.

2.7.6 Parallelization

To speed up your analyses, you can run MCMC chains in parallel. This is what the package `jagsUI`¹ accomplishes for JAGS users. Here, we use the `parallel` package for parallel computation:

¹<https://github.com/kenkellner/jagsUI>

```
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 = runitif(1,0,1))

  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 values 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 and incorrect 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 purposefully 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 0.3402 0.338 0.06025    0.2277    0.4608
```

When working with larger models, it can happen that NIMBLE's internal simulation produces initial values for different parameters (nodes) that are incompatible with each other and violate certain model assumptions. If we go ahead and run the MCMC on a model where this is the case, a range of different warning messages may appear to indicate the problem. At first, they may not seem very intuitive (e.g. "Log probability is -Inf", "Log probability is NaN", "Slice sampler reached the maximum number of contractions", etc.), but they are signals that you may want to double-check our initialization.

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 $x[1], \dots, x[n]$. As an example, you may have a look to the vectorized flavor of the binomial distribution written by Pierre Dupont at <https://github.com/nimble-dev/>

[nimbleSCR/blob/master/nimbleSCR/R/dbinom_vector.R](#). Note that per now, vectorization only works for deterministic nodes (relationships with `<-`) but not stochastic ones (relationships with `~`).

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.
- 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/ch-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.
- 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.
- 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. *Journal of Computational and Graphical Statistics* **26** (2): 403–13.

- Last, the packages to process NIMBLE results are developed by people whose work should be acknowledged: see [Youngflesh \[2018\]](#) for `MCMCvis`, [Turek \[2022\]](#) for `basicMCMCplots`, [Gabry and Mahr \[2022\]](#) for `bayesplot` and [Fernández-i Marín \[2016\]](#) for `ggmcmc`.

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:

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

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

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

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.

```
## # A tibble: 57 x 6
##       id `winter 1` `winter 2` `winter 3` `winter 4` `winter 5`
##     <int>     <int>     <int>     <int>     <int>
## 1     1         1         1         1         1
## 2     2         2         1         1         1
## 3     3         3         1         1         1
## 4     4         4         1         1         1
## 5     5         5         1         1         1
## 6     6         6         1         1         2
## 7     7         7         1         1         1
## 8     8         8         1         2         2
## 9     9         9         1         1         1
## 10   10        10        1         2         2
## # i 47 more rows
## # i 1 more variable: `winter 5` <int>
```

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 have already encountered Markov models in Section 1.5.2.

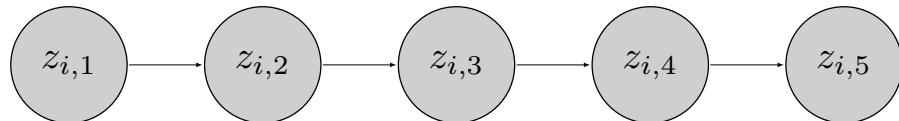
3.3.1 Assumptions

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.

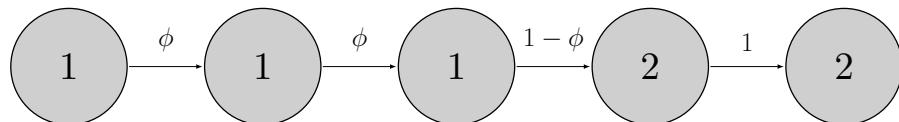
Second, if an animal is alive in a given winter, the probability it survives to the next winter is ϕ . The probability it dies is $1 - \phi$.

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 $\Pr(z_t = 2 | z_{t-1} = 2) = 1$.

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

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

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:

$$\times = \begin{pmatrix} z_t = 1 & z_t = 2 \\ \phi & 1 - \phi \\ 0 & 1 \end{pmatrix} \begin{matrix} z_{t-1} = 1 \text{ (alive)} \\ z_{t-1} = 2 \text{ (dead)} \end{matrix}$$

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 $\times = (\Pr(z_{i,1} = 1), \Pr(z_{i,1} = 2))$. 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 $\times = (1, 0)$.

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 $\Pr(\mathbf{z}) = \Pr(z_1, z_2, \dots, z_{T-2}, z_{T-1}, z_T)$.

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:

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

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

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

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

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

Then apply the Markovian property:

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

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

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

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:

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

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:

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

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. The categorical distribution is a multinomial dis-

tribution with a single draw. 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:

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

Alternatively, you can use the `sample()` function and `sample(x = 1:3, size = 1, replace = FALSE, prob = c(0.1, 0.3, 0.6))`. 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 to be alive, and one minus that probability to be dead. Of course, if you want your Markov chain to start, you'd better say it's alive so that δ is just (1, 0):

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

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 $\times = \gamma_{z_{t-1}, z_t}$:

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

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

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

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 \times :

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

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:

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

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 $z[i,1] \leftarrow 1$.

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 replace `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()`, `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, in which animals are captured, individually marked, and released alive. Then, these animals may be detected again, or go undetected. 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:

```
## # A tibble: 57 x 6
##       id `winter 1` `winter 2` `winter 3` `winter 4` 
##   <int>     <int>     <int>     <int>     <int>
## 1     1         1         1         1         1
## 2     2         2         1         1         1
## 3     3         3         1         1         1
## 4     4         4         1         1         1
## 5     5         5         1         1         1
## 6     6         6         1         1         2
## 7     7         7         1         1         1
## 8     8         8         1         2         2
## 9     9         9         1         1         1
```

```
## 10      10      1      2      2      2
## # i 47 more rows
## # i 1 more variable: `winter 5` <int>
```

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 $y = 0$:

```
## # A tibble: 57 x 6
##       id `winter 1` `winter 2` `winter 3` `winter 4` 
##   <int>     <int>     <int>     <int>     <int>
## 1     1         1         1         1         1
## 2     2         2         1         1         1
## 3     3         3         1         1         1
## 4     4         4         1         1         1
## 5     5         5         1         1         1
## 6     6         6         1         1         0
## 7     7         7         1         1         1
## 8     8         8         1         0         0
## 9     9         9         1         1         1
## 10    10        10        1         0         0
## # i 47 more rows
## # i 1 more variable: `winter 5` <int>
```

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.

```
## # A tibble: 57 x 6
##       id `winter 1` `winter 2` `winter 3` `winter 4` 
##   <int>     <dbl>     <dbl>     <dbl>     <dbl>
## 1     1         1         0         0         0
## 2     2         2         1         0         1         0
```

```

##   3     3      1      0      0      0
##   4     4      1      1      1      1
##   5     5      1      1      1      1
##   6     6      1      0      0      0
##   7     7      1      0      1      1
##   8     8      1      1      1      1
##   9     9      1      1      1      1
##  10    10     1      1      0      0
## # i 47 more rows
## # i 1 more variable: `winter` <dbl>

```

Compare with the previous z 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 y table 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 \times . 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):

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

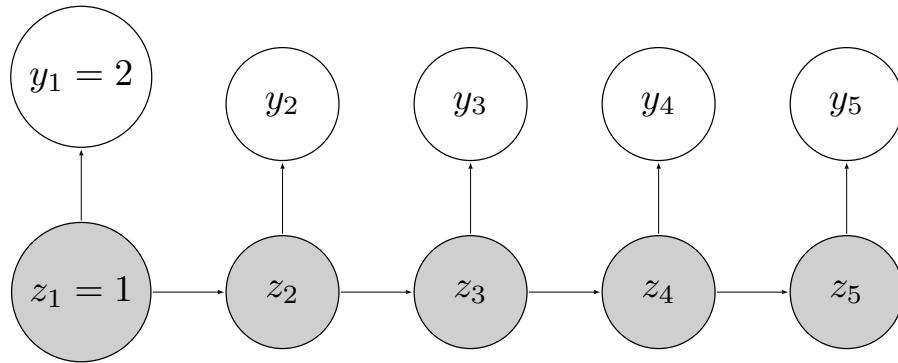
In survival models we will use throughout this book, we condition the fate of individuals on first detection, which boils down to set the corresponding detection probability to 1.

The observation matrix is:

$$\times = \begin{pmatrix} y_t = 1 & y_t = 2 \\ \text{(non-detected)} & \text{(detected)} \\ 1-p & p \\ 1 & 0 \end{pmatrix} z_t = 1 \text{ (alive)} \\ z_t = 2 \text{ (dead)}$$

3.6.3 Hidden Markov model

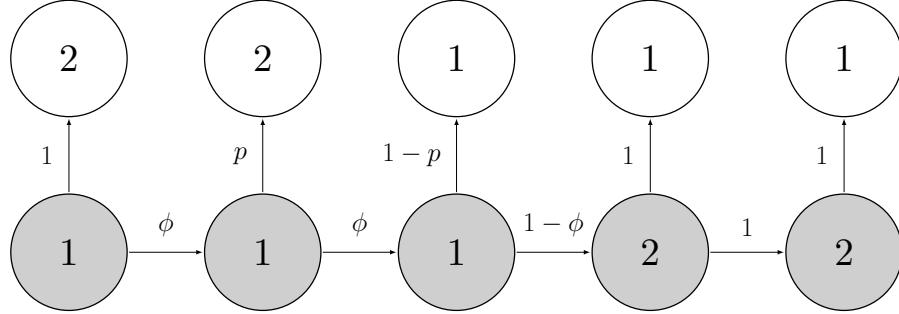
Our hidden Markov model can be represented this way:



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 for the observations generated from the states. HMM are a special case of state-space models in which latent states are discrete.

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 $z_4 = z_5 = 2$. Because we condition on first detection, the link between state and observation at $t = 1$ is deterministic and $p = 1$.



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:

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

Using the definition of a conditional probability, we have:

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

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

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

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

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

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

If we denote *first* the time of first detection, then our model so far is written as follows:

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

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 = first) = 1
delta[2] <- 0          # Pr(dead t = first) = 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])
    }
  }
}
```

The loop over time for each individual `for (j in 2:T){}` starts after the first time individuals are detected (this is time 2 for all of them here), because we work conditional on the first detection.

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 = first) = 1
  delta[2] <- 0           # Pr(dead t = first) = 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:

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

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

```

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

```

As initial values for the latent states, we assumed that whenever an individual was non-detected, it was alive, with `zinit <- y + 1`, and we make sure dead individuals are alive with `zinit[zinit == 2] <- 1`.

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 32.43 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, as we condition on first detection
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, you get the marginal likelihood by summing over all possible states in the complete likelihood:

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

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 is coded 2, while alive is coded 1 and dead is coded 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:

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

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:

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

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

$$\Pr(y_1 = 2, y_2 = 2, y_3 = 1) = 1(1 - p) \times 1\phi\phi + 1p1 \times 1\phi(1 - \phi)$$

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 we will deal with more states 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, some 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 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, \dots, y_t , that is $\alpha_t(j) = \Pr(y_1, \dots, y_t, z_t = j)$.

Using the law of total probability, we can write the marginal likelihood as a function of $\alpha_T(j)$, namely we have $\Pr(\mathbf{y}) = \sum_{j=1}^N \Pr(y_1, \dots, y_t, z_t = j) = \sum_{j=1}^N \alpha_T(j)$.

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:

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

Second, using conditional probabilities, we get:

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

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

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

which, using conditional independence, simplifies into:

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

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

In practice, the forward algorithm works as follows. First you initialize the procedure by calculating for the states $j = 1, \dots, N$ the quantities $\alpha_1(j) = \Pr(z_1 = j) \omega_{j,y_1}$. Then you compute for the states $j = 1, \dots, N$ the relationship $\alpha_t(j) = \sum_{i=1}^N \alpha_{t-1}(i) \gamma_{i,j} \omega_{j,y_t}$ for $t = 2, \dots, T$.

Finally, you compute the marginal likelihood as $\Pr(\mathbf{y}) = \sum_{j=1}^N \alpha_T(j)$.

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, which is much less than the $2TN^T$ calculations 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:

$$\alpha_1(1) = \Pr(z_1 = 1)\omega_{1,y_1=2} = 1$$

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

$$\alpha_1(2) = \Pr(z_1 = 2)\omega_{2,y_1=2} = 0$$

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:

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

and:

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

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

3.8.3.1 Do it yourself

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:

```
dHMMhomemade <- nimbleFunction(
  run = function(x = double(1),
                 probInit = double(1), # vector of initial states
                 probObs = double(2), # observation matrix
                 probTrans = double(2), # transition matrix
                 len = double(0, default = 0), # number of sampling occasions
                 log = integer(0, default = 0)) {
  alpha <- probInit[1:2] # * probObs[1:2,x[1]] == 1 due to conditioning on first detection
  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 (see Section 2.5). Then we write a function to simulate values from a HMM:

```

rHMMhomemade <- 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('dHMMhomemade', dHMMhomemade, .GlobalEnv)
assign('rHMMhomemade', rHMMhomemade, .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] ~ dHMMhomemade(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)
## |-----|-----|-----|-----|
## |-----|
## |-----|-----|-----|-----|
## |-----|
end_time <- Sys.time()
end_time - start_time
## Time difference of 26.49 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
```

3.8.3.2 Do it with `nimbleEcology`

Writing NIMBLE functions is not easy. Fortunately, the NIMBLE folks got you covered. They developed the package `nimbleEcology` that implements some of the most popular ecological models with latent states.

We will use the function `dHMM0` which provides the distribution of a hidden Markov model with time-independent transition matrix and time-dependent observation matrix. Why time-dependent observation matrix? Because we need to tell NIMBLE that detection at first encounter is 1.

We load the package:

```
library(nimbleEcology)
```

The NIMBLE code is:

```
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] <- 0      # Pr(alive first -> non-detected first)
  omega[1,2,1] <- 1      # Pr(alive first -> detected first)
  omega[2,1,1] <- 1      # Pr(dead first -> non-detected first)
  omega[2,2,1] <- 0      # Pr(dead first -> detected first)
  for (t in 2:5){
    omega[1,1,t] <- 1 - p    # Pr(alive t -> non-detected t)
    omega[1,2,t] <- p        # 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)
}
for (i in 1:N){
  y[i,1:5] ~ dHMMo(init = delta[1:2], # vector of initial state probabilities
                     probObs = omega[1:2,1:2,1:5], # observation matrix
                     probTrans = gamma[1:2,1:2], # transition matrix
                     len = 5, # nb of sampling occasions
                     checkRowSums = 0) # skip validity checks
}
})

```

You may see that we no longer have the states in the code as we use the marginalized likelihood. The `dHMMo` takes several arguments, including `init` the vector of initial state probabilities, `probObs` the observation matrix, `probTrans` the transition matrix and `len` the number of sampling occasions.

Next steps are similar to the workflow we used before. The only difference is that we do not need to specify initial values for the latent states:

```

# constants
my.constants <- list(N = nrow(y))
# 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

```

Now we 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 31.11 secs

```

Now we display the numerical summaries of the posterior distributions:

```

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 similar what we obtained previously with our home-made marginalized likelihood (Section 3.8.3.1), or with the full likelihood (3.7).

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. This idea is used in routine in capture-recapture software. For Bayesian software however, it is only recently that the trick was tested in NIMBLE by Turek et al. [2016].

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 `dHMMhomemade` 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),
                 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 `rHMMhomemade` 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)
## |-----|-----|-----|
## |-----|-----|-----|
## |-----|-----|-----|
## |-----|-----|
end_time <- Sys.time()
end_time - start_time
## Time difference of 27.01 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 1455
## phi 0.76 0.04 0.67 0.76  0.84    1 1524
```

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.

The pooled likelihood is not yet implemented in `nimbleEcology`, but you can hack the code for the function `dHMM` <https://github.com/nimble-dev/nimbleEcology/blob/master/R/dHMM.R> to implement it yourself by adding a `size` argument.

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

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 $p = 0.6$.

Imagine you do not know the truth. What is the chance that animal #15 was alive throughout the study when observing the encounter history detected in first winter, then missed in each subsequent winter? The chance of being alive in the first winter and detected when alive is 1. The chance of being alive in the second winter and non-detected is $0.8 \times (1 - 0.6) = 0.32$. The same goes for third, fourth and fifth winters. In total, the probability of being alive throughout the study for an animal with encounter history (2, 1, 1, 1, 1) is $1 \times 0.32 \times 0.32 \times 0.32 \times 0.32 = 0.01048576$.

Now what is the chance that animal #15 was alive, then dead for the rest of the study, when observing the encounter history (2, 1, 1, 1, 1)? And the chance of being alive in first and second winters, then dead after when observing the same encounter history? And so on. You need to enumerate all possible sequences of states and compute the probability for each of them, and choose the most probable sequence, that is with maximum probability. In our example, we would need to compute $2^5 = 32$ of these probabilities, and N^T in general. Needless to say, these calculations quickly become cumbersome, if not impossible, as the number of states and/or the number of sampling occasions increases.

This is where the Viterbi algorithm comes in. The idea is to decompose this overall complex problem in a sequence of smaller problems that are easier to solve. If dynamic programming rings a bell, the Viterbi algorithm should look familiar to you. The Viterbi algorithm is based on the fact that the optimal path to each winter and each state can be deduced from the optimal path to the previous winter and each state.

For first winter, the probability of being alive and detected is 1, while the probability of being dead and detected is 0. Now what is the probability of being alive in the second winter and non-detected? If the animal was alive in the first winter, it remains alive and is missed with probability $1 \times \phi(1 - p) = 0.32$. If it was dead in the first winter,

then this probability is 0. The maximum probability is 0.32 obviously so the most probable scenario to being alive in the second winter is being alive in the first winter. What about being dead in the second winter? If the animal was alive in first winter, then the probability is $1 \times (1 - \phi) \times 1 = 0.2$. If dead, then this probability is $0 \times 0 \times (1 - p) = 0$. The maximum probability is 0.2 obviously so the most probable scenario to being dead in the second winter is being alive in the first winter. Doing these calculations for third, fourth and fifth winters, we get the probabilities:

	winter				
	1	winter2	winter 3	winter 4	winter 5
state	1	$0.32 =$	$0.2304 =$	$0.110592 =$	$0.053084416 =$
alive		$\max(0,$ $0.32)$	$\max(0,$ $0.2304)$	$\max(0,$ $0.110592)$	$\max(0,$ $0.05308416)$
state	0	$1 = \max(1,$ $0.2)$	$1 =$ $\max(0.096,$ $1)$	$1 =$ $\max(0.04608,$ $1)$	$1 =$ $\max(0.0221184,$ $1)$
dead					
observation	acted	missed	missed	missed	missed

Finally, to get (or decode) the optimal path, you work backwards and trace back the previous value that yielded the maximum probability. The most probable state in the last winter is dead ($1 > 0.05308416$), dead again in the fourth winter ($1 > 0.110592$), dead in the third winter ($1 > 0.2304$), alive in the second winter ($1 > 0.48$) and alive in the first winter ($1 > 0$). According to the Viterbi algorithm, the sequence of states that most likely generated the sequence of observations (2, 1, 1, 1, 1) is alive then dead, dead, dead and dead or (1 2 2 2 2). This differs slightly from the actual sequence of states (1, 1, 2, 2, 2) in that the state in second winter is decoded dead while animal #15 only dies in third winter.

In contrast to the brute force approach, calculations are not duplicated but stored and used again like in the forward algorithm. Briefly speaking, the Viterbi algorithm works like the forward algorithm where sums are replaced by calculating maximums.

In practice, the Viterbi algorithm works as illustrated in Figure 3.1. First you initialize the procedure by calculating at $t = 1$ for all states

$j = 1, \dots, N$ the values $\nu_1(j) = \Pr(z_1 = j)\omega_{j,y_1}$. Then you compute for all states $j = 1, \dots, N$ the values $\nu_t(j) = \max_{i=1, \dots, N} \nu_{t-1}(i)\gamma_{i,j}\omega_{j,y_t}$ for $t = 2, \dots, T$. Here at each time t we determine the probability of the best path ending at each of the states $j = 1, \dots, N$. Finally, you compute the probability of the best global path $\max_{j=1, \dots, N} \nu_T(j)$.

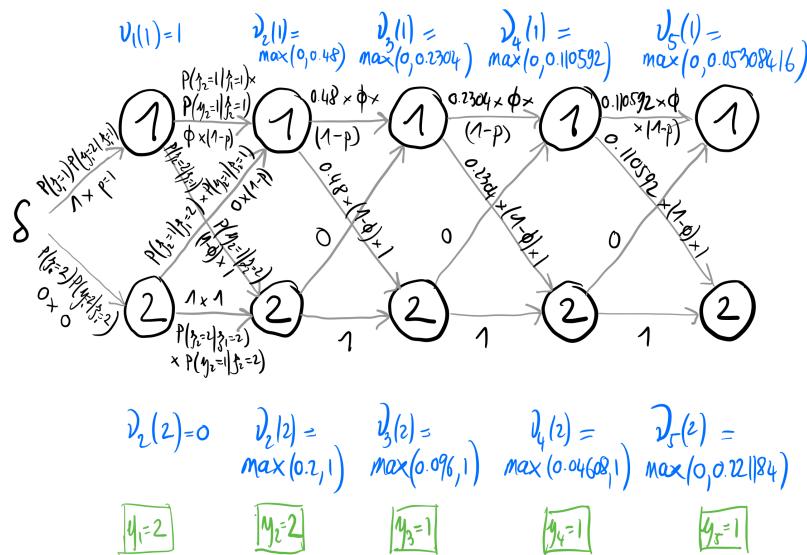


FIGURE 3.1: Graphical representation of the Viterbi algorithm with $\phi = 0.8$ and $p = 0.6$. States are alive $z = 1$ or dead $z = 2$ and observations are non-detected $y = 1$ or detected $y = 2$. To be done properly w/ tikz.

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)

  # nu is the corresponding likelihood
  nu <- matrix(0, nrow = N, ncol = T)
  # zz contains the most likely states up until this point
  zz <- matrix(0, nrow = N, ncol = T)
  firstObs <- y[1]

  # fill in first columns of both matrices
  #nu[,1] <- initial * emission[,firstObs]
  #zz[,1] <- 0
  nu[,1] <- c(1,0) # initial = (1, 0) * emission[,firstObs] = (1, 0)
  zz[,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
      nu[j,i] <- -1
      # loop to find max and argmax for k
      for (k in 1:N) {
        value <- nu[k,i-1] * Gamma[k,j] * Omega[j,obs]
        if (value > nu[j,i]) {
```

```

# maximizing for k
nu[j,i] <- value
# argmaximizing for k
zz[j,i] <- k
}
}
}
}
}

# mlp = most likely path
mlp <- numeric(T)
# argmax for stateSeq[,T]
am <- which.max(nu[,T])
mlp[T] <- zz[am,T]

# backtrace using backpointers
for (i in T:2) {
  zm <- which.max(nu[,i])
  mlp[i-1] <- zz[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 (for example, the `viterbi()` function from the `HMM` and `depmixS4` packages), and call it from NIMBLE as we have seen in Section 2.4.2. The difficulty is that HMM for capture-recapture data have specific features that make standard functions not adapted and requires coding your own Viterbi function. In particular, we have to deal with detection at first encounter, which is not estimated but is always one because an individual has to be captured to be marked and released for the first time. Also, our transition and observation matrices are not always homogeneous and may depend on time.

Let's test our `getViterbi()` function with our previous example. Remember animal #15 has the encounter history (2, 1, 1, 1, 1) which was

generated from the sequence of states (1, 1, 2, 2, 2). 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, despite incorrectly decoding a death in the second winter while individual #15 only dies in the third winter. We obtained the same results by implementing the Viterbi algorithm by hand in Section 3.10.1.

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

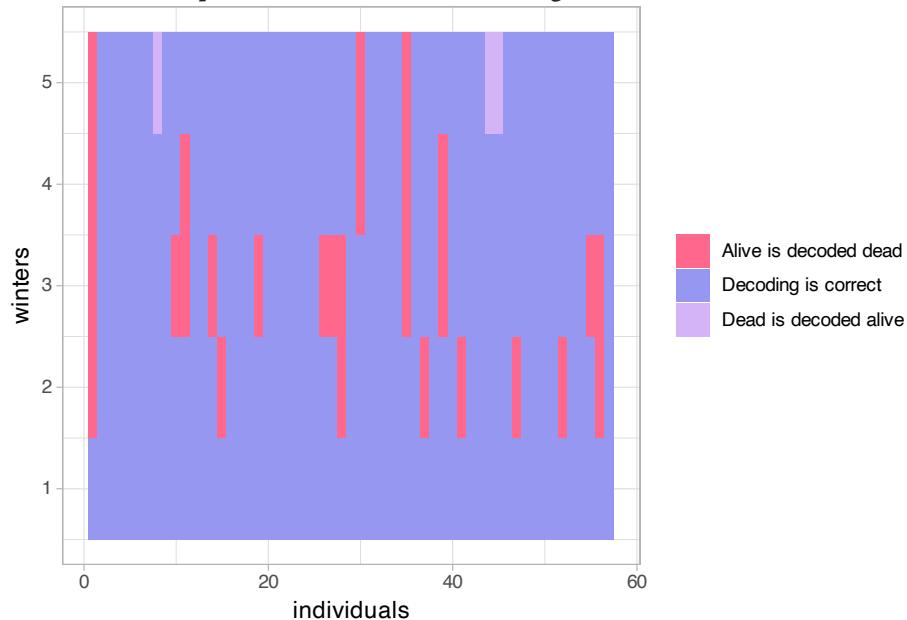
First option is to apply Viterbi to each MCMC sample, then to compute median of the MCMC Viterbi paths for each observed sequence:

```

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 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 turn the R function `getViterbi()` into a NIMBLE function and plug it in your model code to apply Viterbi. This would 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)
```

```

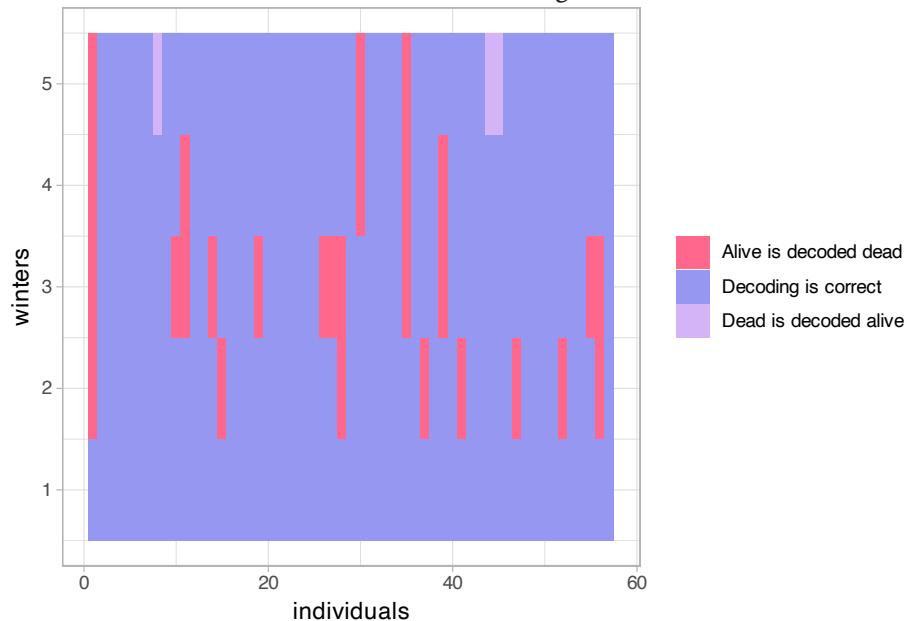
transition[2,1] <- 0                      # Pr(dead t -> alive t+1)
transition[2,2] <- 1                      # Pr(dead t -> dead t+1)

# 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 we simulated and used to generate the observations:



The results are very similar to those we obtained in Section 3.10.3, and Figure ?? is indistinguishable from Figure ??.

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

- A landmark paper on HMM is Rabiner [1989].
- Check out Jurafsky and Martin [2023] for a nice introduction to

HMM and [Zucchini et al. \[2016\]](#) for an excellent book that covers theory and applications.

- The paper by [McClintock et al. \[2020\]](#) reviews the applications of HMM in ecology.
- The package `nimbleEcology` is developed by [Goldstein et al. \[2021\]](#) and [Ponisio et al. \[2020\]](#) for application to occupancy and N-mixture models.

Part II

Transitions



Introduction

WORK IN PROGRESS

This second part `Transitions` will teach you all about capture-recapture models for open populations, with reproducible R code to ease the learning process.



4

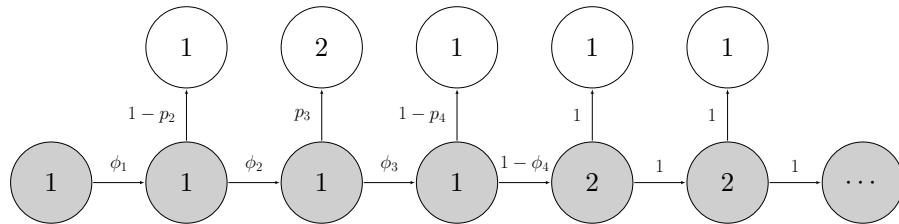
Alive and dead

4.1 Introduction

In this fourth chapter, you will learn about the Cormack-Jolly-Seber model that allows estimating survival based on capture-recapture data. You will also see how to deal with covariates to try and explain temporal and/or individual variation in survival. This chapter will also be the opportunity to introduce tools to compare models and assess their quality of fit to data.

4.2 The Cormack-Jolly-Seber (CJS) model

In Chapter 3, we introduced a capture-recapture model with constant survival and detection probabilities which we formulated as a HMM and fitted to data in NIMBLE. Historically, however, it was a slightly more complicated model that was first proposed – the so-called Cormack-Jolly-Seber (CJS) model – in which survival and recapture probabilities are time-varying. This feature of the CJS model is useful to account for variation due to environmental conditions in survival or to sampling effort in detection. Schematically the CJS model can be represented this way:



Note that the states (in gray) and the observations (in white) do not change. We still have $z = 1$ for alive, $z = 2$ for dead, $y = 1$ for non-detected, and $y = 2$ for detected.

Parameters are now indexed by time. The survival probability is defined as the probability of staying alive (or “ah, ha, ha, ha, stayin’ alive” like the Bee Gees would say) over the interval between t and $t + 1$, that is $\phi_t = \Pr(z_{t+1} = 1 | z_t = 1)$. The detection probability is defined as the probability of being observed at t given you’re alive at t , that is $p_t = \Pr(y_t = 1 | z_t = 1)$. It is important to bear in mind for later (see Section 4.8) that survival operates over an interval while detection occurs at a specific time.

The CJS model is named after the three statisticians – Richard Cormack, George Jolly and George Seber – who each published independently a paper introducing more or less the same approach, a year apart! In fact, Richard Cormack and George Jolly were working in the same corridor in Scotland back in the 1960’s. They would meet every day at coffee and would play some game together, but never mention work and were not aware of each other’s work.

4.3 Capture-recapture data

Before we turn to fitting the CJS model to actual data, let’s talk about capture-recapture for a minute. We said in Section 3.6.1 that animals are individually marked. This can be accomplished in two ways, either with artificial marks like rings for birds or ear tags for mammals, or (non-invasive) natural marks like coat patterns or feces DNA sequencing (Figure 4.1).

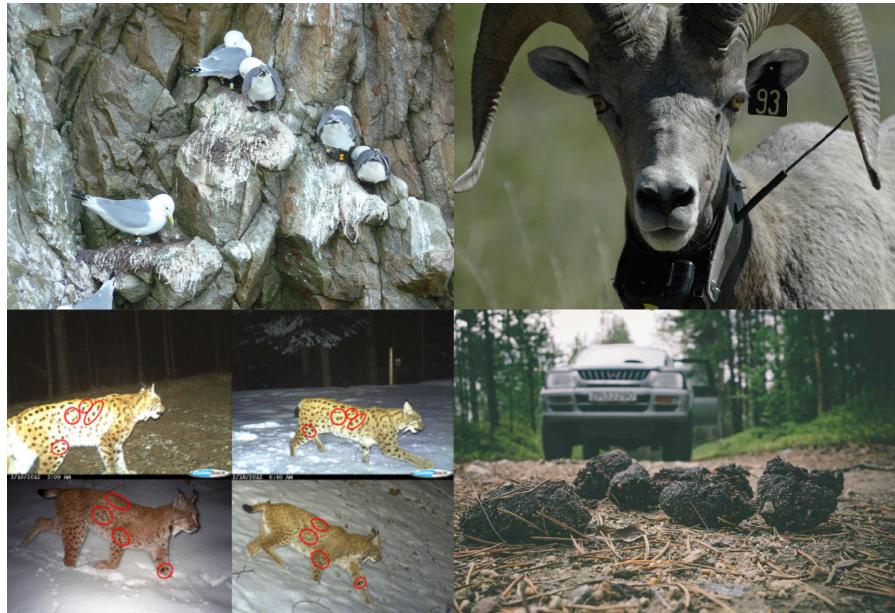


FIGURE 4.1: Animal individual marking. Top-left: rings (credit: Emanuelle Cam and Jean-Yves Monat); Top-right: ear-tags (credit: Kelly Powell); Bottom left: coat patterns (credit: Fridolin Zimmermann); Bottom right: ADN feces (credit: Alexander Kopatz)

Throughout this chapter, we will use data on the White-throated Dipper (*Cinclus cinclus*; dipper hereafter) kindly provided by Gilbert Marzolin (Figure 4.2). In total, 294 dippers with known sex and wing length were captured and recaptured between 1981 and 1987 during the March-June period. Birds were at least 1 year old when initially banded.

The data look like:

```
dipper <- read_csv("dat/dipper.csv")
dipper
## # A tibble: 294 x 9
##   year_1981 year_1982 year_1983 year_1984 year_1985
##       <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1         1         1         1         1         1
```



FIGURE 4.2: White-throated Dipper (*Cinclus cinclus*). Credit: Gilbert Marzolin.

```

##  2      1      1      1      1      1
##  3      1      1      1      1      0
##  4      1      1      1      1      0
##  5      1      1      0      1      1
##  6      1      1      0      0      0
##  7      1      1      0      0      0
##  8      1      1      0      0      0
##  9      1      1      0      0      0
## 10     1      1      0      0      0
## # i 284 more rows
## # i 4 more variables: year_1986 <dbl>, year_1987 <dbl>,
## #   sex <chr>, wing_length <dbl>

```

The first seven columns are years in which Gilbert went on the field and captured the birds. A 0 stands for a non-detection, and a 1 for a detection. The eighth column informs on the sex of the bird, with F for

female and M for male. The last column gives a measure wing length the first time a bird was captured.

4.4 Fitting the CJS model to the dipper data with NIMBLE

To write the NIMBLE code corresponding to the CJS model, we only need to make a few adjustments to the NIMBLE code for the model with constant parameters from Section 3.7. The main modification concerns the observation and transition matrices which we need to make time-varying. These matrices therefore become arrays and inherit a third dimension for time, besides that for rows and columns. Also we need priors for all time-varying survival $\phi_i[t] \sim \text{dunif}(0, 1)$ and detection $p_i[t] \sim \text{dunif}(0, 1)$ probabilities. We write:

```
...
# parameters
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)
}
...
```

The likelihood does not change, except that the time-varying observation and transition matrices need to be used appropriately. Also,

because we now deal with several cohorts of animals first captured, marked and released each year (in contrast with a single cohort as in Chapter 3), we need to start the loop over time from the first capture for each individual. Therefore, we write:

```
...
# 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])
  }
}
...
...
```

At first capture, all individuals are alive $z[i, \text{first}[i]] \sim \text{dcat}(\delta[1:2])$ and detection is 1, then after first capture for $(j \in (\text{first}[i]+1):T)$ we apply the transition and observation matrices. Overall, the code looks like:

```
hmm.phitpt <- nimbleCode({
  # parameters
  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)
  }
}
})

```

We extract the detections and non-detections from the data:

```

y <- dipper %>%
  select(year_1981:year_1987) %>%
  as.matrix()

```

We get the occasion of first capture for all individuals, by finding the position of detections in the encounter history (`which(x !=0)`), and keeping the first one:

```
first <- apply(y, 1, function(x) min(which(x !=0)))
```

Now we specify the constants:

```

my.constants <- list(N = nrow(y),      # number of animals
                      T = ncol(y),    # number of sampling occasions
                      first = first) # first capture for all animals

```

We then put the data in a list. We add 1 to the data to code non-detections as 1's detections as 2's (see Section 3.7).

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

Let's write a function for the initial values. For the latent states, we go the easy way, and say that all individuals are alive through the study period:

```

zinit <- y + 1 # non-detection -> alive
zinit[zinit == 2] <- 1 # dead -> alive

initial.values <- function() list(phi = runif(my.constants$T-1,0,1),
                                    p = runif(my.constants$T-1,0,1),
                                    z = zinit)

```

We specify the parameters we would like to monitor, survival and detection probabilities here:

```
parameters.to.save <- c("phi", "p")
```

We provide MCMC details:

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

And we run NIMBLE:

```
niter = n.iter,
nburnin = n.burnin,
nchains = n.chains)
```

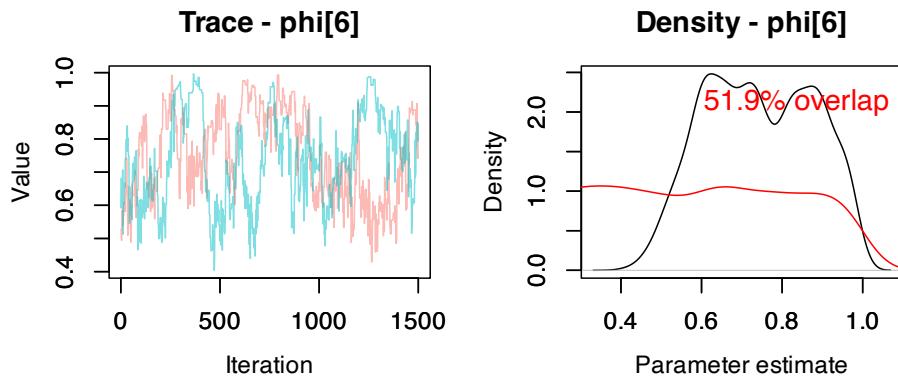
We may have a look to the numerical summaries:

```
MCMCsummary(mcmc.phitpt, params = c("phi","p"), round = 2)
##      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
```

There is not so much time variation in the detection probability which is estimated high around 0.90. Note that $p[1]$ corresponds to the detection probability in 1982 that is p_2 , $p[2]$ to detection in 1983 therefore p_3 , and so on. The dippers seem to have experienced a decrease in survival in years 1982-1983 ($\phi[2]$) and 1983-1984 ($\phi[4]$). We will get back to that in Section 4.8.

You may have noticed the small effective sample size for the last survival ($\phi[6]$) and detection ($p[6]$) probabilities. Let's have a look to mixing for parameter $\phi[6]$ for example:

```
priors <- runif(3000, 0, 1)
MCMCtrace(object = mcmc.phitpt,
           ISB = FALSE,
           exact = TRUE,
           params = c("phi[6]"),
           pdf = FALSE,
           priors = priors)
```



Clearly mixing (left panel in the plot above) is bad and there is a big overlap between the prior and the posterior for this parameter (right panel) suggesting that its prior was not well updated with the data. What is going on? If you could inspect the likelihood of the CJS model, you would realize that these two parameters ϕ_6 and p_7 appear only as the product $\phi_6 p_7$ and cannot be estimated separately. In other words, one of these parameters is redundant, and you'd need an extra sampling occasion to be able to disentangle them. This is not a big issue as long as you're aware of it and you do not attempt to ecologically interpret these parameters.

4.5 CJS model derivatives

Besides the model we considered with constant parameters (see Chapter 3) and the CJS model with time-varying parameters, you might

want to fit in-between or models with time variation on either detection or survival.

But I realize that we did not actually fit the model with constant parameters from Chapter 3. Let's do it. You should be familiar with the process by now:

```
# NIMBLE code
hmm.php <- 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])
    }
  }
})
# occasions of first capture
first <- apply(y, 1, function(x) min(which(x != 0)))
# constants
my.constants <- list(N = nrow(y),
                      T = ncol(y),
                      first = first)
# data
```

```

my.data <- list(y = y + 1)
# initial values
zinit <- y + 1 # non-detection -> alive
zinit[zinit == 2] <- 1 # dead -> alive
initial.values <- function() list(phi = runif(1,0,1),
                                    p = runif(1,0,1),
                                    z = zinit)
# parameters to monitor
parameters.to.save <- c("phi", "p")
# MCMC details
n.iter <- 2500
n.burnin <- 1000
n.chains <- 2
# run NIMBLE
mcmc.phip <- nimbleMCMC(code = hmm.phip,
                           constants = my.constants,
                           data = my.data,
                           inits = initial.values,
                           monitors = parameters.to.save,
                           niter = n.iter,
                           nburnin = n.burnin,
                           nchains = n.chains)
## |-----|-----|-----|-----|
## |-----|-----|-----|-----|
## |-----|-----|-----|-----|
## |-----|-----|-----|-----|
# numerical summaries
MCMCsummary(mcmc.phip, round = 2)
##      mean   sd 2.5% 50% 97.5% Rhat n.eff
## p    0.89 0.03 0.83 0.90  0.94 1.02   231
## phi 0.56 0.02 0.51 0.56  0.61 1.00   595

```

Let's now turn to the model with time-varying survival and constant detection. We modify the CJS model NIMBLE code by no longer having the observation matrix time-specific. I'm just providing the model code to save space:

```

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])
    }
  }
})

```

We obtain the following numerical summaries for parameters, confirming high detection and temporal variation in survival:

```

##          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

```

Now the model with time-varying detection and constant survival, for which the NIMBLE code has a constant over time transition matrix:

```

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])
    }
  }
})

```

Numerical summaries for the parameters are:

```

##      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
```

We note that these two models do no longer have parameter redundancy issues.

Before we move to the next section, you might ask how to fit these models with `nimbleEcology` as in Section 3.8.3. Conveniently there are specific NIMBLE functions for the marginalized likelihood of the CJS model and its derivatives. These function are named generically `dCJSxx()` where the first `x` is for survival and the second `x` is for detection and `x` can be `s` for scalar or constant or `v` for vector or time-dependent. For example, to implement the model with constant survival and detection probabilities, you would use `dCJSss()`:

```
# load nimbleEcology in case we forgot previously
library(nimbleEcology)
# data
y <- dipper %>%
  select(year_1981:year_1987) %>%
  as.matrix()
y <- y[ first != ncol(y), ] # get rid of individuals for which first detection = last capture
# NIMBLE code
hmm.phip.nimbleecology <- nimbleCode({
  phi ~ dunif(0, 1) # survival prior
  p ~ dunif(0, 1) # detection prior
  # likelihood
  for (i in 1:N){
    y[i, first[i]:T] ~ dCJS_ss(probSurvive = phi,
                                  probCapture = p,
                                  len = T - first[i] + 1)
  }
})
# occasions of first capture
first <- apply(y, 1, function(x) min(which(x != 0)))
# constants
```

```

my.constants <- list(N = nrow(y),
                      T = ncol(y),
                      first = first)

# data
my.data <- list(y = y) # format is 0 for non-detected and 1 for detected
# initial values; we use the marginalized likelihood, so no latent states
# in it, therefore no need for initial values for the latent states
initial.values <- function() list(phi = runif(1,0,1),
                                    p = runif(1,0,1))

# parameters to monitor
parameters.to.save <- c("phi", "p")

# MCMC details
n.iter <- 2500
n.burnin <- 1000
n.chains <- 2

# run NIMBLE
mcmc.phip.nimbleecology <- nimbleMCMC(code = hmm.phip.nimbleecology,
                                           constants = my.constants,
                                           data = my.data,
                                           inits = initial.values,
                                           monitors = parameters.to.save,
                                           niter = n.iter,
                                           nburnin = n.burnin,
                                           nchains = n.chains)

## |-----|-----|-----|-----|
## |-----|
## |-----|-----|-----|-----|
## |-----|
# numerical summaries
MCMCsummary(mcmc.phip.nimbleecology, round = 2)
##      mean   sd 2.5% 50% 97.5% Rhat n.eff
## p    0.89 0.03 0.84 0.90  0.94 1.02    672
## phi 0.56 0.02 0.52 0.56  0.61 1.00    716

```

We are left with four models, each saying a different story about the data, with temporal variation or not in either survival or detection

probability. How to quantify the most plausible of these four ecological hypotheses? Rendez-vous in the next section.

4.6 Model comparison with WAIC

Which of the four models above is best supported by the data? To answer this question, we need to bear in mind that we used all the observed data to fit these models, and how close to the truth these models will perform when predicting for future data – also known as predictive accuracy – should be assessed. A natural candidate measure for predictive accuracy is the likelihood which is often referred in the context of model comparison as the predictive density. However, we know neither the true process, nor the future data, and we can only estimate the predictive density with some bias.

You may have heard about the Akaike Information Criterion (AIC) in the frequentist framework, and the Deviance Information Criterion (DIC) in the Bayesian framework. Here, we will also consider the Widely Applicable Information Criterion or Watanabe Information Criterion (WAIC). AIC, DIC and WAIC each aim to provide an approximation of predictive accuracy.

While AIC is the predictive measure of choice in the frequentist framework for ecologists, DIC has been around for some time for Bayesian applications due to its availability in population BUGS pieces of software. However, AIC and BIC use a point estimate of the unknown parameters, while we have access to their entire (posterior) distribution with the Bayesian approach. Also, DIC has been shown to misbehave when the posterior distribution is not well summarized by its mean. For a more fully Bayesian approach we would like to use the entire posterior distribution to evaluate the predictive performance, which is exactly what WAIC does.

Conveniently, NIMBLE calculates WAIC for you. The only modification you need to make is to add `WAIC = TRUE` in the call to the `nimbleMCMC()` function. For example, for the CJS model, we write:

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

I re-ran the four models to calculate the WAIC value for each of them

```
data.frame(model = c("both survival & detection constant",
                     "time-dependent survival, constant detection",
                     "constant survival, time-dependent detection",
                     "both survival & detection time-dependent"),
           WAIC = c(mcmc.phip$WAIC$WAIC,
                     mcmc.phitp$WAIC$WAIC,
                     mcmc.phipt$WAIC$WAIC,
                     mcmc.phiptpt$WAIC$WAIC))
```

	model	WAIC
## 1	both survival & detection constant	265.9
## 2	time-dependent survival, constant detection	277.6
## 3	constant survival, time-dependent detection	270.2
## 4	both survival & detection time-dependent	308.8

Lower values of WAIC imply higher predictive accuracy, therefore we would favor model with constant parameters.

4.7 Goodness of fit

In the previous section, Section 4.6, we compared models between each other based on their predictive accuracy – we assessed their *relative* fit. However, even though we were able to rank these models according to predictive accuracy, it could happen that all models actually had poor predictive performance – this has to do with *absolute* fit.

How to assess the goodness of fit of the CJS model to capture-recapture data?

4.7.1 Posterior predictive checks

In the Bayesian framework, we rely on posterior predictive checks to assess absolute fit. Briefly speaking, the idea is to compare the observed data to replicated data generated from the model. If the model is a good fit to the data, then the replicated data predicted from the model should look similar to the observed data. To simplify the comparison, some summary statistics are generally used. For the CJS model, we would use the so-called m-array which gathers the number of individuals released in a year and first recaptured in other years. I illustrate the use of posterior predictive checks in Section 7.6.

4.7.2 Classical tests

In the frequentist literature, there are well established procedures for assessing absolute fit and departures from specific CJS model assumptions, and it would be a shame to just ignore them.

We focus on two such assumptions that have an ecological interpretation, transience and trap-dependence. The transience procedure assesses whether newly encountered individuals have the same chance to be later detected as previously encountered individuals. Transience is often seen as an excess of individuals never seen again. The trap-dependence procedure assesses whether non-detected individuals have the same chance to be encountered at the next occasion as de-

tected individuals. Trap-dependence is often seen as an effect of trapping on detection. Although these procedures are called *test of transience* and *test of trap-dependence*, when it comes to interpretation, you should keep in mind that transience and trap-dependence are just two specific reasons why the tests might detect a lack of fit.

These tests are implemented in the package `R2ucare`, and we illustrate its use with the dipper data.

We load the package `R2ucare`:

```
library(R2ucare)
```

We get the capture-recapture data:

```
# capture-recapture data
dipper <- read_csv("dat/dipper.csv")
# get encounter histories
y <- dipper %>%
  select(year_1981:year_1987) %>%
  as.matrix()
# number of birds with a particular capture-recapture history
size <- rep(1, nrow(y))
```

We may perform a test specifically to assess a transient effect:

```
test3sr(y, size)
## $test3sr
##      stat       df     p_val sign_test
##      1.773     5.000    0.880   -0.054
##
## $details
##   component stat p_val signed_test test_perf
## 1           2  0.08  0.778      0.283 Chi-square
```

```

## 2      3 0.232 0.63      0.482 Chi-square
## 3      4 0.847 0.358     -0.92 Chi-square
## 4      5 0.288 0.592     -0.537 Chi-square
## 5      6 0.326 0.568      0.571 Chi-square

```

Or trap-dependence:

```

test2ct(y, size)
## $test2ct
##   stat      df    p_val sign_test
##   9.463    4.000  0.051   -1.538
##
## $details
##   component dof  stat p_val signed_test test_perf
## 1          2   1    0    1            0 Fisher
## 2          3   1    0    1            0 Fisher
## 3          4   1    0    1            0 Fisher
## 4          5   1 9.463  0.002     -3.076 Fisher

```

None of these tests is significant, but what to do if these tests are significant? Transience may occur when animals transit through but do not belong to the study population (true transients), or reproduce only once then die or permanently disperse (cost of first reproduction), or die or permanently disperse due to an effect of marking (marking effect). These transient individuals are never detected again after their initial capture, which means they have a zero probability of local survival after this initial capture. This suggests a way to account for the transience issue that we cover in a case study (see Section 7.2).

Trap-dependence may occur when animals are affected (positively or negatively) by trapping (true trap-dependence), when observers tend to visit some parts of the study area more often when individuals have been detected (preferential sampling), when some patches of a heterogeneous habitat are more accessible so that individuals stationed there have higher detection probabilities (access bias), when age, sex

or social status are unknown, but determine individual movements or activity patterns so that the susceptibility to be detected varies (individual heterogeneity), or when non random temporary emigration occurs (skipped reproduction). Trap-dependence therefore designates some correlation between detection events. To account for the trap-dependence issue, this correlation needs to be accounted for, and we show how to do just that in a case study (see Section 7.3).

4.7.3 Design considerations

So far, we have addressed assumptions relative to the model. There are also assumptions relative to the design. In particular, survival refers to the study area, and so we need to think carefully about what survival does actually mean. We actually estimate what we usually call *apparent* survival, not exactly *true* survival. Apparent survival probability is the product of true survival and study area fidelity. Consequently, apparent survival is always lower than true survival unless study area fidelity is exactly 1.

There are other assumptions relative to the design, which we simply list here. There should be no mark loss, the identity of individuals should be recorded without error (no false positives), and captured animals should be a random sample of the population.

4.8 Covariates

In the models we have considered so far, survival and detection probabilities may vary over time, but we do not include ecological drivers that might explain these variation. Luckily, in the spirit of generalized linear models, we can make these parameters dependent on external covariates over time, such as environmental conditions for survival or sampling effort for detection.

Besides variation over time, we will also cover individual variation in

these parameters, in which for example survival vary according to sex or some phenotypic characteristics (e.g. size or body mass).

Let's illustrate the use of covariates with the dipper example.

4.8.1 Temporal covariates

4.8.1.1 Discrete

A major flood occurred during the 1983 breeding season (from October 1982 to May 1983). Because captures during the breeding season occurred before and after the flood, survival in the two years 1982-1983 and 1983-1984 were likely to be affected. Indeed survival of a species living along and feeding in the river in these two flood years was most likely lower than in nonflood years. Here we will use a discrete or categorical covariate, or a group.

Let's use a covariate `flood` that contains 1s and 2s, indicating whether we are in a flood or nonflood year for each year: 1 if nonflood year, and 2 if flood year.

```
flood <- c(1, # 1981-1982 (nonflood)
         2, # 1982-1983 (flood)
         2, # 1983-1984 (flood)
         1, # 1984-1985 (nonflood)
         1, # 1985-1986 (nonflood)
         1) # 1986-1987 (nonflood)
```

Then we write the model code:

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

```

We use nested indexing above when specifying survival in the transition matrix. E.g. for year $t = 2$, `phi[flood[t]]` gives `phi[flood[2]]` which will be `phi[2]` because `flood[2]` is 2 that (a flood year).

Let's provide the constants in a list:

```

my.constants <- list(N = nrow(y),
                      T = ncol(y),
                      first = first,
                      flood = flood)

```

Now a function to generate initial values:

```
initial.values <- function() list(phi = runif(2,0,1),  
                                p = runif(1,0,1),  
                                z = zinits)
```

The parameters to be monitored:

```
parameters.to.save <- c("p", "phi")
```

We're all set, and we run NIMBLE:

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

The numerical summaries are given by:

```
MCMCsummary(mcmc.phifloodp, round = 2)
```

```
##          mean    sd 2.5% 50% 97.5% Rhat n.eff  
## p      0.89 0.03 0.83 0.90  0.95     1   609  
## phi[1] 0.61 0.03 0.55 0.61  0.67     1  1474  
## phi[2] 0.47 0.04 0.38 0.47  0.56     1  1700
```

Having a look to the numerical summaries, we see that as expected, survival in flood years ($\text{phi}[2]$) is much lower than survival in non-flood years ($\text{phi}[1]$). You could formally test this difference by considering

the difference `phi[1] - phi[2]`. Alternatively, this can be done afterwards and calculating the probability that this difference is positive (or `phi[1] > phi[2]`). Using a single chain for convenience, we do:

```
phi1 <- mcmc.phifloodp$chain1[, 'phi[1]']
phi2 <- mcmc.phifloodp$chain1[, 'phi[2]']
mean(phi1 - phi2 > 0)
## [1] 0.9952
```

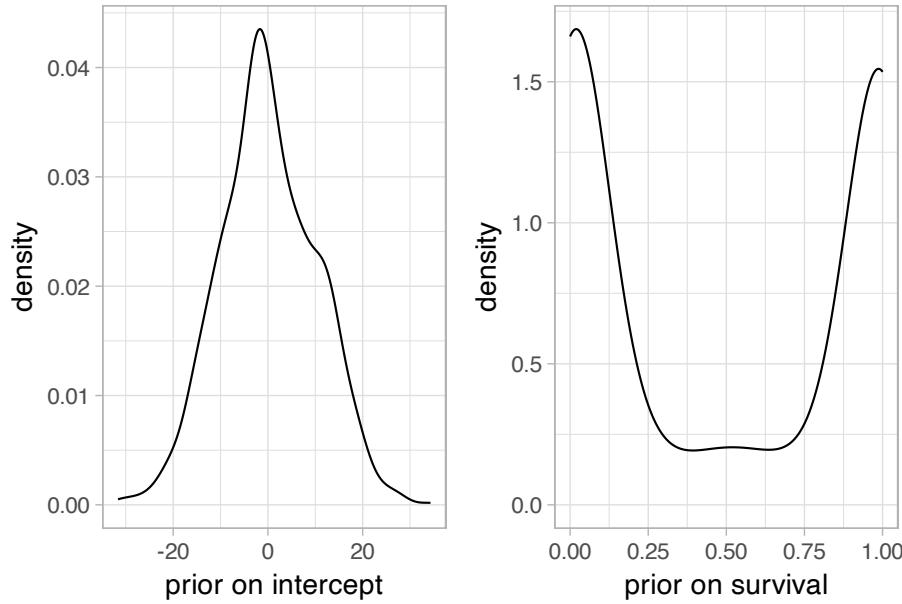
An important point here is that we formulated an ecological hypothesis which we translated into a model. The next step would consist in calculating WAIC for this model and compare it with that of the four other model we have fitted so far (see Section 4.6).

Another method to include a discrete covariate consists in considering the effect of the difference between its levels. For example, we could consider survival in nonflood years as the reference and test the difference in survival with flood years.

To do so, we write survival as a linear function of our covariate on some scale, e.g. $\text{logit}(\phi_t) = \beta_1 + \beta_2 \text{flood}_t$ where the β 's are regression coefficients we need to estimate (intercept β_1 and slope β_2), and $\text{logit}(x) = \log\left(\frac{x}{1-x}\right)$ is the logit function. The logit function lives between $-\infty$ and $+\infty$, and sends values between 0 and 1 onto the real line. For example $\log(0.2/(1-0.2)) = -1.39$, $\log(0.5/(1-0.5)) = 0$ and $\log(0.8/(1-0.8)) = 1.39$. Why use the logit function? Because survival is a probability bounded between 0 and 1, and if we used directly $\phi_t = \beta_1 + \beta_2 \text{flood}_t$ we might end up with estimates for the regression coefficients that make survival out of bound. Therefore, we consider survival as a linear function of our covariates on the scale of the logit function, working on the real line, then back-transform using the inverse-logit (reciprocal function) to obtain survival on its natural scale. The inverse-logit function is $\text{logit}^{-1}(x) = \frac{\exp(x)}{1 + \exp(x)} = \frac{1}{1 + \exp(-x)}$. The logit function is often called a link function like in generalized linear models. Other link functions exist, we'll see an example in Section 5.4.2.

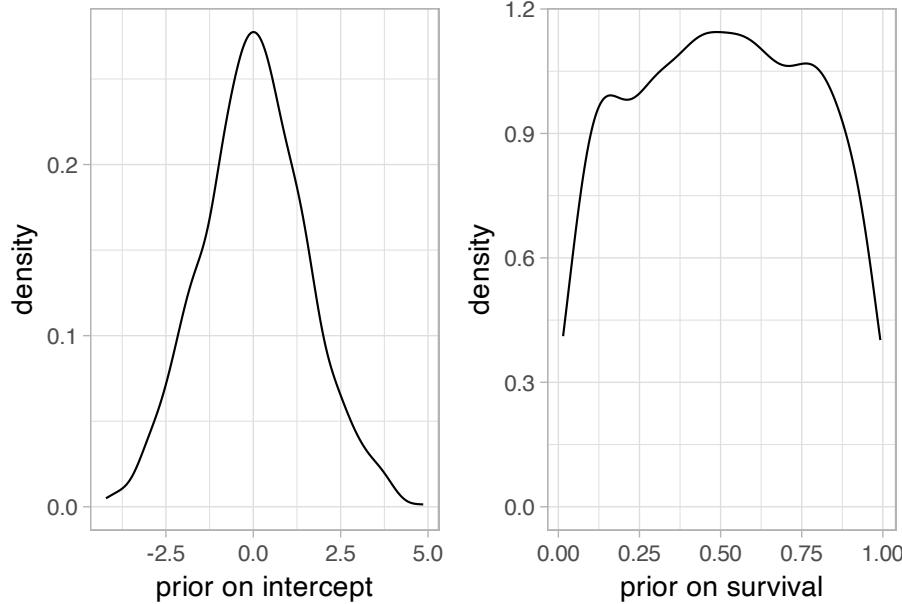
Another point of attention is the prior we will assign to regression coefficients. We no longer assign a prior to survival directly like previously, but we need to assign prior to the β 's which will induce some prior on survival. And sometimes, your priors on the regression coefficients are non-informative, but the prior on survival is not. Consider for example the case of a single intercept with no covariate. If you assign as a prior to this regression coefficient a normal distribution with mean 0 and large standard deviation (left figure below), which would be my first reflex, then you end up with a very informative prior on survival with a bathtub shape, putting much importance on low and high values (right figure below):

```
# 1000 random values from a N(0, 10)
intercept <- rnorm(1000, mean = 0, sd = 10)
# plogis() is the inverse-logit function in R
survival <- plogis(intercept)
df <- data.frame(intercept = intercept, survival = survival)
plot1 <- df %>%
  ggplot(aes(x = intercept)) +
  geom_density() +
  labs(x = "prior on intercept")
plot2 <- df %>%
  ggplot(aes(x = survival)) +
  geom_density() +
  labs(x = "prior on survival")
plot1 + plot2
```



Now if you go for a lower standard deviation for the intercept prior (left figure below), e.g. 1.5, the prior on survival is non-informative, looking like a uniform distribution between 0 and 1 (right figure below):

```
set.seed(123)
# 1000 random values from a N(0,1.5)
intercept <- rnorm(1000, mean = 0, sd = 1.5)
# plogis() is the inverse-logit function in R
survival <- plogis(intercept)
df <- data.frame(intercept = intercept, survival = survival)
plot1 <- df %>%
  ggplot(aes(x = intercept)) +
  geom_density() +
  labs(x = "prior on intercept")
plot2 <- df %>%
  ggplot(aes(x = survival)) +
  geom_density() +
  labs(x = "prior on survival")
plot1 + plot2
```



Now let's go back to our model. We first define our flood covariate with 0 if nonflood year, and 1 if flood year:

```
flood <- c(0, # 1981-1982 (nonflood)
         1, # 1982-1983 (flood)
         1, # 1983-1984 (flood)
         0, # 1984-1985 (nonflood)
         0, # 1985-1986 (nonflood)
         0) # 1986-1987 (nonflood)
```

Then we write the NIMBLE code:

```
hmm.phifloodp <- nimbleCode({
  delta[1] <- 1           # Pr(alive t = 1) = 1
  delta[2] <- 0           # Pr(dead t = 1) = 0
  for (t in 1:(T-1)){
    logit(phi[t]) <- beta[1] + beta[2] * flood[t]
    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
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)
beta[1] ~ dnorm(0, sd = 1.5) # prior intercept
beta[2] ~ dnorm(0, sd = 1.5) # prior slope
# 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])
  }
}
})

```

We wrote $\text{logit}(\phi_t) = \beta_1 + \beta_2 \text{flood}_t$, meaning that survival in non-flood years ($\text{flood}_t = 0$) is $\text{logit}(\phi_t) = \beta_1$ and survival in flood years ($\text{flood}_t = 1$) is $\text{logit}(\phi_t) = \beta_1 + \beta_2$. We see that β_1 is survival in non-flood years (on the logit scale) and β_2 is the difference between survival in flood years and survival in nonflood years (again, on the logit scale). In passing we assigned the same prior for both β_1 and β_2 but in certain situations, we might think twice before doing that as β_2 is a difference between two survival probabilities (on the logit scale).

Let's put our constants in a list:

```

my.constants <- list(N = nrow(y),
                      T = ncol(y),
                      first = first,
                      flood = flood)

```

Then our function for generating initial values:

```
initial.values <- function() list(beta = rnorm(2,0,1),  
                                p = runif(1,0,1),  
                                z = zinits)
```

And the parameters to be monitored:

```
parameters.to.save <- c("beta", "p", "phi")
```

Finaly, we run NIMBLE:

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

You may check that we get the same numerical summaries as above for survival in nonflood years (`phi[1]`, `phi[4]`, `phi[5]` and `phi[6]`) and flood years (`phi[2]` and `phi[3]`):

```
MCMCssummary(mcmc.phifloodp, round = 2)
```

```

##          mean    sd  2.5%   50% 97.5% Rhat n.eff
## beta[1]  0.44  0.13  0.17  0.44  0.69 1.01    725
## beta[2] -0.54  0.22 -0.96 -0.54 -0.11 1.00    770
## p        0.89  0.03  0.83  0.89  0.94 1.00    631
## phi[1]   0.61  0.03  0.54  0.61  0.67 1.01    724
## phi[2]   0.47  0.04  0.39  0.47  0.56 1.00   1597
## phi[3]   0.47  0.04  0.39  0.47  0.56 1.00   1597
## phi[4]   0.61  0.03  0.54  0.61  0.67 1.01    724
## phi[5]   0.61  0.03  0.54  0.61  0.67 1.01    724
## phi[6]   0.61  0.03  0.54  0.61  0.67 1.01    724

```

You may also check how to go from the β 's to the survival probabilities ϕ . Let's get the draws from the posterior distribution of the β 's first:

```

beta1 <- c(mcmc.phifloodp$chain1[, 'beta[1]'], # beta1 chain 1
            mcmc.phifloodp$chain2[, 'beta[1]']) # beta1 chain 2
beta2 <- c(mcmc.phifloodp$chain1[, 'beta[2]'], # beta2 chain 1
            mcmc.phifloodp$chain2[, 'beta[2]']) # beta2 chain 2

```

Then apply the inverse-logit function to get survival in nonflood years, e.g. its posterior mean and credible interval:

```

mean(plogis(beta1))
## [1] 0.6066
quantile(plogis(beta1), probs = c(2.5, 97.5)/100)
##    2.5% 97.5%
## 0.5435 0.6651

```

Same thing for survival in flood years:

```

mean(plogis(beta1 + beta2))
## [1] 0.4744
quantile(plogis(beta1 + beta2), probs = c(2.5, 97.5)/100)
##    2.5% 97.5%
## 0.3895 0.5606

```

4.8.1.2 Continuous

Instead of a discrete covariate varying over time, we may want to consider a continuous covariate, say x_t , through $\text{logit}(\phi_t) = \beta_1 + \beta_2 x_t$. For example, let's investigate the effect of water flow on dipper survival, which should reflect the flood that occurred during the 1983 breeding season.

We build a covariate with water flow in liters per second measured during the March to May period each year, starting with year 1982:

```
# water flow in L/s
water_flow <- c(443, # March-May 1982
               1114, # March-May 1983
               529, # March-May 1984
               434, # March-May 1985
               627, # March-May 1986
               466) # March-May 1987
```

We do not need water flow in 1981 because we will write the probability ϕ_t of being alive in year $t + 1$ given a bird was alive in year t as a linear function of the water flow in year $t + 1$.

You may have noticed the high value of water flow for 1983, twice as much as in the other years, corresponding to the flood. Importantly, we standardize our covariate to improve convergence:

```
water_flow_st <- (water_flow - mean(water_flow)) / sd(water_flow)
```

Now we write the model code:

```
hmm.phiflowp <- nimbleCode({
  delta[1] <- 1 # Pr(alive t = 1) = 1
  delta[2] <- 0 # Pr(dead t = 1) = 0
```

```

for (t in 1:(T-1)){
  logit(phi[t]) <- beta[1] + beta[2] * flow[t]
  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
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)
beta[1] ~ dnorm(0, 1.5) # prior intercept
beta[2] ~ dnorm(0, 1.5) # prior slope
# 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)
  }
}
})

```

We put the constants in a list:

```

my.constants <- list(N = nrow(y),
                      T = ncol(y),
                      first = first,
                      flow = water_flow_st)

```

Initial values as usual:

```
initial.values <- function() list(beta = rnorm(2,0,1),  
                                    p = runif(1,0,1),  
                                    z = zinits)
```

And parameters to be monitored:

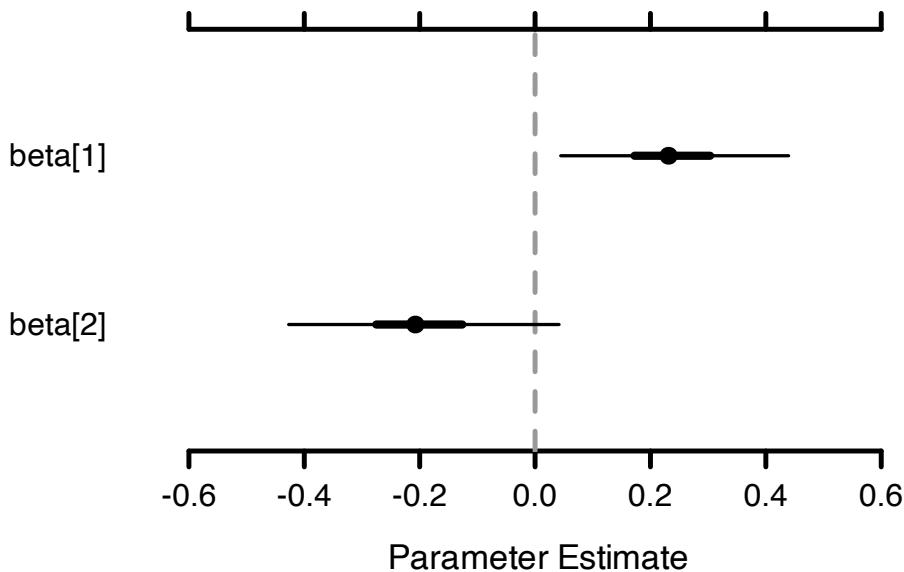
```
parameters.to.save <- c("beta", "p", "phi")
```

Eventually, we run NIMBLE:

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

We can have a look to the results through a caterpillar plot of the regression parameters:

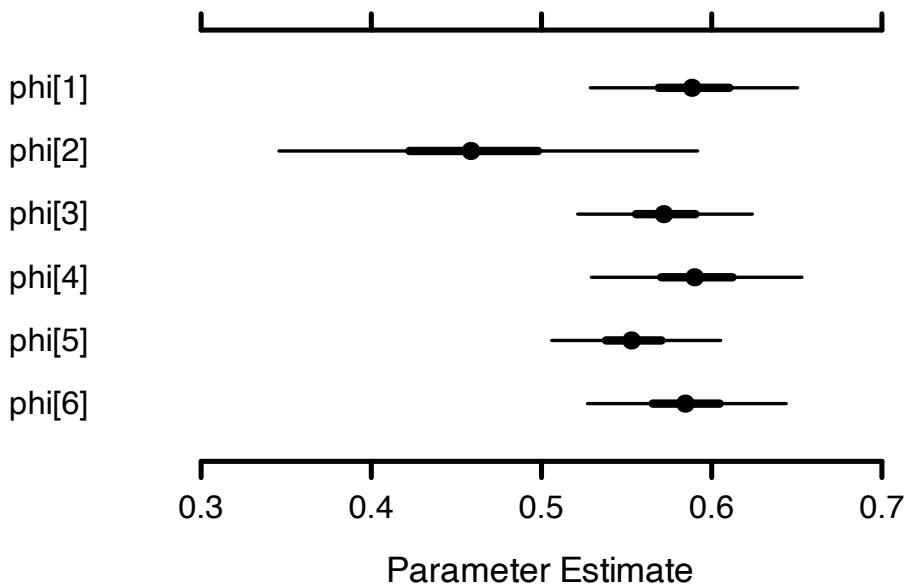
```
MCMCplot(object = mcmc.phiflowp, params = "beta")
```



The posterior distribution of the slope (β_2) is centered on negative values, suggesting that as water flow increases, survival decreases.

Let's inspect the time-dependent survival probability:

```
MCMCplot(object = mcmc.phiflowp, params = "phi", ISB = TRUE)
```



Survival between 1982 and 1983 ($\phi[2]$) was greatly affected and much lower than on average. This decrease corresponds to the high water flow in 1983 and the flood. These results are in line with our previous findings obtained by considering a discrete covariate for nonflood vs. flood years.

4.8.2 Individual covariates

In the previous section, we learnt how to explain temporal heterogeneity in survival and detection. Heterogeneity could also originate from individual differences between animals. You may think of a difference in survival between males and females for a discrete covariate example, or size and body mass for examples of a continuous covariate. Let's illustrate both discrete and continuous covariates on the dipper.

4.8.2.1 Discrete

We first consider a covariate `sex` that contains 1's and 2's indicating the sex of each bird: 1 if male, and 2 if female. We implement the model with sex effect using nested indexing, similarly to the model with flood vs. nonflood years. The section of the NIMBLE code that needs to be amended is:

```
...
for (i in 1:N){
  gamma[1,1,i] <- phi[sex[i]]      # Pr(alive t -> alive t+1)
  gamma[1,2,i] <- 1 - phi[sex[i]]   # Pr(alive t -> dead t+1)
  gamma[2,1,i] <- 0                # Pr(dead t -> alive t+1)
  gamma[2,2,i] <- 1                # Pr(dead t -> dead t+1)
}
phi[1] ~ dunif(0,1) # male survival
phi[2] ~ dunif(0,1) # female survival
...
```

After running NIMBLE, we get:

```
MCMCsummary(object = mcmc.phisexp.ni, round = 2)
```

```
##          mean    sd 2.5% 50% 97.5% Rhat n.eff
## p      0.90 0.03 0.84 0.90 0.95     1   643
## phi[1] 0.57 0.03 0.50 0.57 0.64     1 1668
## phi[2] 0.55 0.03 0.48 0.55 0.62     1 1482
```

Male survival (`phi[1]`) looks very similar to female survival (`phi[2]`).

4.8.2.2 Continuous

Besides discrete individual covariates, you might want to have continuous individual covariates, e.g. wing length in the dipper example. Note that we're considering an individual trait that takes the same value whatever the occasion. We consider wing length here, and more precisely its measurement at first detection. We first standardize the covariate:

```
wing.length.st <- as.vector(scale(dipper$wing_length))
head(wing.length.st)
## [1] 0.7581 -0.8671  0.5260 -1.5637 -1.3315  1.2225
```

Now we write the model:

```
hmm.phiwlp <- nimbleCode({
  p ~ dunif(0, 1) # prior detection
  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){
    logit(phi[i]) <- beta[1] + beta[2] * winglength[i]
    gamma[1,1,i] <- phi[i]      # Pr(alive t -> alive t+1)
```

```

gamma[1,2,i] <- 1 - phi[i] # Pr(alive t -> dead t+1)
gamma[2,1,i] <- 0           # Pr(dead t -> alive t+1)
gamma[2,2,i] <- 1           # Pr(dead t -> dead t+1)
}
beta[1] ~ dnorm(mean = 0, sd = 1.5)
beta[2] ~ dnorm(mean = 0, sd = 1.5)
delta[1] <- 1               # Pr(alive t = 1) = 1
delta[2] <- 0               # Pr(dead t = 1) = 0
# 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, i])
    y[i,j] ~ dcat(omega[z[i,j], 1:2])
  }
}
})

```

We put the constants in a list:

```

my.constants <- list(N = nrow(y),
                      T = ncol(y),
                      first = first,
                      winglength = wing.length.st)

```

We write a function for generating initial values:

```

initial.values <- function() list(beta = rnorm(2,0,1),
                                    p = runif(1,0,1),
                                    z = zinits)

```

And we run NIMBLE:

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

Let's inspect the numerical summaries for the regression parameters:

```
MCMCssummary(mcmc.phiwlps, params = "beta", round = 2)
```

```
##          mean     sd  2.5%   50% 97.5% Rhat n.eff
## beta[1]  0.25 0.10  0.04  0.25  0.45     1 1472
## beta[2] -0.02 0.09 -0.20 -0.02  0.17     1 1555
```

Wing length does not seem to explain much individual-to-individual variation in survival – the posterior distribution of the slope (`beta[2]`) is centered on 0 as we can see from the credible interval.

Let's plot the relationship between survival and wing length. First, we gather the values generated from the posterior distribution of the regression parameters in the two chains:

```
beta1 <- c(mcmc.phiwlps$chain1[, 'beta[1']], # intercept, chain 1
            mcmc.phiwlps$chain2[, 'beta[1']]) # intercept, chain 2
beta2 <- c(mcmc.phiwlps$chain1[, 'beta[2']], # slope, chain 1
            mcmc.phiwlps$chain2[, 'beta[2']]) # slope, chain 2
```

Then we define a grid of values for wing length, and predict survival for each MCMC iteration:

```

predicted_survival <- matrix(NA,
                             nrow = length(beta1),
                             ncol = length(my.constants$winglength))

for (i in 1:length(beta1)){
  for (j in 1:length(my.constants$winglength)){
    predicted_survival[i,j] <- plogis(beta1[i] +
                                         beta2[i] * my.constants$winglength[j])
  }
}

```

Now we calculate posterior mean and the credible interval (note the ordering):

```

mean_survival <- apply(predicted_survival, 2, mean)
lci <- apply(predicted_survival, 2, quantile, prob = 2.5/100)
uci <- apply(predicted_survival, 2, quantile, prob = 97.5/100)
ord <- order(my.constants$winglength)

df <- data.frame(wing_length = my.constants$winglength[ord],
                  survival = mean_survival[ord],
                  lci = lci[ord],
                  uci = uci[ord])

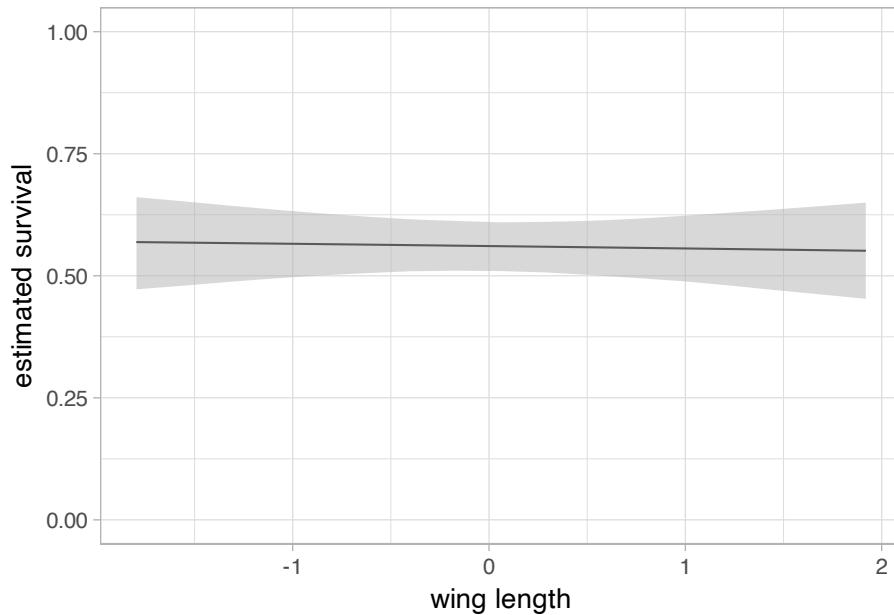
```

Now time to visualize:

```

df %>%
  ggplot() +
  aes(x = wing_length, y = survival) +
  geom_line() +
  geom_ribbon(aes(ymin = lci, ymax = uci),
              fill = "grey70",
              alpha = 0.5) +
  ylim(0,1) +
  labs(x = "wing length", y = "estimated survival")

```



The flat relationship between survival and wing length is confirmed.

4.8.3 Several covariates

You may wish to have an effect of both sex and wing length in a model. Let's consider an additive effect of both covariates. We use a covariate `sex` that takes value 0 if male, and 1 if female. We have $\text{logit}(\phi_i) = \beta_1 + \beta_2 \text{sex}_i + \beta_3 \text{winglength}_i$ for bird i , so that male survival is $\beta_1 + \beta_3 \text{winglength}_i$ and female survival is $\beta_1 + \beta_2 + \beta_3 \text{winglength}_i$ (both on the logit scale). The relationship between survival and wing length is parallel between males and females, on the logit scale, and the gap between the two is measured by β_2 (hence the term *additive effect*).

The NIMBLE code is:

```
hmm.phisexwlp <- nimbleCode({
  p ~ dunif(0, 1) # prior detection
  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){
  logit(phi[i]) <- beta[1] + beta[2] * sex[i] + beta[3] * winglength[i]
  gamma[1,1,i] <- phi[i]      # Pr(alive t -> alive t+1)
  gamma[1,2,i] <- 1 - phi[i]  # Pr(alive t -> dead t+1)
  gamma[2,1,i] <- 0          # Pr(dead t -> alive t+1)
  gamma[2,2,i] <- 1          # Pr(dead t -> dead t+1)
}
beta[1] ~ dnorm(mean = 0, sd = 1.5) # intercept male
beta[2] ~ dnorm(mean = 0, sd = 1.5) # difference bw male and female
beta[3] ~ dnorm(mean = 0, sd = 1.5) # slope wing length
delta[1] <- 1                  # Pr(alive t = 1) = 1
delta[2] <- 0                  # Pr(dead t = 1) = 0
# 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, i])
    y[i,j] ~ dcat(omega[z[i,j], 1:2])
  }
}
})

```

We put constants and data in lists:

```

first <- apply(y, 1, function(x) min(which(x != 0)))
wing.length.st <- as.vector(scale(dipper$wing_length))
my.constants <- list(N = nrow(y),
                      T = ncol(y),
                      first = first,
                      winglength = wing.length.st,
                      sex = ifelse(dipper$sex == "M", 0, 1))
my.data <- list(y = y + 1)

```

We write a function to generate initial values:

```

zinits <- y
zinits[zinits == 0] <- 1
initial.values <- function() list(beta = rnorm(3,0,2),
                                    p = runif(1,0,1),
                                    z = zinits)

```

We specify the parameters to be monitored:

```
parameters.to.save <- c("beta", "p")
```

And now we run NIMBLE:

```

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

```

Let's display numerical summaries for all parameters:

```
MCMCsummary(mcmc.phisexwlp, round = 2)
```

	mean	sd	2.5%	50%	97.5%	Rhat	n.eff
## beta[1]	0.47	0.24	-0.02	0.48	0.94	1.01	102
## beta[2]	-0.43	0.43	-1.28	-0.44	0.46	1.01	85
## beta[3]	-0.19	0.20	-0.60	-0.19	0.22	1.00	106
## p	0.89	0.03	0.83	0.90	0.94	1.01	643

The slope `beta[3]` is the same for both males and females. Although its posterior mean is negative, its credible interval suggests that its posterior distribution largely encompasses 0, therefore a very weak signal, if any.

Let's visualize survival as a function of wing length for both sexes. First we put together the values from the two chains we generated in the posterior distributions of the regression parameters:

```
beta1 <- c(mcmc.phisexlp$chain1[, 'beta[1]'], # beta1 chain 1
            mcmc.phisexlp$chain2[, 'beta[1]']) # beta1 chain 2
beta2 <- c(mcmc.phisexlp$chain1[, 'beta[2]'], # beta2 chain 1
            mcmc.phisexlp$chain2[, 'beta[2]']) # beta2 chain 2
beta3 <- c(mcmc.phisexlp$chain1[, 'beta[3]'], # beta3 chain 1
            mcmc.phisexlp$chain2[, 'beta[3]']) # beta3 chain 2
```

We get survival estimates for each MCMC iteration:

```
predicted_survivalM <- matrix(NA, nrow = length(beta1),
                                ncol = length(my.constants$winglength))
predicted_survivalF <- matrix(NA, nrow = length(beta1),
                                ncol = length(my.constants$winglength))

for (i in 1:length(beta1)){
  for (j in 1:length(my.constants$winglength)){
    predicted_survivalM[i,j] <- plogis(beta1[i] +
                                         beta3[i] * my.constants$winglength[j])
    predicted_survivalF[i,j] <- plogis(beta1[i] +
                                         beta2[i] +
                                         beta3[i] * my.constants$winglength[j])
  }
}
```

From here, we may calculate posterior mean and credible intervals:

```

mean_survivalM <- apply(predicted_survivalM, 2, mean)
lciM <- apply(predicted_survivalM, 2, quantile, prob = 2.5/100)
uciM <- apply(predicted_survivalM, 2, quantile, prob = 97.5/100)
mean_survivalF <- apply(predicted_survivalF, 2, mean)
lciF <- apply(predicted_survivalF, 2, quantile, prob = 2.5/100)
uciF <- apply(predicted_survivalF, 2, quantile, prob = 97.5/100)
ord <- order(my.constants$winglength)
df <- data.frame(wing_length = c(my.constants$winglength[ord],
                                         my.constants$winglength[ord]),
                     survival = c(mean_survivalM[ord],
                                     mean_survivalF[ord]),
                     lci = c(lciM[ord],lciF[ord]),
                     uci = c(uciM[ord],uciF[ord]),
                     sex = c(rep("male", length(mean_survivalM)),
                               rep("female", length(mean_survivalF))))

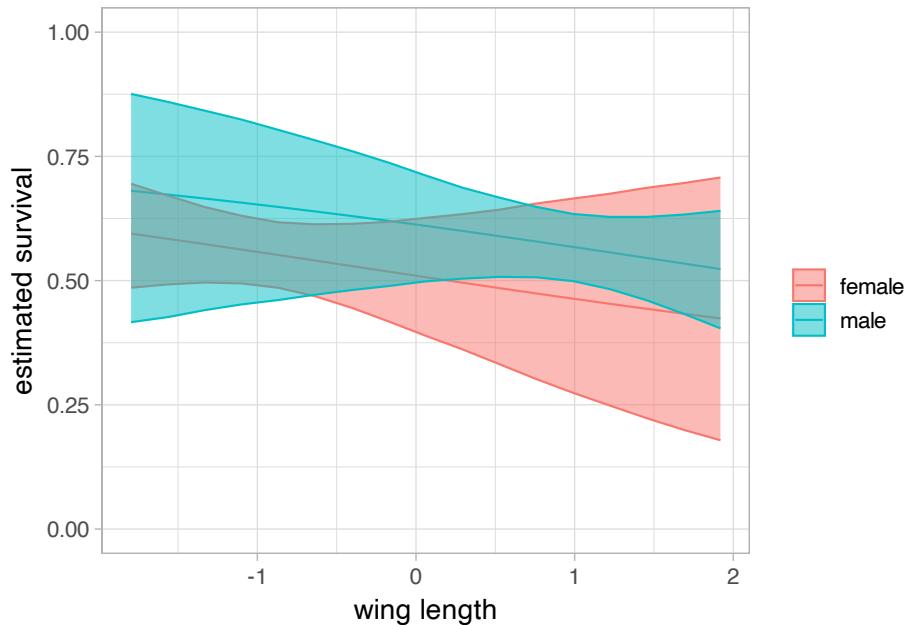
```

Now on a plot:

```

df %>%
  ggplot() +
  aes(x = wing_length, y = survival, color = sex) +
  geom_line() +
  geom_ribbon(aes(ymin = lci, ymax = uci, fill = sex), alpha = 0.5) +
  ylim(0,1) +
  labs(x = "wing length", y = "estimated survival", color = "", fill = "")

```

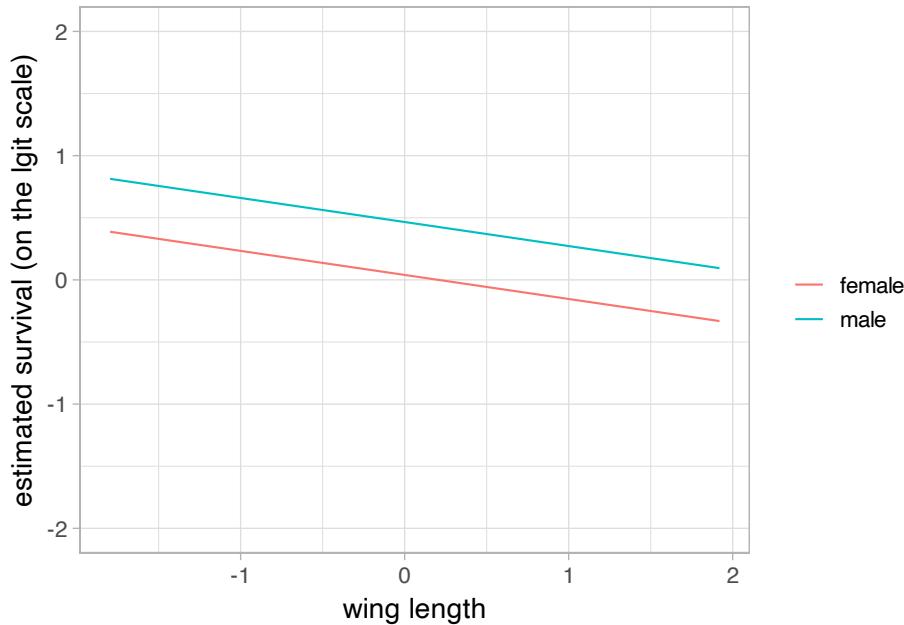


Note that the two curves are not exactly parallel because we back-transformed the linear part of the relationship between survival and wing length. You may check that parallelism occurs on the logit scale:

```

survival = c(mean_lsurvivalM[ord],
            mean_lsurvivalF[ord]),
sex = c(rep("male", length(mean_lsurvivalM)),
        rep("female", length(mean_lsurvivalF)))

df %>%
  ggplot() +
  aes(x = wing_length, y = survival, color = sex) +
  geom_line() +
  ylim(-2,2) +
  labs(x = "wing length",
       y = "estimated survival (on the logit scale)",
       color = "")
```



4.8.4 Random effects

If individual variation in survival is not fully explained by our covariates, we may add random effects through $\text{logit}(\phi_i) = \beta_1 + \beta_2 x_i + \varepsilon_i$ where $\varepsilon_i \sim N(0, \sigma^2)$. We consider individual variation in this section, but the reasoning holds for temporal variation. In essence, we

are treating the individual survival probabilities ϕ_i as a sample from a population of survival probabilities, and we assume a normal distribution with mean the linear component (with or without a covariate) and standard deviation σ . The variation that is unexplained by the covariate x_i is measured by the variation σ in the residuals ε_i .

Why is it important? Ignoring individual heterogeneity generated by individuals having contrasted performances over life may mask senescence or hamper the understanding of life history trade-offs. Overall, failing to incorporate unexplained residual variance may induce bias in parameter estimates and lead to detecting an effect of the covariate more often than it should be.

Let's go back to our dipper example with wing length as a covariate, and write the NIMBLE code:

```

hmm.phiwlrrep <- nimbleCode({
  p ~ dunif(0, 1) # prior detection
  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){
    logit(phi[i]) <- beta[1] + beta[2] * winglength[i] + eps[i]
    eps[i] ~ dnorm(mean = 0, sd = sdeps)
    gamma[1,1,i] <- phi[i]        # Pr(alive t -> alive t+1)
    gamma[1,2,i] <- 1 - phi[i]    # Pr(alive t -> dead t+1)
    gamma[2,1,i] <- 0            # Pr(dead t -> alive t+1)
    gamma[2,2,i] <- 1            # Pr(dead t -> dead t+1)
  }

  beta[1] ~ dnorm(mean = 0, sd = 1.5)
  beta[2] ~ dnorm(mean = 0, sd = 1.5)
  sdeps ~ dunif(0, 10)

  delta[1] <- 1                # Pr(alive t = 1) = 1
  delta[2] <- 0                # Pr(dead t = 1) = 0

  # 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, i])
  y[i,j] ~ dcat(omega[z[i,j], 1:2])
}
})

```

The prior on the standard deviation of the random effect is uniform between 0 and 10.

We now write a function for generating initial values:

```

initial.values <- function() list(beta = rnorm(2,0,1.5),
                                    sdeps = runif(1,0,3),
                                    p = runif(1,0,1),
                                    z = zinits)

```

We specify the parameters to be monitored:

```

parameters.to.save <- c("beta", "sdeps", "p")

```

We increase the number of iterations and the length of the burn-in period to reach better convergence:

```

n.iter <- 10000
n.burnin <- 5000
n.chains <- 2

```

And finally, we run NIMBLE:

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

We inspect the numerical summaries:

```
MCMCssummary(mcmc.phiwlrp, round = 2)
```

```
##          mean    sd  2.5%   50% 97.5% Rhat n.eff
## beta[1]  0.22 0.12 -0.02  0.22  0.44 1.24 1335
## beta[2] -0.01 0.10 -0.20 -0.01  0.19 1.02 1894
## p        0.90 0.03  0.84  0.90  0.95 1.03  752
## sdeps   0.27 0.25  0.01  0.16  0.83 4.68     12
```

The effective sample size for the standard deviation of the random effect is very low. Let's try something else. We use non-centering to reparameterize our model. **Explain**.

```
...
for (i in 1:N){
  logit(phi[i]) <- beta[1] + beta[2] * winglength[i] + sdeps * eps[i]
  eps[i] ~ dnorm(mean = 0, sd = 1)
...
}
```

After running NIMBLE, we inspect the numerical summaries, and see that effective sample sizes are much better:

```
MCMCsummary(mcmc.phiwrep, round = 2)
```

```
##          mean    sd  2.5%   50% 97.5% Rhat n.eff
## beta[1]  0.21 0.12 -0.04  0.21  0.43 1.00 1202
## beta[2] -0.01 0.10 -0.20 -0.01  0.19 1.00 1789
## p        0.90 0.03  0.83  0.90  0.95 1.01  790
## sdeps   0.40 0.26  0.02  0.37  0.95 1.01  170
```

4.8.5 Individual time-varying covariates

So far, we allowed covariates to vary along a single dimension, either time or individual. What if we need to consider a covariate that varies from one animal to the other, and over time. Think of age for example, its value is specific to each individual, and it (sadly) changes over time.

Age has a particular meaning in the capture-recapture framework. It is the time elapsed since first encounter, which is a proxy of true age, but not true age. If age is known at first encounter, then it is true age. For example, if the dippers were marked as young, then we would have the true biological age of each bird.

The convenient thing is that age has no missing value because age at $t + 1$ is just age at t to which we add 1 (we will see an example of an individual covariate with missing values in Section 6.2). This suggests a way to code the age effect in NIMBLE as follows:

```
hmm.phiage.in <- nimbleCode({
  p ~ dunif(0, 1) # prior detection
  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){
    for (t in first[i]:(T-1)){
      # phi1 = beta1 + beta2; phi1+ = beta1
```

```

logit(phi[i,t]) <- beta[1] + beta[2] * equals(t, first[i])
gamma[1,1,i,t] <- phi[i,t]           # Pr(alive t -> alive t+1)
gamma[1,2,i,t] <- 1 - phi[i,t]        # Pr(alive t -> dead t+1)
gamma[2,1,i,t] <- 0                  # Pr(dead t -> alive t+1)
gamma[2,2,i,t] <- 1                  # Pr(dead t -> dead t+1)
}
}
beta[1] ~ dnorm(mean = 0, sd = 1.5) # phi1+
beta[2] ~ dnorm(mean = 0, sd = 1.5) # phi1 - phi1+
phi1plus <- plogis(beta[1])         # phi1+
phi1 <- plogis(beta[1] + beta[2])   # phi1
delta[1] <- 1                      # Pr(alive t = 1) = 1
delta[2] <- 0                      # Pr(dead t = 1) = 0
# 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, i, j-1])
    y[i,j] ~ dcat(omega[z[i,j], 1:2])
  }
}
})

```

Here we used `equals(t, first[i])` which renders 1 when t is the first encounter `first[i]` and 0 otherwise. Therefore we distinguish survival over the first interval after first encounter ϕ_1 (`logit(phi[i,t]) <- beta[1] + beta[2]`) from survival afterwards ϕ_{1+} (`logit(phi[i,t]) <- beta[1]`).

We put all constants in a list:

```

first <- apply(y, 1, function(x) min(which(x != 0)))
my.constants <- list(N = nrow(y),
                     T = ncol(y),
                     first = first)

```

And the data in a list:

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

We write a function to generate initial values:

```
zinit <- y
zinit[zinit == 0] <- 1
initial.values <- function() list(beta = rnorm(2,0,5),
                                    p = runif(1,0,1),
                                    z = zinit)
```

And specify the parameters to be monitored:

```
parameters.to.save <- c("phi1", "phi1plus", "p")
```

We now run NIMBLE:

```
mcmc.phi.age.in <- nimbleMCMC(code = hmm.phiage.in,
                                   constants = my.constants,
                                   data = my.data,
                                   inits = initial.values,
                                   monitors = parameters.to.save,
                                   niter = n.iter,
                                   nburnin = n.burnin,
                                   nchains = n.chains)
```

We display the results:

```
MCMCsummary(mcmc.phi.age.in, round = 2)
```

```
##          mean    sd 2.5% 50% 97.5% Rhat n.eff
## p        0.89 0.03 0.83 0.90  0.94 1.00    402
## phi1     0.56 0.03 0.49 0.55  0.63 1.01    689
## phi1plus 0.57 0.04 0.50 0.57  0.64 1.00    309
```

Age or time elapsed since first encounter does not seem to have an effect on survival here.

Another method to include an age effect is to create an individual by time covariate and use nested indexing (as in the flood/nonflood example) to distinguish survival over the interval after first detection from survival afterwards:

```
age <- matrix(NA, nrow = nrow(y), ncol = ncol(y) - 1)
for (i in 1:nrow(age)){
  for (j in 1:ncol(age)){
    if (j == first[i]) age[i,j] <- 1 # age = 1
    if (j > first[i]) age[i,j] <- 2 # age > 1
  }
}
```

Now we may write the NIMBLE code for this model. We just need to remember that survival is no longer defined on the logit scale as in the previous model, so we simply use uniform priors:

```
hmm.phiage.out <- nimbleCode({
  p ~ dunif(0, 1) # prior detection
  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){
  for (t in first[i]:(T-1)){
    phi[i,t] <- beta[age[i,t]] # beta1 = phi1, beta2 = phi1+
    gamma[1,1,i,t] <- phi[i,t]      # Pr(alive t -> alive t+1)
    gamma[1,2,i,t] <- 1 - phi[i,t]   # Pr(alive t -> dead t+1)
    gamma[2,1,i,t] <- 0             # Pr(dead t -> alive t+1)
    gamma[2,2,i,t] <- 1             # Pr(dead t -> dead t+1)
  }
}
beta[1] ~ dunif(0, 1) # phi1
beta[2] ~ dunif(0, 1) # phi1+
phi1 <- beta[1]
phi1plus <- beta[2]
delta[1] <- 1          # Pr(alive t = 1) = 1
delta[2] <- 0          # Pr(dead t = 1) = 0
# 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, i, j-1])
    y[i,j] ~ dcat(omega[z[i,j], 1:2])
  }
}
})

```

We put all constants in a list, including the age covariate:

```

first <- apply(y, 1, function(x) min(which(x != 0)))
my.constants <- list(N = nrow(y),
                     T = ncol(y),
                     first = first,
                     age = age)

```

We re-write a function to generate initial values:

```

zinits <- y
zinits[zinits == 0] <- 1
initial.values <- function() list(beta = runif(2,0,1),
                                    p = runif(1,0,1),
                                    z = zinits)

```

And we run NIMBLE:

```

mcmc.phi.age.out <- nimbleMCMC(code = hmm.phiage.out,
                                  constants = my.constants,
                                  data = my.data,
                                  inits = initial.values,
                                  monitors = parameters.to.save,
                                  niter = n.iter,
                                  nburnin = n.burnin,
                                  nchains = n.chains)

```

We display numerical summaries for the model parameters, and acknowledge that we obtain similar results to the other parameterization:

```
MCMCssummary(mcmc.phi.age.out, round = 2)
```

```

##          mean    sd 2.5% 50% 97.5% Rhat n.eff
## p      0.90 0.03 0.84 0.90 0.95 1.02   438
## phi1   0.55 0.03 0.48 0.55 0.62 1.00   878
## philplus 0.57 0.04 0.50 0.57 0.64 1.02  1048

```

Like I mentioned earlier, age is easy to deal with as it does not contain missing values. Now think of size or body mass for a minute. The problem is that we cannot record size or body mass when an animal is non-detected. The easiest way to cope with individual time-varying

covariates is to discretize e.g. in small, medium and large as in Chapter 5. Another option is to come up with a model for the covariate and fill in missing values by simulating from this model (see case study in Section 6.2).

4.9 Why Bayes? Incorporate prior information

4.9.1 Prior elicitation

Before we close this section, I'd like to cover one last topic. Think again of the CJS model with constant parameters. So far, we have assumed a non-informative prior on survival $\text{Beta}(1, 1) = \text{Uniform}(0, 1)$. With this prior, we have seen in Section 4.5 that mean posterior survival is $\phi = 0.56$ with credible interval $[0.52, 0.62]$.

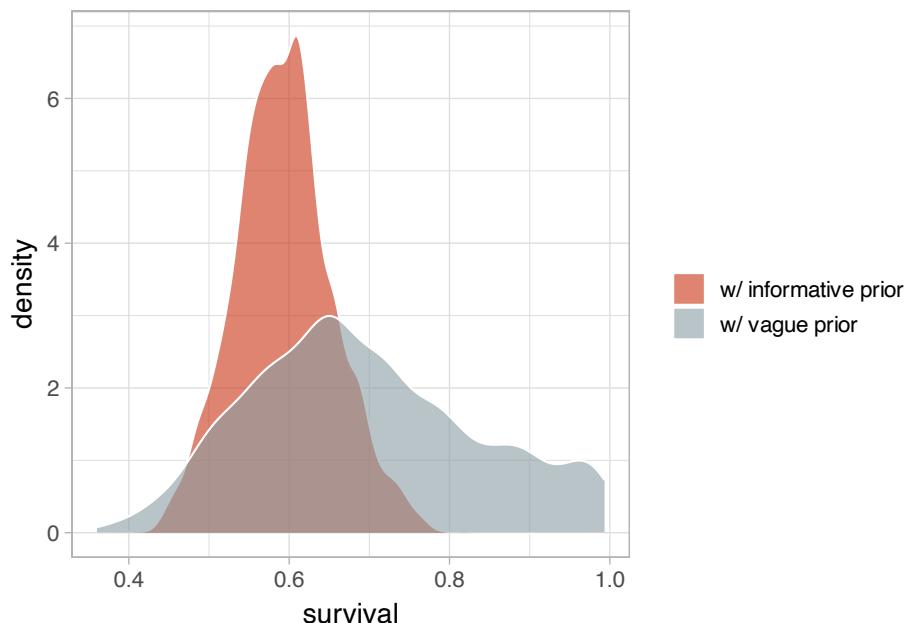
The thing is that we know a lot about passerines and it is a shame not to be able to use this information and act as if we have to start from scratch and know nothing.

We illustrate how to incorporate prior information by acknowledging that species with similar body masses have similar survival. By gathering information on several other European passerines than the dipper, let's assume we have built a regression of survival vs. body mass – an allometric relationship.

Now knowing dippers weigh on average 59.8g, we're in the position to build a prior for dipper survival probability by predicting its value using the regression. We obtain a predicted survival of 0.57 and a standard deviation of 0.075. Using an informative prior `phi ~ dnorm(0.57, sd = 0.073)` in NIMBLE instead of our `phi ~ dunif(0,1)`, we get a mean posterior of 0.56 with credible interval $[0.52, 0.61]$. There's barely no difference with the non-informative prior, quite a disappointment.

Now let's assume that we had only the three first years of data, what would have happened? We fit the model with constant parameters with both the non-informative and informative priors to the dataset from which we delete the final 4 years of data. Now the benefit of us-

ing the prior information becomes clear as the credible interval when prior information is ignored has a width of 0.53, which is more than twice as much as when prior information is used (0.24), illustrating the increased precision provided by the prior. We may assess visually this gain in precision by comparing the survival posterior distributions with and without informative prior:



If the aim is to get an estimate of survival, Gilbert did not have to conduct further data collection after 3 years, and he could have reached the same precision as with 7 years of data by using prior information derived from body mass. In brief, the prior information was worth 4 years of field data. Of course, this is assuming that the ecological question remains the same whether you have 3 or 7 years of data, which is unlikely to be the case, as with long-term data, there is so much we can ask, more than “just” what annual survival probability is.

4.9.2 Moment matching

The prior `phi ~ dnorm(0.57, sd = 0.073)` is not entirely satisfying because it is not constrained to be positive or less than one, which is the minimum for a probability (of survival) to be well defined. In our spe-

cific example, the prior distribution is centered on positive values far from 0, and the standard deviation is small enough so that the chances to get values smaller than 0 or higher than 1 are null (to convince yourself, just type in `hist(rnorm(1000, mean = 0.57, sd = 0.073))` in R). Can we do better? The answer is yes.

Remember the Beta distribution? Recall that the Beta distribution is a continuous distribution with values between 0 and 1, which is very convenient to specify priors for survival and detection probabilities. Plus we know everything about the Beta distribution, in particular its moments. If $X \sim Beta(\alpha, \beta)$, then the first (mean) and second moments (variance) of X are $\mu = E(X) = \frac{\alpha}{\alpha+\beta}$ and $\sigma^2 = \text{Var}(X) = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$.

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:

$$\begin{aligned}\alpha &= \left(\frac{1-\mu}{\sigma^2} - \frac{1}{\mu} \right) \mu^2 \\ \beta &= \alpha \left(\frac{1}{\mu} - 1 \right)\end{aligned}$$

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 we simply have to use `phi ~ dbeta(25.6,19.3)` as a prior instead of our `phi ~ dnorm(0.57, sd = 0.073)`.

4.10 Summary

- The CJS model is a HMM with time-varying parameters to account for variation due to environmental conditions in survival or to sampling effort in detection.
 - Covariates can be considered to try and explain temporal and/or individual variation in survival and detection probabilities. If needed, random effects can be added to cope with unexplained variation.
 - For model comparison, the WAIC can be used to evaluate the relative predictive performance of capture-recapture models.
 - Statistical models rely on assumptions, and the CJS model makes no exception. There are procedures to assess the goodness of fit of the CJS model to capture-recapture data.
-

4.11 Suggested reading

- [Buckland \[2016\]](#) provides a historical perspective on the CJS model and anecdotes. The monography by [Lebreton et al. \[1992\]](#) is a must-read to better understand the CJS model and its applications. The HMM formulation of the CJS model was proposed by [Gimenez et al. \[2007\]](#) and [Royle \[2008\]](#).
- [Gimenez et al. \[2009c\]](#) deals with parameter redundancy in capture-recapture models in a Bayesian framework. For an exhaustive treatment, see [Cole \[2020\]](#) excellent book.
- Relative to model comparison, I warmly recommend [McElreath \[2020\]](#) to better understand WAIC (there is a video as well, see <https://www.youtube.com/watch?v=vSjL2Zc-gEQ>). The paper by [Gelman et al. \[2014\]](#) is also much helpful.

- On posterior predictive checks, you may check out [Conn et al. \[2018\]](#). The `R2ucare` package is introduced in [Gimenez et al. \[2018b\]](#).
- Temporal heterogeneity is addressed in papers by [Grosbois et al. \[2008\]](#) and [Frederiksen et al. \[2014\]](#), while individual heterogeneity is reviewed by [Gimenez et al. \[2018a\]](#).
- Regarding covariates, I did not use the formalization of linear models and stucked to an intuitive (hopefully) illustration of the use of covariates. More details can be found in Chapter 6 of [Cooch and White \[2017\]](#).
- The example on how to incorporate prior information is in [McCarthy and Masters \[2005\]](#).

5

Sites and states

5.1 Introduction

In this fifth chapter, you will learn about the Arnason-Schwarz model that allows estimating transitions between sites and states based on capture-recapture data. You will also see how to deal with uncertainty in the assignment of states to individuals.

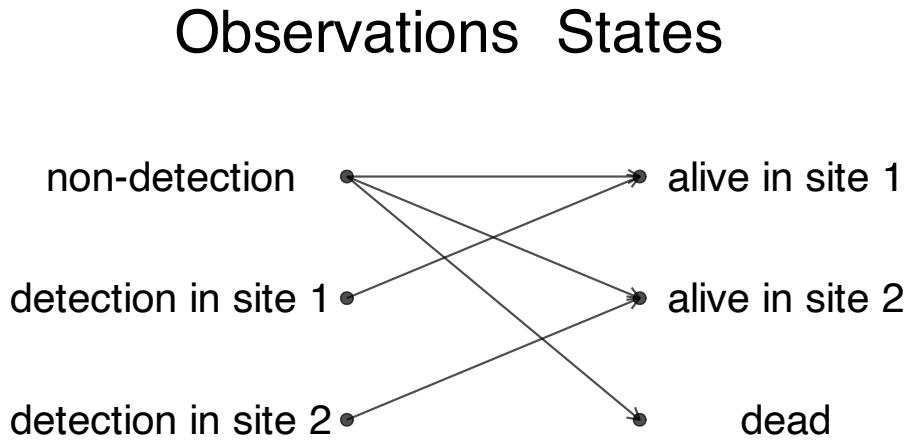
5.2 The Arnason-Schwarz (AS) model

In Chapter 4, we got acquainted with the Cormack-Jolly-Seber (CJS) model which accommodates transitions between the states alive and dead while accounting for imperfect detection. It is often the case that besides being alive, more detailed information is collected on the state of animals when they are detected. For example, if the study area is split into several discrete sites, you may record where an animal is detected, the state being now alive in this particular site. The Arnason-Schwarz (AS) model can be viewed as an extension to the CJS model in which we estimate movements between sites on top of survival. The AS model is named after the two statisticians – Neil Arnason and Carl Schwarz – who came up with the idea.

5.2.1 Theory

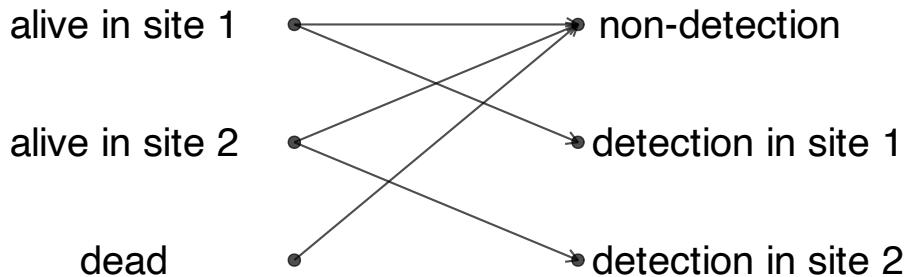
Let's assume for now that we have two sites, say 1 and 2. The way we usually think of analyzing the data is to start from the detections

and non-detections and infer the transitions between sites and movements. Schematically, when an animal is detected in site 1 or site 2, it obviously means that it is alive in that site, whereas when it is not detected, it may be dead or alive in either site. Schematically, we have:



Observations and states are indeed closely related, but we do not have a perfect states to observations correspondence, and the HMM framework will help you make the distinction clear, which in turn will make the modelling easier. Usually, we focus our energy on the observations but what we'd really like is to spend time thinking of the ecological processes that we observed imperfectly. In the HMM framework, when we are to build a model, we think of the states and their dynamic over time, and these states emit the observations we're given to make. Going back to our example, when an animal is alive in either site, it may get detected in that site or go undetected. When an animal is dead, then it goes undetected for sure. Schematically, we obtain:

States Observations



We have $z = 1$ for alive in site 1, $z = 2$ for alive in site 2 and $z = 3$ for dead. We will code $y = 1$ for non-detected, $y = 2$ for detected in site 1 and $y = 3$ for detected in site 2. The parameters are:

- π^r is the probability the a newly encountered individual is in state r ;
- p_t^r is the probability of detection at t for a bird in site r at t ;
- ϕ_t^r is the survival probability for birds in site r at between t and $t + 1$;
- ψ_t^{rs} is the probability of being in site s at time $t + 1$ for animals that were in site r at t and have survived to $t + 1$, in short movement conditional on survival.

These parameters can be modelled as functions of covariates as in Section 4.8. But for now we will drop the time index for simplicity.

We follow the presentation of HMM as in Chapter 3. Let's start with the vector of initial states. At first encounter, an animal may be alive in site 1 with probability π^1 or in site 2 with the complementary probability $1 - \pi^1$, but it cannot be dead. Therefore, we have:

$$\times = \begin{pmatrix} z_t = 1 & z_t = 2 & z_t = 3 \\ \pi^1 & 1 - \pi^1 & 0 \end{pmatrix}$$

We move on to the transition matrix which connects the states at $t - 1$ in rows to the states at t in columns. For example, the probability of moving from site 1 at $t - 1$ to site 2 at t is the product of the survival probability in site 1 over that time interval, times the probability of moving from site 1 to 2 for those animals who survived. We have:

$$\times = \begin{pmatrix} z_t = 1 & z_t = 2 & z_t = 3 \\ \bar{\phi}^T(1 - \psi^{12}) & \bar{\phi}^T\psi^{12} & 1 - \bar{\phi}^T \\ \phi^2\psi^{21} & \phi^2(1 - \psi^{21}) & 1 - \phi^2 \\ 0 & 0 & 1 \end{pmatrix} \begin{matrix} z_{t-1} = 1 \\ z_{t-1} = 2 \\ z_{t-1} = 3 \end{matrix}$$

Finally, the observation matrix relates the states an animal is in at t in rows to the observations at t in columns. If an animal is dead ($z_t = 3$), it cannot be detected ($\Pr(y_t = 1|z_t = 3) = \Pr(y_t = 2|z_t = 3) = 0$ and $\Pr(y_t = 3|z_t = 3) = 1$), whereas when it is alive in either site, it can be detected or not. We have:

$$\times = \begin{pmatrix} y_t = 1 & y_t = 2 & y_t = 3 \\ 1 - p^1 & p^1 & 0 \\ 1 - p^2 & 0 & p^2 \\ 1 & 0 & 0 \end{pmatrix} \begin{matrix} z_t = 1 \\ z_t = 2 \\ z_t = 3 \end{matrix}$$

The vector of initial state probabilities sums up to one, as well as the rows of the transition and observation matrices.

5.3 Fitting the AS model

5.3.1 Geese data

To introduce this chapter, we will use data on the Canada goose (*Branta canadensis*; geese hereafter) kindly provided by Jay Hestbeck (Figure 4.2).

In total, 21277 geese were captured, marked with coded neck bands and recaptured between 1984 and 1989 in their wintering locations. Specifically, geese were monitored in the Atlantic flyway, in large areas along the East coast of the USA, namely 3 sites in the mid-Atlantic (New York, Pennsylvania, New Jersey), Chesapeake (Delaware, Maryland, Virginia), and Carolinas (North and South Carolina). Birds were adults and sub-adults when banded.

You may see the data below:



FIGURE 5.1: Canada goose (*Branta canadensis*). Credit: Max McCarthy.

```
y <- read_csv("dat/geese.csv") %>% as.matrix()
head(y)
##      year_1984 year_1985 year_1986 year_1987 year_1988
## [1,]      0       2       2       0       0
## [2,]      0       0       0       0       0
## [3,]      0       0       0       1       0
## [4,]      0       0       2       0       0
## [5,]      0       3       0       0       3
## [6,]      0       0       0       2       0
##      year_1989
## [1,]      0
## [2,]      2
## [3,]      0
## [4,]      0
```

```
## [5,]      2
## [6,]      0
```

The six columns are years in which the geese were captured, banded and recapture. A 0 stands for a non-detection, and detections were coded in the 3 wintering sites 1, 2 and 3 for mid-Atlantic, Chesapeake and Carolinas respectively. This is only a subsample of 500 individuals of the whole dataset that I will use for illustration here.

5.3.2 NIMBLE implementation

To write the NIMBLE code corresponding to the AS model, we will make our life easier and start with 2 sites only – we drop the Carolinas wintering site for now. We replace all 3's by 0's in the dataset:

```
y[y==3] <- 0
```

Also we consider parameters constant over time.

We start with comments to define the quantities – parameters, states and observations – we will use in the NIMBLE code:

```
multisite <- nimbleCode({
  # -----
  # Parameters:
  # phi1: survival probability site 1
  # phi2: survival probability site 2
  # psi12: movement probability from site 1 to site 2
  # psi21: movement probability from site 2 to site 1
  # p1: detection probability site 1
  # p2: detection probability site 2
  # -----
  # States (z):
```

```

# 1 alive at site 1
# 2 alive at site 2
# 3 dead
# Observations (y):
# 1 not seen
# 2 seen at site 1
# 3 seen at site 2
# -----
...

```

The we specify priors for the survival, transition and detection probabilities:

```

multisite <- nimbleCode({
...
# Priors
phi1 ~ dunif(0, 1)
phi2 ~ dunif(0, 1)
psi12 ~ dunif(0, 1)
psi21 ~ dunif(0, 1)
p1 ~ dunif(0, 1)
p2 ~ dunif(0, 1)
...

```

We now write the vector of initial state probabilities:

```

multisite <- nimbleCode({
...
# initial state probabilities
delta[1] <- pi1          # Pr(alive in 1 at t = first)
delta[2] <- 1 - pi1        # Pr(alive in 2 at t = first)
delta[3] <- 0              # Pr(dead at t = first) = 0
...

```

Actually, the initial state is known exactly: It is alive at site of initial capture, and π^1 is the proportion of individuals first captured in site 1, there is no need to make it explicit in the model and estimate it. Therefore, in the likelihood, instead of `z[i,first[i]] ~ dcat(delta[1:3])`, you can use `z[i,first[i]] <- y[i,first[i]] - 1` and just forget about π^1 , for now. Note that the same trick applies to the CJS model.

We write the transition matrix:

```
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] <- phi1 * (1 - psi12)
  gamma[1,2] <- phi1 * psi12
  gamma[1,3] <- 1 - phi1
  gamma[2,1] <- phi2 * psi21
  gamma[2,2] <- phi2 * (1 - psi21)
  gamma[2,3] <- 1 - phi2
  gamma[3,1] <- 0
  gamma[3,2] <- 0
  gamma[3,3] <- 1
  ...
})
```

In the same way, the observation matrix is:

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

  omega[1,1] <- 1 - p1      # Pr(alive 1 t -> non-detected t)
  omega[1,2] <- p1          # Pr(alive 1 t -> detected site 1 t)
  omega[1,3] <- 0           # Pr(alive 1 t -> detected site 2 t)
```

```

omega[2,1] <- 1 - p2      # Pr(alive 2 t -> non-detected t)
omega[2,2] <- 0           # Pr(alive 2 t -> detected site 1 t)
omega[2,3] <- p2          # Pr(alive 2 t -> detected site 2 t)
omega[3,1] <- 1           # Pr(dead t -> non-detected t)
omega[3,2] <- 0           # Pr(dead t -> detected site 1 t)
omega[3,3] <- 0           # Pr(dead t -> detected site 2 t)
...

```

At last, we are ready to specify the likelihood which, and this the magic of HMM, is the same as the likelihood of the CJS model, only the vector of initial states, the transition and observation matrices were changed:

```

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])
    }
  }
})

```

We need to provide NIMBLE with constants, data, initial values, some parameters to monitor and MCMC details:

```

# occasions of first capture
first <- apply(y, 1, function(x) min(which(x != 0)))

```

```

# constants
my.constants <- list(first = first,
                      K = ncol(y),
                      N = nrow(y))

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

# initial values
zinit <- y # say states are observations, detections in A or B are taken as alive in same site
zinit[zinit==0] <- sample(c(1,2), sum(zinit==0), replace = TRUE) # non-detections become alive
initial.values <- function(){list(phi1 = runif(1, 0, 1),
                                    phi2 = runif(1, 0, 1),
                                    psi12 = runif(1, 0, 1),
                                    psi21 = runif(1, 0, 1),
                                    p1 = runif(1, 0, 1),
                                    p2 = runif(1, 0, 1),
                                    pi1 = runif(1, 0, 1),
                                    z = zinit)}

# parameters to monitor
parameters.to.save <- c("phi1", "phi2", "psi12", "psi21", "p1", "p2", "pi1")
# MCMC details
n.iter <- 5000
n.burnin <- 1000
n.chains <- 2

```

Now we may run NIMBLE:

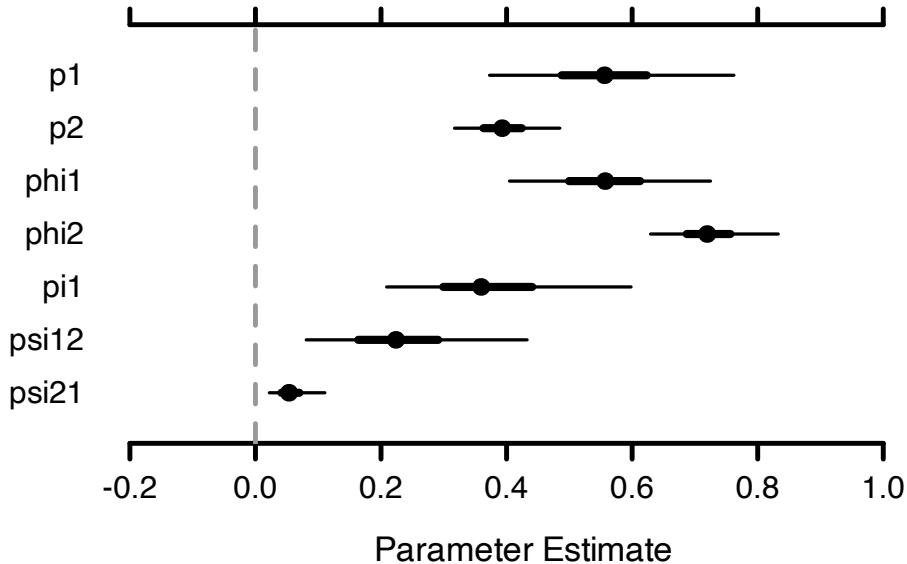
```

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

```

We may have a look to the results via a caterpillar plot:

MCMCplot(mcmc.multisite)



Remember mid-Atlantic is site 1, and Chesapeake site 2. Detection in mid-Atlantic (around 0.5) is higher than in Chesapeake (around 0.4) although it comes with more uncertainty (wider credible interval). Survival in both sites are similar estimated at around 0.6–0.7. Note that by going multisite, we could make these parameters site-specific and differences might reflect habitat quality for example. Now the novelty lies in our capability to estimate movements from site 1 to site 2 and from site 2 to site 1 from a winter to the next. The annual probability of remaining in the same site for two successive winters, used as a measure of site fidelity, was lower in the mid-Atlantic (around 0.7) than in the Chesapeake (0.9). The estimated probability of moving to the Chesapeake from the mid-Atlantic was three times as high as the probability of moving in the opposite direction.

We may also have a look to numerical summaries, which confirm our ecological interpretation of the model parameter estimates:

```
MCMCsummary(mcmc.multisite, round = 2)
##      mean    sd 2.5% 50% 97.5% Rhat n.eff
## p1    0.56 0.10 0.37 0.56  0.76 1.00   157
## p2    0.40 0.04 0.32 0.39  0.48 1.00   217
## phi1  0.56 0.08 0.40 0.56  0.72 1.04   155
## phi2  0.72 0.05 0.63 0.72  0.83 1.02   120
## pi1   0.37 0.10 0.21 0.36  0.60 1.02    62
## psi12 0.23 0.09 0.08 0.22  0.43 1.03   171
## psi21 0.06 0.02 0.02 0.05  0.11 1.01   336
```

You could add time-dependence to the demographic parameters, e.g. survival and movements, and assess the effect of winter harshness with some temporal covariates. Individual covariates could also be considered.

Before we move to the next section, I will illustrate the use of `nimbleEcology` to fit the AS to the geese data with 2 sites (see Section 3.8.3). Using the function `dHMM()` which implements HMM with time-independent observation and transition matrices, we have:

```
# read in data
geese <- read_csv("geese.csv", col_names = TRUE)
y <- as.matrix(geese)
# drop Carolinas
y[y==3] <- 0 # act as if there was no detection in site 3 Carolinas
mask <- apply(y, 1, sum)
y <- y[mask!=0,] # remove rows w/ 0s only
# get occasions of first encounter
get.first <- function(x) min(which(x != 0))
first <- apply(y, 1, get.first)
# filter out individuals that are first captured at last occasion.
# These individuals do not contribute to parameter estimation,
# and also they cause problems with nimbleEcology
mask <- which(first!=ncol(y))
y <- y[mask, ] # keep only these
```

```
first <- first[mask]

# NIMBLE code
multisite.marginalized <- nimbleCode({ 

  # -----
  # Parameters:
  # phi1: survival probability site 1
  # phi2: survival probability site 2
  # psi12: movement probability from site 1 to site 2
  # psi21: movement probability from site 2 to site 1
  # p1: recapture probability site 1
  # p2: recapture probability site 2
  # pi1: prop of being in site 1 at initial capture
  # ----

  # States (z):
  # 1 alive at 1
  # 2 alive at 2
  # 3 dead

  # Observations (y):
  # 1 not seen
  # 2 seen at site 1
  # 3 seen at site 2
  # ----

  # priors
  phi1 ~ dunif(0, 1)
  phi2 ~ dunif(0, 1)
  psi12 ~ dunif(0, 1)
  psi21 ~ dunif(0, 1)
  p1 ~ dunif(0, 1)
  p2 ~ dunif(0, 1)
  pi1 ~ dunif(0, 1)

  # probabilities of state z(t+1) given z(t)
  gamma[1,1] <- phi1 * (1 - psi12)
```

```

gamma[1,2] <- phi1 * psi12
gamma[1,3] <- 1 - phi1
gamma[2,1] <- phi2 * psi21
gamma[2,2] <- phi2 * (1 - psi21)
gamma[2,3] <- 1 - phi2
gamma[3,1] <- 0
gamma[3,2] <- 0
gamma[3,3] <- 1

# probabilities of y(t) given z(t)
omega[1,1] <- 1 - p1      # Pr(alive 1 t -> non-detected t)
omega[1,2] <- p1        # Pr(alive 1 t -> detected 1 t)
omega[1,3] <- 0          # Pr(alive 1 t -> detected 2 t)
omega[2,1] <- 1 - p2      # Pr(alive 2 t -> non-detected t)
omega[2,2] <- 0          # Pr(alive 2 t -> detected 1 t)
omega[2,3] <- p2        # Pr(alive 2 t -> detected 2 t)
omega[3,1] <- 1          # Pr(dead t -> non-detected t)
omega[3,2] <- 0          # Pr(dead t -> detected 1 t)
omega[3,3] <- 0          # Pr(dead t -> detected 2 t)

# likelihood
# initial state probs
for(i in 1:N) {
  init[i, 1:3] <- gamma[ y[i, first[i]] - 1, 1:3 ] # first state propagation
}
for (i in 1:N){
  y[i,(first[i]+1):K] ~ dHMM(init = init[i,1:3], # count data from first[i] + 1
                                probObs = omega[1:3,1:3],    # observation matrix
                                probTrans = gamma[1:3,1:3],   # transition matrix
                                len = K - first[i],         # nb of occasions
                                checkRowSums = 0)           # do not check whether elements ...
}
}

# constants
my.constants <- list(first = first,
                      K = ncol(y),

```

```

N = nrow(y))

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

# initial values
initial.values <- function(){list(phi1 = runif(1, 0, 1),
                                    phi2 = runif(1, 0, 1),
                                    psi12 = runif(1, 0, 1),
                                    psi21 = runif(1, 0, 1),
                                    p1 = runif(1, 0, 1),
                                    p2 = runif(1, 0, 1))}

# parameters to monitor
parameters.to.save <- c("phi1", "phi2", "psi12", "psi21", "p1", "p2")

# MCMC details
n.iter <- 5000
n.burnin <- 1000
n.chains <- 2

# run NIMBLE
mcmc.multisite.marginalized <- nimbleMCMC(code = multisite.marginalized,
                                              constants = my.constants,
                                              data = my.data,
                                              inits = initial.values,
                                              monitors = parameters.to.save,
                                              niter = n.iter,
                                              nburnin = n.burnin,
                                              nchains = n.chains)

```

We obtain:

```

MCMCsummary(mcmc.multisite.marginalized, round = 2)
##      mean   sd 2.5% 50% 97.5% Rhat n.eff
## p1    0.53 0.09 0.36 0.53  0.71     1   651
## p2    0.40 0.04 0.32 0.39  0.49     1   618
## phi1  0.60 0.05 0.51 0.60  0.71     1   935
## phi2  0.70 0.04 0.63 0.70  0.76     1   804

```

```
## psi12 0.27 0.06 0.17 0.26 0.39    1   973
## psi21 0.07 0.02 0.04 0.07 0.11    1   965
```

There are slight differences in these parameters estimates compared to those we obtained earlier. This is probably due to the effective sample sizes being much bigger here (by a factor 3) with the marginalized likelihood for the same number of MCMC iterations.

5.3.3 Goodness of fit

Like for the CJS model, we need to ensure that the AS model provides a good fit to the data. To do so, both posterior predictive checks and classical procedures can be used. I illustrate the use of posterior predictive checks in Section 7.6. With regard to classical tests, the goodness-of-fit testing procedures we covered in Section 4.7 for the CJS model have been extended to the AS model. These procedure are also available in the package `R2ucare`. While the transience and trap-dependence tests can be used on each site, it is worth mentionning a test that is specific to the AS model with several sites. The “where before where after” (WBWA) tests for memory, in other words it’s a test for the Markovian property of the HMM in the particular context of capture-recapture. WBWA quantifies whether there is a relationship between where an animal has last been seen and where it will next be seen again. If a relationship exists, positive or negative, this suggests memory in the movements, and the probability for an animal of moving to a site at t , given it is present in a site at $t - 1$ should be made dependent of where it was at $t - 2$. This is called a second-order Markov process.

Going back to the geese data, the WBWA test is implemented as follows on the whole dataset:

```
library(R2ucare)
geese <- read_csv2("dat/allgeese.csv") %>% as.matrix()
y <- geese[,1:6]
size <- geese[,7]
```

```
wbwa <- test3Gwbwa(y, size)
wbwa$test3Gwbwa
## stat    df p_val
## 472.9  20.0   0.0
```

There is clearly a strong (not to say significant) positive relationship judging by the value of the statistic. I will demonstrate how to account for this memory issue in an extension of the AS model in a case study in Section 7.5.

5.4 What if more than 2 sites?

So far we have considered two sites only, for the sake of simplicity. Indeed when going for three sites (or more), a difficulty arises. While the movement probabilities still need to be between 0 and 1, the sum of all probabilities of moving from a site should also sum to one. This is because an individual alive in site 1 for example, has to stay in 1, move to 2 or move to 3, it has no other choice. This is no problem when you have only two sites because the probability of moving from 1 to 2 is always estimated between 0 and 1, and its complementary, the probability of moving from 2 to 1 will be too. When you have three sites, it might happen that the sum of the estimates for ψ^{12} and ψ^{13} is larger than one, even though they're both between 0 and 1, which would make $\psi^{11} = 1 - \psi^{12} - \psi^{13}$ negative.

There are basically two methods to fulfill both constraints, either to assign a Dirichlet (which is pronounced deer-eesh-lay) prior to the movement probabilities, or to use a multinomial logit link function.

5.4.1 Dirichlet prior

The Dirichlet distribution extends the Beta distribution multivariate we have seen previously (Figure 1.2) for values between 0 and 1 that

add up to 1. Going back to our example with 3 sites, we would like to come up with a prior for parameters of moving from 1, which are ψ^{11} , ψ^{12} and ψ^{13} . The Dirichlet distribution of dimension 3 has a vector of parameters $(\alpha_1, \alpha_2, \alpha_3)$ that controls its shape. The sum of all α 's can be interpreted as a measure of precision: The higher the sum, the more peaked is the distribution on the mean value, the mean value along each dimension being the ratio of the corresponding over the sum of all α 's. When all α 's are equal, the distribution is symmetric (Figure 5.2). Its mean is $(1/3, 1/3, 1/3)$ in our example with 3 parameters, whatever the value of α . When $\alpha_1 = \alpha_2 = \alpha_3 = 1$, we obtain a uniform marginal distribution for the ψ 's, while values below 1 ($\alpha_1 = \alpha_2 = \alpha_3 = 0.1$) result in the distribution concentrating in the corners (skewed U-shaped forms) and values above 1 ($\alpha_1 = \alpha_2 = \alpha_3 = 10$) result in unimodal marginal distributions.

```
library(gtools) # to make the rdirichlet() function available
library(ggtern) # to visually represent multidim prob distribution
set.seed(123) # for reproducibility
n <- 1000 # number of values drawn from Dirichlet distribution
alpha1 <- c(.1, .1, .1)
p1 <- rdirichlet(n, alpha1)
alpha2 <- c(1, 1, 1)
p2 <- rdirichlet(n, alpha2)
alpha3 <- c(10, 10, 10)
p3 <- rdirichlet(n, alpha3)
df <- cbind(rbind(p1, p2, p3), c(rep("alpha = c(0.1, 0.1, 0.1)", n),
                                         rep("alpha = c(1, 1, 1)", n),
                                         rep("alpha = c(10, 10, 10)", n))) %>%
  as_tibble() %>%
  mutate(x = as.numeric(V1),
        y = as.numeric(V2),
        z = as.numeric(V3),
        alpha = V4)

df %>%
  ggtern(aes(x = x, y = y, z = z)) +
```

```

stat_density_tern(aes(fill=..level.., alpha=..level..),
                  geom = 'polygon',
                  bdl = 0.005) + # a 2D kernel density estimation of the distribution
scale_fill_viridis_b() +
geom_point(alpha = 0.3, pch = "+") +
theme_showarrows() +
scale_T_continuous(breaks = seq(0, 1, by = 0.2),
                   labels = seq(0, 1, by = 0.2)) +
scale_L_continuous(breaks = seq(0, 1, by = 0.2),
                   labels = seq(0, 1, by = 0.2)) +
scale_R_continuous(breaks = seq(0, 1, by = 0.2),
                   labels = seq(0, 1, by = 0.2)) +
labs(x = "",
      y = "",
      z = "",
      Tarrow = "psi11",
      Larrow = "psi12",
      Rarrow = "psi13") +
guides(color = "none", fill = "none", alpha = "none") +
facet_wrap(~alpha, ncol = 3)

```

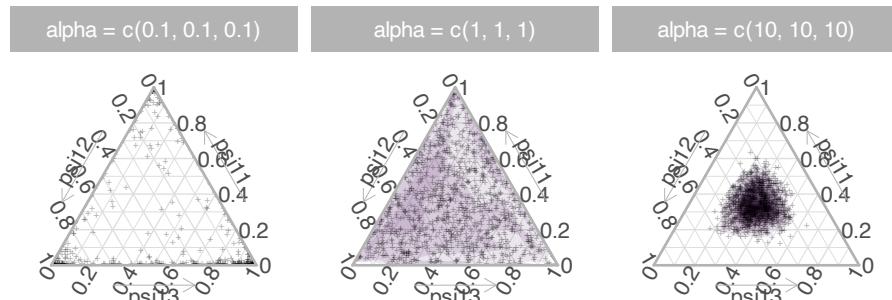


FIGURE 5.2: The Dirichlet distribution as a prior for $(\psi^{11}, \psi^{12}, \psi^{13})$ with vector of parameters α . Here all components of α are equal which makes the distribution symmetric, and its mean is $(1/3, 1/3, 1/3)$. When $\alpha = 1$, the prior for the ψ 's is uniform (middle panel), unimodal when $\alpha = 10$ (right panel) and concentrated in the corners (0 and 1) when $\alpha = 0.1$ (left panel).

Going back to our example, in NIMBLE, we consider a Dirichlet prior for each triplet of movement parameters, from site 1 (ψ^{11} , ψ^{12} and ψ^{13}), from site 2 (ψ^{21} , ψ^{22} and ψ^{23}) and from site 3 (ψ^{31} , ψ^{32} and ψ^{33}).

We start by setting the scene with comments:

```
...
# -----
# Parameters:
# phi1: survival probability site 1
# phi2: survival probability site 2
# phi3: survival probability site 3
# psi11 = psi1[1]: movement probability from site 1 to site 1 (reference)
# psi12 = psi1[2]: movement probability from site 1 to site 2
# psi13 = psi1[3]: movement probability from site 1 to site 3
# psi21 = psi2[1]: movement probability from site 2 to site 1
# psi22 = psi2[2]: movement probability from site 2 to site 2 (reference)
# psi23 = psi2[3]: movement probability from site 2 to site 3
# psi31 = psi3[1]: movement probability from site 3 to site 1
# psi32 = psi3[2]: movement probability from site 3 to site 2
# psi33 = psi3[3]: movement probability from site 3 to site 3 (reference)
# p1: recapture probability site 1
# p2: recapture probability site 2
# p3: recapture probability site 3
# -----
# States (z):
# 1 alive at 1
# 2 alive at 2
# 2 alive at 3
# 3 dead
# Observations (y):
# 1 not seen
# 2 seen at 1
# 3 seen at 2
# 3 seen at 3
...
...
```

```

multisite <- nimbleCode({
  ...
  # transitions: Dirichlet priors
  psi1[1:3] ~ ddirch(alpha[1:3]) # psi11, psi12, psi13
  psi2[1:3] ~ ddirch(alpha[1:3]) # psi21, psi22, psi23
  psi3[1:3] ~ ddirch(alpha[1:3]) # psi31, psi32, psi33
  ...
}

```

Then we use these parameters (which now respect the constraints) to define the transition matrix:

```

multisite <- nimbleCode({
  ...
  # probabilities of state z(t+1) given z(t)
  gamma[1,1] <- phi1 * psi1[1]
  gamma[1,2] <- phi1 * psi1[2]
  gamma[1,3] <- phi1 * psi1[3]
  gamma[1,4] <- 1 - phi1
  gamma[2,1] <- phi2 * psi2[1]
  gamma[2,2] <- phi2 * psi2[2]
  gamma[2,3] <- phi2 * psi2[3]
  gamma[2,4] <- 1 - phi2
  gamma[3,1] <- phi3 * psi3[1]
  gamma[3,2] <- phi3 * psi3[2]
  gamma[3,3] <- phi3 * psi3[3]
  gamma[3,4] <- 1 - phi3
  gamma[4,1] <- 0
  gamma[4,2] <- 0
  gamma[4,3] <- 0
  gamma[4,4] <- 1
  ...
}

```

When we fit this model to the geese dataset with the detections in the Carolinas wintering site back in, and with `alpha <- c(1, 1, 1)` passed to the constants, we obtain the following results:

```
MCMCsummary(mcmc.multisite, round = 2)
##          mean    sd 2.5% 50% 97.5% Rhat n.eff
## p1      0.51 0.08 0.36 0.51  0.67 1.02   306
## p2      0.45 0.05 0.36 0.45  0.55 1.00   217
## p3      0.26 0.06 0.15 0.25  0.39 1.01   166
## phi1    0.60 0.05 0.51 0.60  0.70 1.01   384
## phi2    0.70 0.04 0.63 0.70  0.77 1.00   233
## phi3    0.75 0.06 0.62 0.76  0.87 1.03   218
## psi1[1] 0.74 0.05 0.63 0.75  0.84 1.01   818
## psi1[2] 0.24 0.05 0.14 0.24  0.35 1.02   817
## psi1[3] 0.02 0.02 0.00 0.01  0.07 1.05   487
## psi2[1] 0.07 0.02 0.04 0.07  0.12 1.00   668
## psi2[2] 0.84 0.04 0.75 0.84  0.90 1.00   292
## psi2[3] 0.09 0.03 0.04 0.08  0.17 1.00   220
## psi3[1] 0.02 0.01 0.00 0.02  0.06 1.00  1022
## psi3[2] 0.22 0.05 0.12 0.21  0.33 1.01   525
## psi3[3] 0.76 0.06 0.64 0.76  0.86 1.01   506
```

Survival probabilities are similar among sites, but the detection probability in Carolinas seems much lower than in the two other wintering sites. The estimated probability of moving to the Chesapeake from the Carolinas is 2 times as high as the probability of moving in the opposite direction.

In theory, you could include covariates as in 4.8 through the α parameters and the use a log link function (to ensure $\alpha > 0$), e.g. $\log(\alpha) = \beta_1 + \beta_2 x$. However, NIMBLE does not allow that. Fortunately, there is another way to specify the Dirichlet distribution through a ratio of gamma distributions that allows to incorporate covariates.

The gamma distribution is continuous. It has two parameters α and θ that control its shape and scale (Figure 5.3). Another parameterization considers its shape and rate which is the inverse of the scale.

If we have three independent random variables Y_1, Y_2 and Y_3 distributed as gamma distributions with shape parameters the α 's and same scale parameter θ , that is $Y_j \sim \text{Gamma}(\alpha_j, \theta)$, then it can be shown that the vector $(Y_1/V, Y_2/V, Y_3/V)$ is Dirichlet with vector

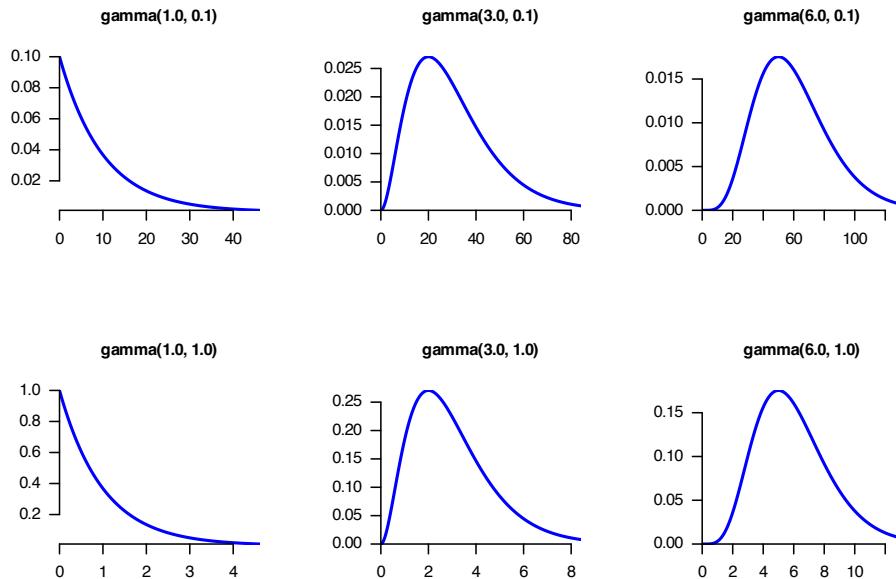


FIGURE 5.3: The distribution $\text{gamma}(\alpha, \theta)$ for different values of α and θ . The shape argument α determines the overall shape while the scale parameter θ only affects the scale values (compare the values on X- and Y-axes between the bottom and upper panels). The exponential and chi-square distributions are particular cases of the gamma distribution. If the parameter shape is close to zero, the gamma is very similar to the exponential (bottom and upper left panels). If the parameter shape is large, then the gamma is similar to the chi-squared distribution (bottom and upper right panels).

of parameters the α 's, where V is the sum of the Y 's (which is also gamma distributed). In NIMBLE, this suggests using $\theta = 1$ to get a uniform prior:

```

...
# transitions: Dirichlet priors with Gamma formulation
for (s in 1:3){
  lpsi1[s] ~ dgamma(alpha[s], 1) # Y1, Y2, Y3 for psi11, psi12, psi13
  psi1[s] <- lpsi1[s]/V1 # psi11, psi12, psi13
  lpsi2[s] ~ dgamma(alpha[s], 1) # Y'1, Y'2, Y'3 for psi21, psi22, psi23
  psi2[s] <- lpsi2[s]/V2 # psi21, psi22, psi23
}

```

```

lpsi3[s] ~ dgamma(alpha[s], 1) # Y'1, Y'2, Y'3 for psi31, psi32, psi33
psi3[s] <- lpsi3[s]/V3 # psi31, psi32, psi33
}
v1 <- sum(lpsi1[1:3])
v2 <- sum(lpsi2[1:3])
v3 <- sum(lpsi3[1:3])
...

```

From there, we can express the shape parameter of the gamma distribution (precisely the α 's here) as a function of covariates as in $\log(\alpha) = \beta_1 + \beta_2 x$ in the spirit of a generalized linear model with a gamma response.

5.4.2 Multinomial logit

Another possibility to build a prior that ensures the movement probabilities are between 0 and 1 and sum up to 1 is to extend the logit link we used for the CJS model in Section 4.8. Remember we had $\text{logit}(\phi) = \beta$, we specified a prior on β say $\beta \sim N(0, 1.5)$ then got a prior on ϕ by back-transforming β with $\phi = \text{logit}^{-1}(\beta)$.

Going back to our example with 3 sites, and focusing on the movement probabilities say, from site 1, we first choose a reference (or pivot) site, say 1, then $\log\left(\frac{\psi^{12}}{\psi^{11}}\right) = \beta_2$ and $\log\left(\frac{\psi^{13}}{\psi^{11}}\right) = \beta_3$. Interestingly, when exponentiated, the β 's here can be interpreted as the increase in the odds of moving versus staying on site resulting from a one-unit increase in the covariate. Any of the sites can be chosen to be the reference, this will not change the likelihood and you will get the same results. Now we specify a normal prior distribution for the β 's. Eventually, to back-transform, we use $\psi^{12} = \frac{\exp(\beta_2)}{1 + \exp(\beta_2) + \exp(\beta_3)}$ and $\psi^{13} = \frac{\exp(\beta_3)}{1 + \exp(\beta_2) + \exp(\beta_3)}$. The reference parameter, here ψ^{11} , is

calculated as $\psi^{11} = \frac{1}{1 + \exp(\beta_2) + \exp(\beta_3)}$, or simply as the complementary probability $\psi^{11} = 1 - \psi^{12} - \psi^{13}$.

Note that when there are only 2 sites instead of 3 or more, then the multinomial logit reduces to the logit link.

In NIMBLE, we write:

```
multisite <- nimbleCode({
  ...
  # transitions: multinomial logit
  for (i in 1:2){
    # normal priors on logit of all but one movement prob
    beta1[i] ~ dnorm(0, sd = 1.5)
    beta2[i] ~ dnorm(0, sd = 1.5)
    beta3[i] ~ dnorm(0, sd = 1.5)
    # constrain the transitions such that their sum is < 1
    psi1[i] <- exp(beta1[i]) / (1 + exp(beta1[1]) + exp(beta1[2]))
    psi2[i] <- exp(beta2[i]) / (1 + exp(beta2[1]) + exp(beta2[2]))
    psi3[i] <- exp(beta3[i]) / (1 + exp(beta3[1]) + exp(beta3[2]))
  }
  # reference movement probability
  psi1[3] <- 1 - psi1[1] - psi1[2]
  psi2[3] <- 1 - psi2[1] - psi2[2]
  psi3[3] <- 1 - psi3[1] - psi3[2]
  ...
})
```

Then we use these parameters (which now respect the constraints) to define the transition matrix:

```
multisite <- nimbleCode({
  ...
  # probabilities of state z(t+1) given z(t)
  gamma[1,1] <- phi1 * psi1[1]
  gamma[1,2] <- phi1 * psi1[2]
```

```

gamma[1,3] <- phi1 * psi1[3]
gamma[1,4] <- 1 - phi1
gamma[2,1] <- phi2 * psi2[1]
gamma[2,2] <- phi2 * psi2[2]
gamma[2,3] <- phi2 * psi2[3]
gamma[2,4] <- 1 - phi2
gamma[3,1] <- phi3 * psi3[1]
gamma[3,2] <- phi3 * psi3[2]
gamma[3,3] <- phi3 * psi3[3]
gamma[3,4] <- 1 - phi3
gamma[4,1] <- 0
gamma[4,2] <- 0
gamma[4,3] <- 0
gamma[4,4] <- 1
...

```

You may check that the results are very similar to those we obtained with the Dirichlet prior:

```

MCMCsummary(mcmc.multisite, round = 2)
##          mean    sd 2.5% 50% 97.5% Rhat n.eff
## p1      0.52  0.08 0.37 0.52  0.69 1.01   297
## p2      0.46  0.05 0.37 0.45  0.57 1.13   209
## p3      0.23  0.06 0.14 0.23  0.36 1.05   137
## phi1    0.60  0.05 0.50 0.60  0.70 1.00   403
## phi2    0.70  0.04 0.63 0.70  0.77 1.14   281
## phi3    0.77  0.06 0.64 0.77  0.89 1.05   199
## psi1[1] 0.74  0.06 0.62 0.74  0.83 1.00   727
## psi1[2] 0.22  0.05 0.13 0.22  0.33 1.03   795
## psi1[3] 0.04  0.03 0.01 0.04  0.12 1.05    88
## psi2[1] 0.07  0.02 0.04 0.07  0.11 1.01   655
## psi2[2] 0.83  0.04 0.74 0.84  0.90 1.04   153
## psi2[3] 0.10  0.04 0.04 0.09  0.19 1.03   122
## psi3[1] 0.03  0.02 0.01 0.02  0.07 1.01   794

```

```
## psi3[2] 0.22 0.05 0.13 0.21  0.32 1.02   477
## psi3[3] 0.76 0.05 0.64 0.76  0.85 1.02   444
```

Both the Dirichlet prior and the multinomial logit link give the same results, and there is not much difference in terms of runtime or quality of convergence, so you should use the option you feel the most comfortable with.

5.5 Sites may be states

So far, we have considered geographical locations (or sites) to refine the alive information when an animal is detected. However, it was quickly realized that sites could actually be states defined by physiology or behavior, hence opening up an avenue for applications of capture-recapture models in many fields of ecology.

Examples of states include:

- Epidemiological or disease states: sick/healthy, uninfect ed/infected/recovered;
- Morphological states: small/medium/big, light/medium/heavy;
- Life-history states: e.g. breeder/non-breeder, failed breeder, first-time breeder;
- Developmental 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.

In brief, states are individual, time-specific discrete covariates.

5.5.1 Titis data

To illustrate this section, we will consider data collected between 1942 and 1956 by Lance Richdale on the Sooty shearwaters (*Ardenna grisea*), also known as titis (Figure 5.4).



FIGURE 5.4: Sooty shearwater (**Ardenna grisea**). Credit: John Harrison.

You may see the data below:

```
titis <- read_csv2("dat/titis.csv",
                      col_names = FALSE)
titis %>%
  rename(year_1942 = X1,
         year_1943 = X2,
         year_1944 = X3,
         year_1949 = X4,
         year_1952 = X5,
         year_1953 = X6,
```

```

year_1956 = X7)
## # A tibble: 1,013 x 7
##   year_1942 year_1943 year_1944 year_1949 year_1952
##   <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 0         0         0         0         0
## 2 0         0         0         0         0
## 3 0         0         0         0         0
## 4 0         0         0         0         0
## 5 0         0         0         0         0
## 6 0         0         0         0         0
## 7 0         0         0         0         0
## 8 0         0         0         0         0
## 9 0         0         0         0         0
## 10 0        0         0         0         0
## # i 1,003 more rows
## # i 2 more variables: year_1953 <dbl>, year_1956 <dbl>

```

In total, 1013 titis were captured, marked and recaptured on a small colony on Whero Island in southern New Zealand. These data were previously analyzed by Richard Scofield 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. For simplicity, we pooled all alive states (except 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). Because burrows were not checked before hatching, some birds in the category NB might have already failed. Therefore birds in the breeder state (B) should be seen as successful breeders, and those in the NB state as nonbreeders plus prebreeders and failed breeders.

In summary, we code the states as 1 for alive and breeding, 2 for alive and non-breeding and 3 for dead. To make the modelling process more self-explanatory, we will use letters B, NB and D but you should keep in mind that these are actually numbers. Observations are non-detected

coded as 1, and detected as breeder or non-breeder coded as 2 and 3 respectively.

The aim here is to study life-history trade-offs. Specifically, we ask the questions: Does breeding affect survival? Does breeding in current year affect breeding next year?

5.5.2 The AS model for states

Basically, the AS model with states is the same model as in the geese example with two sites, see Sections 5.2 and 5.3.

If we let ϕ^B and ϕ^{NB} be the survival probabilities of breeders and non-breeders, ψ^{NBB} the probability of becoming breeder for a non-breeder, and ψ^{BNB} the probability of skipping reproduction, then the transition matrix is:

$$\times = \begin{pmatrix} z_t = B & z_t = NB & z_t = D \\ \bar{\phi}^B(1 - \psi^{BNB}) & \bar{\phi}^B\psi^{BNB} & 1 - \bar{\phi}^B \\ \phi^{NB}\psi^{NBB} & \phi^{NB}(1 - \psi^{NBB}) & 1 - \phi^{NB} \\ 0 & 0 & 1 \end{pmatrix} \begin{array}{l} z_{t-1} = B \\ z_{t-1} = NB \\ z_{t-1} = D \end{array}$$

The costs or reproduction would reflect in future reproduction if breeders have a lower probability of breed next year than non-breeders $\psi^{BB} = 1 - \psi^{BNB} < \psi^{NBB}$ or in survival if the survival of breeders is lower than that of non-breeders $\phi^B < \phi^{NB}$.

The observation matrix is:

$$\times = \begin{pmatrix} y_t = 1 & y_t = 2 & y_t = 3 \\ 1 - p^B & p^B & 0 \\ 1 - p^{NB} & 0 & p^{NB} \\ 1 & 0 & 0 \end{pmatrix} \begin{array}{l} z_t = B \\ z_t = NB \\ z_t = D \end{array}$$

where p^B and p^{NB} are the detection probabilities of non-breeders and breeders respectively.

5.5.3 NIMBLE implementation

We first write the NIMBLE code, which is exactly the same as in the geese example with two sites, see Section 5.3.

First some definitions which we have as comments in the code:

```
multistate <- nimbleCode({
  # -----
  # Parameters:
  # B is for breeder, NB for non-breeder
  # 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: detection probability B
  # pNB: detection 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
  # -----
  ...
})
```

Then the priors:

```
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)

...

```

The transition matrix:

```

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
  ...
}

```

The observation matrix:

```

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)
...

```

And the likelihood:

```

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])
    }
  }
})

```

We run NIMBLE and get the following results:

```

MCMCsummary(mcmc.multistate, round = 2)
##          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

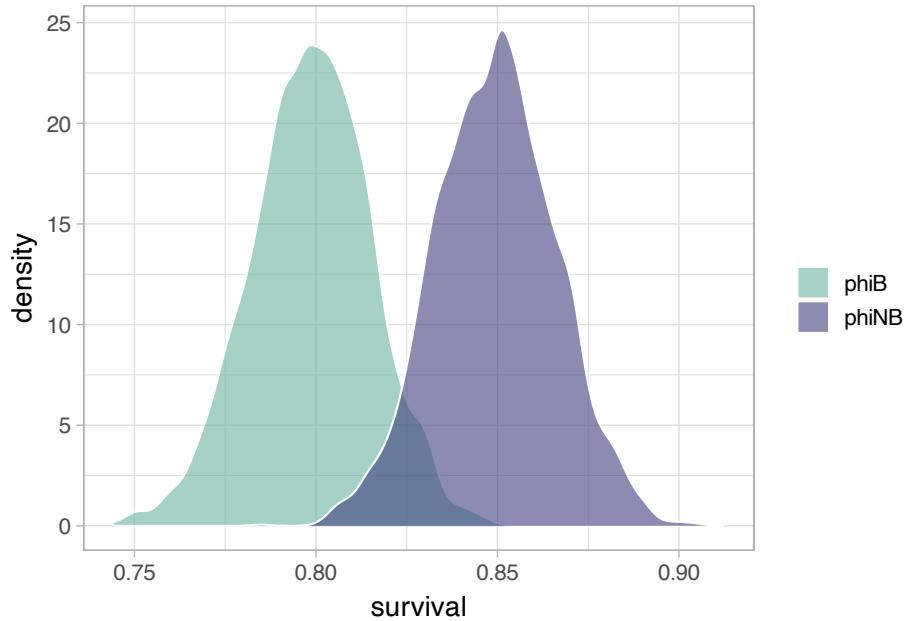
```

Non-breeder individuals seem to have a survival higher than breeder individuals, suggesting a trade-off between reproduction and survival. Let's compare graphically the survival of breeder and non-breeder individuals. First we gather the values generated for ϕ^B and ϕ^{NB} for the two chains:

```
phiB <- c(mcmc.multistate$chain1[, "phiB"], mcmc.multistate$chain2[, "phiB"])
phiNB <- c(mcmc.multistate$chain1[, "phiNB"], mcmc.multistate$chain2[, "phiNB"])
df <- data.frame(param = c(rep("phiB", length(phiB)),
                           rep("phiNB", length(phiB))),
                  value = c(phiB, phiNB))
```

Then, we plot the two posterior distributions:

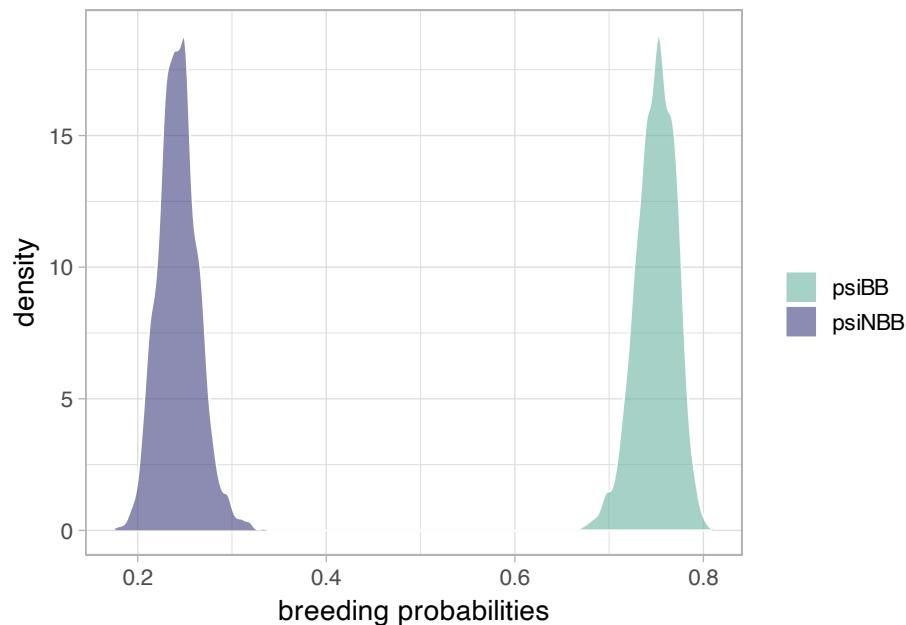
```
df %>%
  ggplot(aes(x = value, fill = param)) +
  geom_density(color = "white", alpha = 0.6, position = 'identity') +
  scale_fill_manual(values = c("#69b3a2", "#404080")) +
  labs(fill = "", x = "survival")
```



There is little overlap between the two distributions, suggesting an actual trade-off. A formal test of the trade-off would consist in fitting the model with survival irrespective of the state, and compare its WAIC value to the model we just fitted.

What about a potential trade-off on reproduction?

```
psiBNB <- c(mcmc.multistate$chain1[, "psiBNB"], mcmc.multistate$chain2[, "psiBNB"])
psiBB <- 1 - psiBNB
psiNBB <- c(mcmc.multistate$chain1[, "psiNBB"], mcmc.multistate$chain2[, "psiNBB"])
df <- data.frame(param = c(rep("psiBB", length(phiB)),
                           rep("psiNBB", length(phiB))),
                  value = c(psiBB, psiNBB))
df %>%
  ggplot(aes(x = value, fill = param)) +
  geom_density(color = "white", alpha = 0.6, position = 'identity') +
  scale_fill_manual(values = c("#69b3a2", "#404080")) +
  labs(fill = "", x = "breeding probabilities")
```



There is no overlap whatsoever, so the two transition probabilities are

clearly different. Interestingly, breeder individuals do much better than non-breeder individuals. This failure at detecting a trade-off is probably due to individual heterogeneity that should be accounted for. We will do just that in a case study in Section 8.2.

5.6 Issue of local minima

In the frequentist approach, we use the maximum likelihood theory to estimate parameters. The maximum likelihood estimates are the values that get you to the maximum of the model likelihood. To find out the maximum of the likelihood, we use iterative optimization algorithms (e.g. the default method is that of Nelder and Mead in the R `optim()` function). However, sometimes, our model likelihood contains several maxima and there is no guarantee that the algorithms will find the global maximum corresponding to the maximum likelihood estimates, and it may get stuck in a local maximum. Let's illustrate this issue with some simulated data that were kindly provided by Jérôme Dupuis. We consider 2 sites (or alive states), say 1 and 2, and 7 sampling occasions. The survival probability is constant $\phi = 1$ as well as the detection probability $p = 0.6$. The probability of moving from 1 to 2 is $\psi^{12} = 0.6$ and $\psi^{21} = 0.85$ in the opposite direction. Here are the encounter histories of the 27 individuals that were simulated by Jérôme:

```
dat <- matrix(c(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,
               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,
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, 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),
byrow = T,
ncol = 7)

```

In Figure 5.5, we provide an illustration of the influence of the choice of initial values when trying to maximize the likelihood, or rather to minimize the deviance (which is minus two times the log of the likelihood). The black curve is the what we called the profile deviance for ψ^{21} . Profiling the deviance consists in taking a slice of it in the direction of a parameter of interest and treating the other parameters as nuisance parameters. In our example, we set ψ^{21} to a value (on the x-axis) and minimize the deviance (on the y-axis) with respect to the other parameters. There are two minima, but only the global minimum (corresponding to the lowest value of deviance) corresponding to ψ^{21} around 0.8 is

of interest to us. The thing is that if you start your optimization algorithm by picking value in the red area, then it will get stuck in the local minimum and will tell you the maximum likelihood estimate of ψ^{21} is around 0.35, which is obviously far from the value we used to simulate the data. In contrast, if you pick initial values in the green area, then the algorithm will converge to the global minimum.

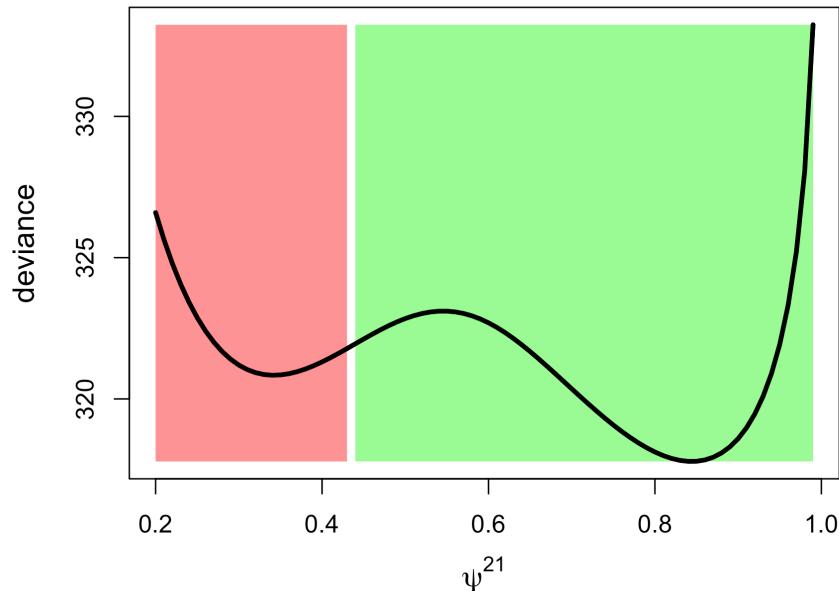
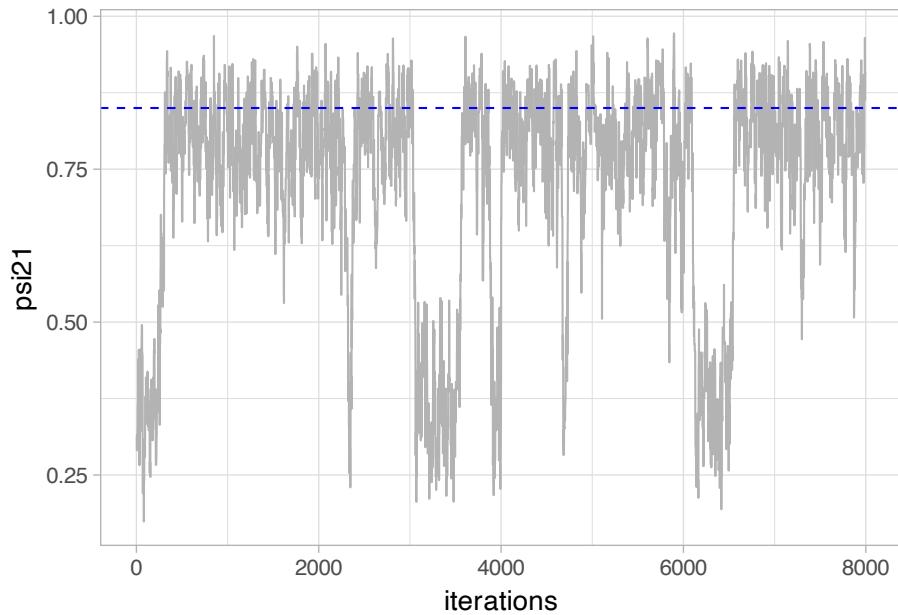


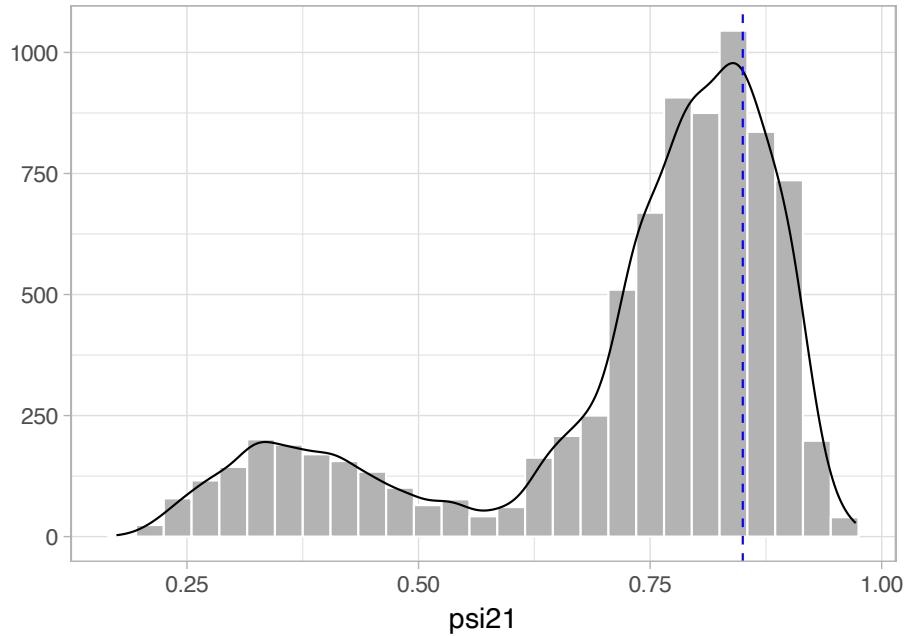
FIGURE 5.5: Influence of the choice of initial values on the convergence to the global minimum of the deviance illustrated with simulated data. The black curve is the profile deviance of the probability to move from site 2 to 1. If an initial value is picked in the red area, we end up in the local minimum while if it is picked in the green area, then we get the global minimum which corresponds to the maximum likelihood estimate.

You might argue that this is a problem of the optimization algorithm and therefore inherent to the frequentist approach. Well, it turns out that MCMC algorithms are not immune to the issue. If you fit the

AS model with constant parameters to the simulated data, here is the trace for the probability of moving from 2 to 1:



Clearly, there are two regimes. The chain spends most of its time around high values of ψ^{21} close to the true value represented by the blue dashed line. But sometimes, the chain jumps to values around 0.3-0.4. This behavior translates into two modes in the posterior distribution for ψ^{21} where the mode on the right is closer to the truth represented by the dashed blue vertical line:



The issue of local minima is a difficult problem. How to get out of this problematic situation? In the frequentist approach, the trick is to fit your model several times with different initial values each time, hoping that you'll get to fall in the green area somehow as in Figure 5.5. In the Bayesian approach, the key to handle distributions with multiple modes is to sample the posteriors efficiently. Assuming the chains we run in NIMBLE spend more time in the region of the parameter space corresponding to the global minimum, then we recommend using the median or the mode to summarize the posterior distribution. In the simulated example, we get a median of 0.79 for ψ^{21} , not too bad given that the data were simulated with a value of 0.85 for that parameter:

```
MCMCsummary(mcmc.multisite, round = 2)
##      mean    sd 2.5% 50% 97.5% Rhat n.eff
## p     0.58  0.03 0.51 0.58  0.65 1.01  1866
## phi   0.99  0.01 0.98 1.00  1.00 1.01   691
## psi12 0.53  0.14 0.21 0.56  0.72 1.05    69
## psi21 0.72  0.18 0.28 0.79  0.92 1.04    37
```

As a general advice, we recommend to always inspect the trace plots to

find out whether you have posterior distributions with multiple modes which would suggest local minima.

5.7 Uncertainty

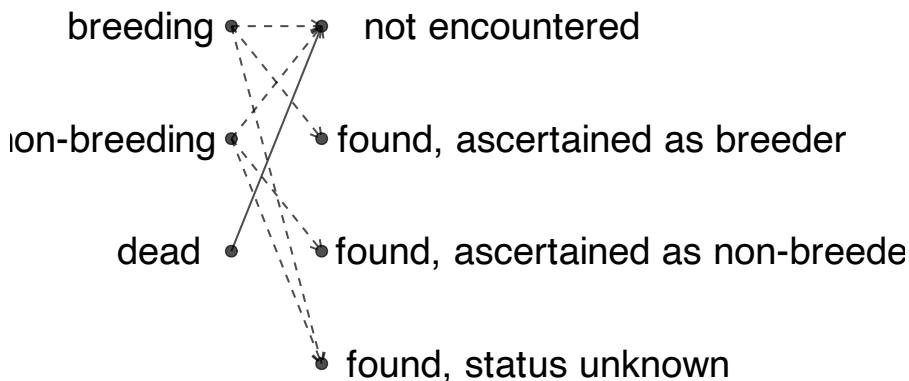
In the AS model, we assume that we can without a doubt assign a site or a state to an animal whenever it is detected. But this is not always the case. For example, when the breeding status in mammals or birds is ascertained based on the presence of offspring or eggs, we are uncertain of whether a female is breeding or not when the offspring or eggs are not seen. Another example is when the epidemiological status in mammals or birds is ascertained based on some tests run on some animals when captured, we are uncertain whether these animals are healthy or sick when detected from distance without possibility to manipulate them for testing. In this section we will cover the extension of the AS model to uncertain states through two examples, one on breeding states and the other on disease states. We will cover other examples in the part on Case studies, in particular in Chapters 8 and 7.

5.7.1 Breeding states

We will revisit the titis example 5.5 and try to assess life-history trade-offs while accounting for uncertainty in breeding status. We still have 3 states, which are alive and breeding, alive and non-breeding and dead. With regard to observations, a bird may be not encountered. It may also be encountered, but in contrast with our previous analysis of the titis data, we don't know its state for sure. It may be found and ascertained (or classified) as breeder. It may be found and ascertained as non-breeder. It may be found be we are unable to determine whether it's breeding or non-breeding.

How do the states generate the observations?

States Observations



Each alive state can generate 3 observations. The only deterministic link is that between the dead state and the observation non-encountered, because if a bird is dead, it cannot be detected for sure.

Let's specify the model. First thing we need, and it's a big difference with the AS model, we need initial state probabilities because we cannot assign states to individuals with certainty. We write down the probability for each state at first encounter, or the vector of initial state probabilities:

$$\pi = \begin{pmatrix} z_t = B & z_t = NB & z_t = D \\ \pi^B & 1 - \pi^B & 0 \end{pmatrix}$$

where π^B is the probability that a newly encountered individual is a breeder, and $\pi^{NB} = 1 - \pi^B$ is the probability that a newly encountered individual is a non-breeder (the complementary probability of π^B). The probability of being dead at first encounter is 0 (a bird is alive when it is first encountered).

Now the transition matrix, this is the easy part as it doesn't change. We have:

$$\times = \begin{pmatrix} z_t = B & z_t = NB & z_t = D \\ \bar{\phi}^B(1 - \psi^{BNB}) & \bar{\phi}^B\psi^{BNB} & 1 - \bar{\phi}^B \\ \phi^{NB}\psi^{NBB} & \phi^{NB}(1 - \psi^{NBB}) & 1 - \phi^{NB} \\ 0 & 0 & 1 \end{pmatrix} \begin{matrix} z_{t-1} = B \\ z_{t-1} = NB \\ z_{t-1} = D \end{matrix}$$

where ϕ^B is the breeder survival, ϕ_{NB} that of non-breeders, ψ^{BNB} is the probability for an individual breeding a year to be a non-breeder the next year, and ψ^{NBB} is the probability for an non-breeder individual to breeder the next year.

Last, the observation matrix. The main difference between multisite/multistate and multievent models is here, in the observation parameters. Besides p^B the detection probability of breeders and p^{NB} that of non-breeders, we introduce two new parameters: β^B is the probability to correctly assign an individual that is in state B to state B, and β^{NB} is the probability to correctly assign an individual that is in state NB to state NB. The complementary of these β parameters are often called false positive probabilities. We put everything in a matrix, as usual. In rows we have the states: breeding, non-breeding and dead. In columns, at the same occasion, we have the observations: non-detected, detected and ascertained B, detected and ascertained NB, and detected but state unknown:

$$\times = \begin{pmatrix} y_t = 1 & y_t = 2 & y_t = 3 & y_t = 4 \\ 1 - p^B & p^B\beta^B & 0 & p^B(1 - \beta^B) \\ 1 - p^{NB} & 0 & p^{NB}\beta^{NB} & p^{NB}(1 - \beta^{NB}) \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} z_t = B \\ z_t = NB \\ z_t = D \end{matrix}$$

For example, the probability of being detected and assigned to state B, given that you're in state B is the product of p^B the detection probability in B and β^B the probability of correctly assigning a breeding individual to state B.

At first encounter, all individuals are captured, but you still need to assign them a state. This means that we should set $p^B = p^{NB} = 1$ and use:

$$\begin{pmatrix} y_{t=\text{first}} = 1 & y_{t=\text{first}} = 2 & y_{t=\text{first}} = 3 & y_{t=\text{first}} = 4 \\ 0 & \beta^B & 0 & (1 - \beta^B) \\ 0 & 0 & \beta^{NB} & (1 - \beta^{NB}) \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} z_{t=\text{first}} = B \\ z_{t=\text{first}} = NB \\ z_{t=\text{first}} = D \end{matrix}$$

To implement this model in NIMBLE, we start by using comments to define the parameters, states and observations in the header of the code:

```
multievent <- 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
  # piB prob. of being in initial state breeder
  # betaNB prob to ascertain the breeding status of an individual encountered as non-breeder
  # betaB prob to ascertain the breeding status of an individual encountered as breeder
  # -----
  # States (z):
  # 1 alive B
  # 2 alive NB
  # 3 dead
  # Observations (y):
  # 1 = non-detected
  # 2 = seen and ascertained as breeder
  # 3 = seen and ascertained as non-breeder
  # 4 = not ascertained
  # -----
  ...
})
```

Then we assign prior to all parameters to be estimated. Because we

deal with probabilities, the uniform distribution between 0 and 1 will do the job:

```
multievent <- 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)
  piB ~ dunif(0, 1)
  betaNB ~ dunif(0, 1)
  betaB ~ dunif(0, 1)
  ...
})
```

Now we write the vector of initial state probabilities:

```
multievent <- nimbleCode({
  ...
  # vector of initial stats probs
  delta[1] <- piB # prob. of being in initial state B
  delta[2] <- 1 - piB # prob. of being in initial state NB
  delta[3] <- 0 # prob. of being in initial state dead
  ...
})
```

The transition matrix:

```
multievent <- 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
...

```

And the observation matrix:

```

multievent <- nimbleCode({
...
# probabilities of y(t) given z(t)
omega[1,1] <- 1 - pB           # Pr(alive B t -> non-detected t)
omega[1,2] <- pB * betaB       # Pr(alive B t -> detected B t)
omega[1,3] <- 0                # Pr(alive B t -> detected NB t)
omega[1,4] <- pB * (1 - betaB) # Pr(alive B t -> detected U 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 * betaNB     # Pr(alive NB t -> detected NB t)
omega[2,4] <- pNB * (1 - betaNB) # Pr(alive NB t -> detected U 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)
omega[3,4] <- 0                # Pr(dead t -> detected U t)
...

```

The observation matrix at first encounter:

```

multievent <- nimbleCode({
...

```

```

# probabilities of y(first) given z(first)
omega.init[1,1] <- 0          # Pr(alive B t = first -> non-detected t = first)
omega.init[1,2] <- betaB      # Pr(alive B t = first -> detected B t = first)
omega.init[1,3] <- 0          # Pr(alive B t = first -> detected NB t = first)
omega.init[1,4] <- 1 - betaB  # Pr(alive B t = first -> detected U t = first)
omega.init[2,1] <- 0          # Pr(alive NB t = first -> non-detected t = first)
omega.init[2,2] <- 0          # Pr(alive NB t = first -> detected B t = first)
omega.init[2,3] <- betaNB    # Pr(alive NB t = first -> detected NB t = first)
omega.init[2,4] <- 1 - betaNB # Pr(alive NB t = first -> detected U t = first)
omega.init[3,1] <- 1          # Pr(dead t = first -> non-detected t = first)
omega.init[3,2] <- 0          # Pr(dead t = first -> detected N t = first)
omega.init[3,3] <- 0          # Pr(dead t = first -> detected NB t = first)
omega.init[3,4] <- 0          # Pr(dead t = first -> detected U t = first)

...

```

Eventually, we get to the likelihood:

```

multievent <- nimbleCode({
  ...
  # likelihood
  for (i in 1:N){
    # latent state at first capture
    z[i,first[i]] ~ dcat(delta[1:3])
    y[i,first[i]] ~ dcat(omega.init[z[i,first[i]],1:4]) # obs at first encounter
    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],1:4])
    }
  }
})

```

The only change is in the line $y[i, \text{first}[i]] \sim$

`dcat(omega.init[z[i],first[i]],1:4])` where we use the observation matrix at first encounter.

We run NIMBLE and get the following numerical summaries for the model parameters:

```
MCMCsummary(mcmc.multievent, round = 2)
##          mean    sd 2.5% 50% 97.5% Rhat n.eff
## betaB 0.19 0.01 0.16 0.19  0.21 1.01   332
## betaNB 0.76 0.05 0.66 0.76  0.86 1.01    65
## pB     0.56 0.03 0.51 0.56  0.62 1.06   229
## pNB    0.60 0.04 0.53 0.60  0.67 1.03   142
## phiB   0.81 0.02 0.78 0.81  0.85 1.01   312
## phiNB 0.84 0.02 0.80 0.84  0.87 1.00   354
## piB    0.71 0.03 0.66 0.71  0.76 1.02   115
## psiBNB 0.23 0.02 0.18 0.22  0.27 1.00   214
## psiNBB 0.25 0.04 0.17 0.25  0.34 1.00    95
```

Breeders are difficult to assign to the correct state β^B , while non-breeders are relatively well classified as non-breeders β^{NB} .

There is again no cost of current reproduction on future reproduction. We no longer detect a cost of breeding on survival.

5.7.2 Disease states

Let's have a look to another example. We consider a system of an emerging pathogen *Mycoplasma gallisepticum* Edward and Kanarek and its host the house finch, *Carpodacus mexicanus* Müller. The pathogen causes moderate to severe eye swelling (see Figure 5.6).

Here we're asking whether the presence of clinical signs of the pathogen influences survival. The objective of the study was also to quantify infection (moving from state healthy to state ill) and recovery (moving from state ill to state healthy) probabilities. The birds were captured via mist nets and marked with individually identifiable color bands over three years. The data were kindly provided by Paul Conn



FIGURE 5.6: A house finch with a heavy infection caused by conjunctivitis. Credit: Jim Mondok.

and Evan Cooch. The difficulty was that ascertaining the disease status of birds seen from distance was difficult since determining the presence of the pathogen was only possible when the bird's eyes were clearly visible. In this context, how to study the dynamics of the disease?

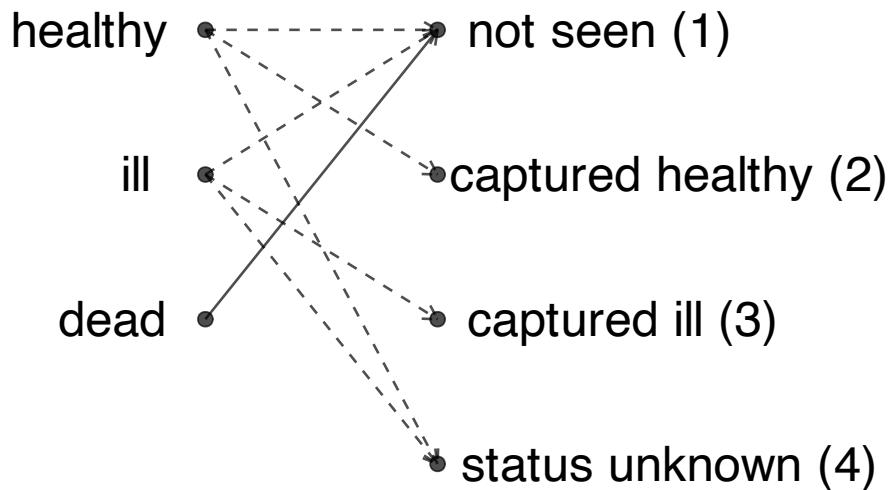
First, we think of states and observations. We have:

- 3 states
 - healthy (H)
 - ill (I)
 - dead (D)
- 4 observations
 - not seen (1)
 - captured healthy (2)
 - captured ill (3)

- health status unknown, i.e. seen at distance (4)

How do the states generate observations?

States Observations



Clearly, this is the same model as in the previous section on titis, Section 5.7.1, in which loosely speaking we replace breeder by healthy and non-breeder by ill.

The vector of initial state probabilities is:

$$\times = \begin{pmatrix} z_t = H \\ \pi^H \\ z_t = I \\ 1 - \pi^H \\ z_t = D \\ 0 \end{pmatrix}$$

where π^H is the probability that a newly encountered individual is healthy, and $\pi^I = 1 - \pi^H$ is the probability that a newly encountered individual is ill.

The transition matrix is:

$$\times = \begin{pmatrix} z_t = H & z_t = I & z_t = D \\ \bar{\phi}^H(1 - \psi^{HI}) & \bar{\phi}^H\psi^{HI} & 1 - \bar{\phi}^H \\ \phi^I\psi^{IH} & \phi^I(1 - \psi^{IH}) & 1 - \phi^I \\ 0 & 0 & 1 \end{pmatrix} \begin{array}{l} z_{t-1} = H \\ z_{t-1} = I \\ z_{t-1} = D \end{array}$$

where ϕ^H is the survival probability of healthy individuals, ϕ^I the survival probability of ill individuals, ψ^{HI} the probability of getting ill (infection rate) and ψ^{IH} the probability of recovering from the disease (recovery rate).

Image you'd like to model the dynamic of an incurable disease, the transition matrix would be modified by having $\psi^{IH} = 0$, and once a bird gets ill, it remains ill $\psi^{II} = 1 - \psi^{IH} = 1$. Therefore we would have:

$$\times = \begin{pmatrix} z_t = H & z_t = I & z_t = D \\ \bar{\phi}^H(1 - \psi^{HI}) & \bar{\phi}^H\psi^{HI} & 1 - \bar{\phi}^H \\ 0 & \phi^I & 1 - \phi^I \\ 0 & 0 & 1 \end{pmatrix} \begin{array}{l} z_{t-1} = H \\ z_{t-1} = I \\ z_{t-1} = D \end{array}$$

For analysing the house finch data, we allow recovering from the disease. The observation matrix is:

$$\times = \begin{pmatrix} y_t = 0 & y_t = 1 & y_t = 2 & y_t = 3 \\ 1 - p^H & p^H\beta^H & 0 & p^H(1 - \beta^H) \\ 1 - p^I & 0 & p^I\beta^I & p^I(1 - \beta^I) \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{array}{l} z_t = H \\ z_t = I \\ z_t = D \end{array}$$

where β^H is the probability to assign a healthy individual to state H, and β^I is the probability to assign a sick individual to state I. p^H is the detection probability of healthy individuals, p^I that of sick individuals.

Using the code we developed for the titis example, we get the following results on the finches by running NIMBLE:

```
##      mean    sd 2.5% 50% 97.5% Rhat n.eff
## betaH 0.99 0.01 0.97 0.99  1.00 1.01 1421
## betaI 0.05 0.01 0.03 0.05  0.08 1.00 6477
```

```

## pH    0.17 0.02 0.13 0.17  0.22 1.01   331
## pI    0.58 0.10 0.41 0.57  0.80 1.04   220
## phiH  0.88 0.02 0.84 0.88  0.92 1.01   360
## phiI  0.99 0.01 0.96 0.99  1.00 1.00  1004
## pi    0.96 0.01 0.93 0.96  0.98 1.00  4190
## psiHI 0.22 0.04 0.16 0.22  0.32 1.02   311
## psiIH 0.46 0.08 0.32 0.45  0.63 1.02   392

```

Healthy individuals are correctly assigned (β^H is almost 1), while infected individuals are difficult to ascertain (β^I is around 0.05). Unexpectedly, ill birds have a better survival than healthy individuals (compare ϕ^I and ϕ^H). Infection rate (ψ^{HI}) is 22%, recovery rate is 46% (ψ^{IH}).

5.8 Summary

- The AS model is a HMM that extends the CJS model by allowing the estimation of movements between sites (e.g. geographical locations) or transitions between states (e.g. breeding status). This flexibility allows addressing all sorts of questions in ecology and evolution.
- Covariates can be considered, and appropriate priors or link functions need to be used when there are more than 2 sites or 2 alive states.
- Model comparison can be achieved with the WAIC and the goodness of fit of the AS model to capture-recapture data can be assessed with classical procedures or posterior predictive checks.
- Importantly, the HMM framework allows to account for uncertainty when assigning states to individuals.
- The models covered in this chapter haven been called multistratum/strata models, multisite models (section 5.2), multistate models (section 5.5) and multievent models (section 5.7). These models are all HMMs.

5.9 Suggested reading

- The AS model was introduced in [Arnason \[1972\]](#), [Arnason \[1973\]](#) and [Schwarz et al. \[1993\]](#). Very soon clever folks realized that sites could be replaced by states as in [Nichols et al. \[1992\]](#) and [Nichols et al. \[1994\]](#). For a review of models with sites and states, see [Lebreton et al. \[2009\]](#).
- The geese data were analyzed in [Hestbeck et al. \[1991\]](#) and [Brownie et al. \[1993\]](#), and the titis data in [Scofield et al. \[2001\]](#). The house finches data were analyzed in [Faustino et al. \[2004\]](#) and [Conn and Cooch \[2009\]](#) (see also [Cooch et al. \[2012\]](#)). Check out [Santoro et al. \[2014\]](#), [Marescot et al. \[2018\]](#) and [Ollivier et al. \[2023\]](#) for other examples in disease ecology.
- Section 5.6 on local minima was inspired by chapter 10 of [Cooch and White \[2017\]](#).
- Classical goodness of fit tests are reviewed in [Pradel et al. \[2005\]](#). See also [Pradel et al. \[2003\]](#) for tests specifically designed for multisite/multistate models.
- Models with uncertainty were introduced in [Pradel \[2005\]](#). [Dupuis \[1995\]](#) had a similar idea for the AS model. For a review, see [Gimenez et al. \[2012\]](#).



Part III

Case studies



Introduction

This third part `case_studies` 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. For each case study, I recall the ecological question, then build the model step by step and conclude by discussing the results. The code and data are available [saywhere](#).



6

Covariates

WORK IN PROGRESS

6.1 Covariate selection with reversible jump MCMC

RJMCMC in [Gimenez et al. \[2009b\]](#) on Common blackbirds or [Gimenez et al. \[2009a\]](#) on White stork.

As an illustration, we use data on the white stork *Ciconia ciconia* population in Baden Wurttemberg (Germany), consisting of 321 capture histories of individuals ringed as chicks between 1956 and 1971. From the 60's to the 90's, all Western European stork populations were declining [Bairlein \[1991\]](#). This trend was likely the result of reduced food availability [Schaub et al. \[2005\]](#) caused by severe droughts observed in the wintering ground of storks in the Sahel region. This hypothesis has been examined in several studies ([Kanyamibwa et al. \[1990\]](#) and [Barbraud et al. \[1999\]](#)).

Check out https://r-nimble.org/nimbleExamples/RJMCMC_example.html and <https://r-nimble.org/variable-selection-in-nimble-using-reversible-jump-mcmc>.

Somewhere explain how to use if-else in model code to consider alternative models, w/ some covariate in/out. Avoids rewriting all models, we see what's changed, and it avoids errors. Example:

```
if(covariate){  
  logit(survival[t]) <- beta[1] + beta[2] *x[t]
```

```
}else{  
  logit(survival[t]) <- beta[1]  
}#ifelse
```

then specify “covariate=TRUE/FALSE”.

6.2 Missing values

Work on missing values by [Bonner and Schwarz \[2006\]](#) (see [Gimenez et al. \[2009a\]](#)) and [Langrock and King \[2013\]](#) and [Worthington et al. \[2015\]](#). See also [Rose et al. \[2018\]](#).

6.3 Sex uncertainty

[Pradel et al. \[2008\]](#) and [Genovart et al. \[2012\]](#)

6.4 Nonlinearities

Splines à la [Gimenez et al. \[2006\]](#), possibly w/ jagam <https://rdrr.io/cran/mgcv/src/R/jagam.r>.

6.5 Spatial

3D Splines as in [Péron et al. \[2011\]](#). (I)CAR as in [Saracco et al. \[2010\]](#) (see <https://github.com/Andrew9Lawson/Bayesian-DM-code-examples>,

https://github.com/Andrew9Lawson/Bayesian_DM_Nimble_code/tree/ICAR-and-other-code and https://r-nimble.org/html_manual/cha-spatial.html for NIMBLE implementation). Add RSR Khan and Calder [2022] (see Jags code at <https://gist.github.com/oliviergimenez/0d5519654adef09060581eb49e2128ce>).



7

Lack of fit

WORK IN PROGRESS

7.1 Individual heterogeneity

On wolf, see [Cubaynes et al. \[2010\]](#), [Gimenez and Choquet \[2010\]](#), or go full non-parametric w/ [Turek et al. \[2021\]](#). See [Pradel \[2009\]](#) for black-headed gull example.

Our example is about individual heterogeneity and how to account for it with HMMs. Gray wolf is a social species with hierarchy in packs which may reflect in species demography. As an example, we'll work with gray wolves.

Gray wolf is a social species with hierarchy in packs which may reflect in demography. Shirley Pledger in a series of papers developed heterogeneity models in which individuals are assigned in two or more classes with class-specific survival/detection probabilities. [Cubaynes et al. \[2010\]](#) used HMMs to account for heterogeneity in the detection process due to social status, see also [Pradel \[2009\]](#). Dominant individuals tend to use path more often than others, and these paths are where we look for scats.

Individual heterogeneity

- 3 states
- alive in class 1 (A1)
- alive in class 2 (A2)



FIGURE 7.1: Dominance in wolves.

- dead (D)
- 2 observations
- not captured (1)
- captured (2)

Vector of initial state probabilities

$$\times = \begin{pmatrix} z_t = A1 & z_t = A2 & z_t = D \\ \pi & 1 - \pi & 0 \end{pmatrix}$$

π is the probability of being alive in class 1. $1 - \pi$ is the probability of being in class 2.

Transition matrix

$$\times = \begin{pmatrix} z_t = A1 & z_t = A2 & z_t = D \\ \phi & 0 & 1 - \phi \\ 0 & \phi & 1 - \phi \\ 0 & 0 & 1 \end{pmatrix} z_{t-1} = A1 \\ z_{t-1} = A2 \\ z_{t-1} = D$$

ϕ is the survival probability, which could be made heterogeneous.

Transition matrix, with change in heterogeneity class

$$\times = \begin{pmatrix} z_t = A1 & z_t = A2 & z_t = D \\ \bar{\phi}(1 - \psi^{12}) & \phi\psi^{12} & 1 - \phi \\ \phi\psi^{21} & \phi(1 - \psi^{21}) & 1 - \phi \\ 0 & 0 & 1 \end{pmatrix} z_{t-1} = A1 \\ z_{t-1} = A2 \\ z_{t-1} = D$$

ψ^{12} is the probability for an individual to change class of heterogeneity, from 1 to 2. ψ^{21} is the probability for an individual to change class of heterogeneity, from 2 to 1.

Observation matrix

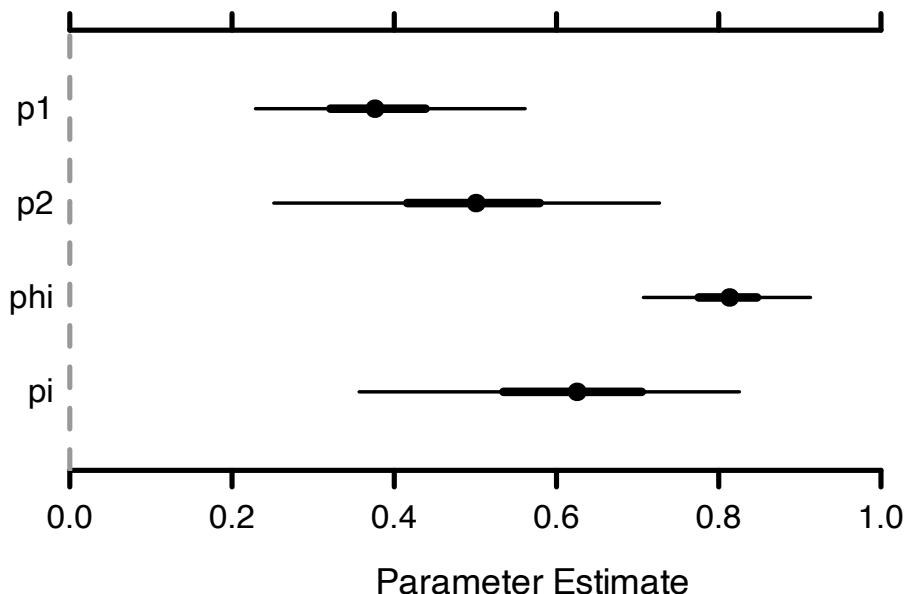
$$\times = \begin{pmatrix} y_t = 1 & y_t = 2 \\ 1 - p^1 & p^1 \\ 1 - p^2 & p^2 \\ 1 & 0 \end{pmatrix} z_t = A1 \\ z_t = A2 \\ z_t = D$$

p^1 is detection for individuals in class 1, and p^2 that of individuals in class 2.

Results

```
##      mean    sd 2.5% 50% 97.5% Rhat n.eff
## p1  0.38 0.09 0.23 0.38  0.56 1.04   210
## p2  0.50 0.12 0.25 0.50  0.73 1.01   229
## phi 0.81 0.05 0.71 0.81  0.91 1.04   317
## pi  0.62 0.12 0.36 0.63  0.83 1.02   164
```

We have lowly detectable individuals (class A1 with p^1) in proportion 62%. And highly (or so) detectable individuals (class A2 with p^2) in proportion 38%. Note that interpretation of classes is made a posteriori. Survival is 81%.



From the simulations I run, seems like the categorical sampler on latent states gets stuck in places that depend on initial values. Changing for the slice sampler improves thing a bit, but not that much. Only option is to get rid of the states and use the marginalized likelihood. Nice illustration of the use of simulations (to check model is doing ok, estimated are valid, etc.), changing samplers, nimbleEcology, NIMBLE functions, etc.

You may consider more classes, and select among models, see [Cubaynes et al. \[2012\]](#). You may also go for a non-parametric approach and let the data tell you how many classes you need. This is relatively easy to do in NIMBLE, see [Turek et al. \[2021\]](#). More about individual heterogeneity in [Gimenez et al. \[2018a\]](#).

Bottom line, the way this model is written is completely destined for failure. The state-space formulation that's written, with the two different groups for individuals (high detection and low detection) sets the model up for failure, and it will never work correctly when written this way. Briefly:

For any individual that is seen at any time $t > 1$ (that is, individuals seen again after the first sighting), the detection history looks something like: 1 0 0 1 (1 represents detection). The initial values for the latent z state are either : 1 1 1 1 or 2 2 2 2 2 putting that individual into one of the two groups (1 or 2 always). when sampling, the latent z can *never* transition to the other group, from group 2 to group 1, or from group 1 to group 2. It will be stuck where ever it started. If the categorical sampler tries to change the *final* state from the initial value of (say) 2 to 1, then this transition is deemed to be impossible by the prior (defined by the gamma state transition matrix), since the state at the previous time was 2, and 2 in one period (the previous period) does not permit a state of 1 in the next time period. Similarly, if the first (or any intermediate) value of z attempts to transition from (say) 2 to 1, then the *following* state is still 1, and that dependency does not allow the state in question to change to 1, because a state of 1 cannot have the next state be 2. Even if some “dead” states (3’s) are added to the end of the z vector over the course of MCMC sampling, and say z becomes: 1 1 1 1 3 3 The 3’s can never propagate “earlier” than shown here (since there are detections at $t=1$ and $t=4$, so the individual cannot be in state 3 at time $t=4$), so the problem described above will always be the case, and this individual will *always* remain in group 1, no matter how long you run the MCMC.

The only time an individual (with the model written as such) could change between groups ($1 \rightarrow 2$, or $2 \rightarrow 1$) from their initial group assignment of z_{init} , would be if the individual is *only* observed on the first time period, the detection history is: 1 0 0 0 0 0, Then say the initial value for z is: 1 1 1 1 1 1 (always in group 1), then the sampler at some point could begin transitioning the final 1 into state 3 (dead), so after an MCMC iteration, the z for this individual could be: 1 1 1 1 1 3, then if we're lucky, the sampler on the final 1 would some time change it to a 3: 1 1 1 1 3 3 then this could happen again later: 1 1 1 3 3 3 and again: 1 1 3 3 3 and once again: 1 3 3 3 3 3 and *only now*, finally, if we're lucky, the sampler operating on the first value of the z vector could change this group assignment from 1 to 2: 2 3 3 3 3 3 And that's the only situation when individuals can possibly change groups in this model.

The problem again, is that any individual seen > 1 time (it's resighted after the first observation occasion) can *never* change group assignments away from their initial value group assignment. So, this model is destined for failure.

There are many ways one could fix this:

- Marginalize over the z 's as you did (perhaps using nimbleEcology)
- Write the model differently, using a binary latent indicator variable to represent the group assignment of each individual. This could also work for > 2 groups, where the group indicator variable follows a categorical prior
- Use a latent state formulation as you have, but write a custom MCMC sampler to update entire rows of the z matrix (the latent state variables for one individual for all time periods) simultaneously, thus enabling transitions between groups
- Probably other ways, also.

7.2 Trap dep

Multievent formulation à la [Pradel and Sanz-Aguilar \[2012\]](#). Also add example w/ individual time-varying covariate.

7.3 Transience

Multievent treatment à la [Genovart and Pradel \[2019\]](#). Remind of the two age-classes on survival technique.

7.4 Temporary emigration

Multistate treatment as in [Schaub et al. \[2004\]](#). See example in [Băncilă et al. \[2018\]](#).

Transition matrix:

$$\times = \begin{pmatrix} z_t = \text{in} & z_t = \text{out} & z_t = D \\ \phi(1 - \psi^{\text{in} \rightarrow \text{out}}) & \phi\psi^{\text{in} \rightarrow \text{out}} & 1 - \phi \\ \phi\psi^{\text{out} \rightarrow \text{in}} & \phi(1 - \psi^{\text{out} \rightarrow \text{in}}) & 1 - \phi \\ 0 & 0 & 1 \end{pmatrix} \begin{array}{l} z_{t-1} = \text{in} \\ z_{t-1} = \text{out} \\ z_{t-1} = D \end{array}$$

Observation matrix:

$$\times = \begin{pmatrix} y_t = 1 & y_t = 2 \\ 1 - p & p \\ 1 & 0 \\ 1 & 0 \end{pmatrix} \begin{array}{l} z_t = \text{in} \\ z_t = \text{out} \\ z_t = D \end{array}$$

7.5 Memory model

How to make your models remember?

So far, the dynamics of the states are first-order Makovian. The site where you will be depends only on the site where you are, and not on the sites you were previously. How to relax this assumption, and go second-order Markovian?

Memory models were initially proposed by [Hestbeck et al. \[1991\]](#) and [Brownie et al. \[1993\]](#), then formulated as HMMs in [Rouan et al. \[2009\]](#). See also [Cole et al. \[2014\]](#).

Remember HMM model for dispersal between 2 sites

Transition matrix

$$\times = \begin{pmatrix} z_t = A & z_t = B & z_t = D \\ \bar{\phi}^A(1 - \psi^{AB}) & \bar{\phi}^B\psi^{AB} & 1 - \bar{\phi}^A \\ \phi^B\psi^{BA} & \phi^B(1 - \psi^{BA}) & 1 - \phi^B \\ 0 & 0 & 1 \end{pmatrix} \begin{array}{l} z_{t-1} = A \\ z_{t-1} = B \\ z_{t-1} = D \end{array}$$

Observation matrix

$$\times = \begin{pmatrix} y_t = 1 & y_t = 2 & y_t = 3 \\ 1 - p^A & p^A & 0 \\ 1 - p^B & 0 & p^B \\ 1 & 0 & 0 \end{pmatrix} \begin{array}{l} z_t = A \\ z_t = B \\ z_t = D \end{array}$$

HMM formulation of the memory model

To keep track of the sites previously visited, the trick is to consider states as being pairs of sites occupied

- States
- AA is for alive in site A at t and alive in site A at $t - 1$
- AB is for alive in site A at t and alive in site B at $t - 1$

- BA is for alive in site B at t and alive in site A at $t - 1$
- BB is for alive in site B at t and alive in site B at $t - 1$
- D is for dead
- Observations
- 1 not captured
- 2 captured at site A
- 3 captured at site B

Vector of initial state probabilities

$$\times = \begin{pmatrix} z_t = AA & z_t = AB & z_t = BA & z_t = BB & z_t = D \\ \pi^{AA} & \pi^{AB} & \pi^{BA} & \pi^{BB} & 0 \end{pmatrix}$$

where $\pi^{BB} = 1 - (\pi^{AA} + \pi^{AB} + \pi^{BA})$, and π^{ij} at site j when first captured at t and site i at $t - 1$.

Transition matrix

$$\times = \begin{pmatrix} z_t = AA & z_t = AB & z_t = BA & z_t = BB & z_t = D \\ \phi^{AA\bar{A}} & \phi^{A\bar{A}B} & 0 & 0 & 1 - \phi^{AA\bar{A}} - \phi^{A\bar{A}B} \\ 0 & 0 & \phi^{ABA} & \phi^{ABB} & 1 - \phi^{ABA} - \phi^{ABB} \\ \phi^{BAA} & \phi^{BAB} & 0 & 0 & 1 - \phi^{BAA} - \phi^{BAB} \\ 0 & 0 & \phi^{BBA} & \phi^{BBB} & 1 - \phi^{BBA} - \phi^{BBB} \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{array}{l} z_{t-1} = AA \\ z_{t-1} = AB \\ z_{t-1} = BA \\ z_{t-1} = BB \\ z_{t-1} = D \end{array}$$

ϕ^{ijk} is probability to be in site k at time $t + 1$ for an individual present in site j at t and in site i at $t - 1$

Transition matrix, alternate parameterization

$$\times = \begin{pmatrix} z_t = AA & z_t = AB & z_t = BA & z_t = BB & z_t = D \\ \phi\psi^{AA\bar{A}} & \phi(1 - \psi^{AA\bar{A}}) & 0 & 0 & 1 - \phi \\ 0 & 0 & \phi(1 - \psi^{ABB}) & \phi\psi^{ABB} & 1 - \phi \\ \phi\psi^{BAA} & \phi(1 - \psi^{BAA}) & 0 & 0 & 1 - \phi \\ 0 & 0 & \phi(1 - \psi^{BBB}) & \phi\psi^{BBB} & 1 - \phi \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{array}{l} z_{t-1} = AA \\ z_{t-1} = AB \\ z_{t-1} = BA \\ z_{t-1} = BB \\ z_{t-1} = D \end{array}$$

ϕ is the probability of surviving from one occasion to the next. ψ_{ijj} is the probability an animal stays at the same site j given that it was at site i on the previous occasion.

Observation matrix

$$\times = \begin{pmatrix} y_t = 1 & y_t = 2 & y_t = 3 \\ 1 - p^A & p^A & 0 \\ 1 - p^B & 0 & p^B \\ 1 - p^A & p^A & 0 \\ 1 - p^B & 0 & p^B \\ 1 & 0 & 0 \end{pmatrix} z_t = \begin{array}{l} AA \\ AB \\ BA \\ BB \\ D \end{array}$$

7.6 Posterior predictive check

Classical m-array (minimal sufficient statistics for CJS model) as in Paganin and de Valpine [2023]. Individual performance in Chambert et al. [2014] and Nater et al. [2020]. Sojourn time is geometric assumption in Conn et al. [2018].

For the CJS model, we would use the so-called m-array which gathers the elements m_{ij} for the number of marked individuals initially released at time i that were first detected again at time j .

Refer to a case study. With m-array and Nimble functions. Refer to paper by Paganin & de Valpine and use code in <https://github.com/salleuska/fastCPPP>. Also papers by Chambert et al. (individual performance) and Conn et al. (geometric time, and hidden semi-Markov models).

Check out https://r-nimble.org/nimbleExamples/posterior_predictive.html.

8

Life history

WORK IN PROGRESS

See <https://www.oxfordbibliographies.com/display/document/obo-9780199830060/obo-9780199830060-0016.xml> and https://en.wikipedia.org/wiki/Life_history_theory for a definition. I think that all case studies below fall in the LH category. I might consider moving the disease ecology case study in the main Sites and states chapter. See however <https://onlinelibrary.wiley.com/doi/epdf/10.1111/ele.13681> for a link between disease ecology and life history theory.

8.1 Access to reproduction

[Pradel et al. \[1997\]](#)

Transition matrix:

$$\times = \begin{pmatrix} z_t = J & z_t = 1yNB & z_t = 2yNB & z_t = B & z_t = D \\ 0 & \phi_1(1 - \alpha_1) & 0 & \phi_1\alpha_1 & 1 - \phi_1 \\ 0 & 0 & \phi_2(1 - \alpha_2) & \phi_2\alpha_2 & 1 - \phi_2 \\ 0 & 0 & 0 & \phi_3 & 1 - \phi_3 \\ 0 & 0 & 0 & \phi_B & 1 - \phi_B \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{array}{l} z_{t-1} = J \\ z_{t-1} = 1yNB \\ z_{t-1} = 2yNB \\ z_{t-1} = B \\ z_{t-1} = D \end{array}$$

First-year and second-year individuals breed with probabilities α_1 and α_2 . Then, everybody breeds from age 3.

Observation matrix:

$$\times = \begin{pmatrix} y_t = 1 & y_t = 2 & y_t = 3 & y_t = 4 \\ 1 & 0 & 0 & 0 \\ 1 - p_1 & p_1 & 0 & 0 \\ 1 - p_2 & 0 & p_2 & 0 \\ 1 - p_3 & 0 & 0 & p_3 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{array}{l} z_t = J \\ z_t = 1yNB \\ z_t = 2yNB \\ z_t = B \\ z_t = D \end{array}$$

Juveniles are never detected.

8.2 Tradeoffs

[Morano et al. \[2013\]](#), [Shefferson et al. \[2003\]](#), and [Cruz-Flores et al.](#)

Case study with simulations as in Oikos paper, see Figure 1 and Table 2. Would be a nice example of the use of simulations. Another example could be the statistical power analyses.

Also consider paper by Sarah on red-footed boobies.

8.3 Breeding dynamics

[Pradel et al. \[2012\]](#), [Desprez et al. \[2011\]](#), [Desprez et al. \[2013\]](#), and [Pacourreau et al. \[2019\]](#)

8.4 Using data on dead recoveries

8.4.1 Ring recovery simple model

8.4.2 Combination of live captures and dead recoveries

Combine live recapture w/ dead recoveries by [Lebreton et al. \[1999\]](#).

Transition matrix

$$\times = \begin{pmatrix} z_t = A & z_t = JD & z_t = D \\ s & 1-s & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{array}{l} z_{t-1} = \text{alive} \\ z_{t-1} = \text{just dead} \\ z_{t-1} = \text{dead for good} \end{array}$$

Observation matrix

$$\times = \begin{pmatrix} y_t = 1 & y_t = 2 & y_t = 3 \\ 1-p & 0 & p \\ 1-r & r & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{array}{l} z_t = A \\ z_t = JD \\ z_t = D \end{array}$$

8.4.3 Cause-specific mortalities

[Koons et al. \[2014\]](#), [Fernández-Chacón et al. \[2016\]](#) and [Ruette et al. \[2015\]](#)

8.5 Stopover duration

[Guérin et al. \[2017\]](#) for a comparison of method, would be great to reproduce all analyses.

8.6 Actuarial senescence

[Choquet et al. \[2011\]](#), [Péron et al. \[2016\]](#) and [Marzolin et al. \[2011\]](#).

8.7 Uncertainty in age

E.g. [Gervasi et al. \[2017\]](#).

8.8 Uncertainty in age and size

E.g. [Gowan et al. \[2021\]](#).

Conclusion

I hope to have convinced you that hidden Markov models combined with the Bayesian framework are very flexible to analyse capture-recapture data. With your data, you may ask a myriad of questions. The limit is your imagination (and CPU time).

Let me leave you with a few pieces of advice. This is not rocket science, just a few things based on my own experience of building HMM to analyse data with Bayesian statistics.

- **Make your ecological question explicit.** First things first. Make sure you've spent some time to make your ecological question explicit. This step will help you to stay on course, and make the right choices. For example, it's fine to use subsets of your data to address different questions.
- Now in terms of modeling. **Think of observations and states first.** Don't jump on your keyboard right away. Spend some time thinking about your model with pen and paper. In particular make sure you have the observations and the states of your HMM. **Then write down the observation and transition matrices on paper.** Write down the transition matrix. You may act as if you had no imperfect detection. This is really what you're after, the ecological process (survival, dispersal, etc). Proceed with the observation matrix.
- When it comes to model fitting with NIMBLE, **start simple**, with all parameters constant for example. Make sure convergence is reached. Then **add complexity one step at a time**. Time effect for example, or random effects, or uncertainty in the assignment of states.
- **Consider doing simulations** to better understand your model. When it comes to model building, consider simulating data to better understand your model. You will always learn something on your model by

seeing it an engine to generate data, instead of estimating its parameters. The nice thing with NIMBLE is that you can use your model to simulate data.

- Another advice, quite general in programming, is to not try to optimize your code or to try to make it elegant right away. **Make your model work first, then think of optimization.**

Bibliography

- J. Albert and J. Hu. *Probability and Bayesian Modeling*. Chapman and Hall/CRC, 1st edition edition, 2019.
- N. A. Arnason. Parameter estimates from mark-recapture experiments on two populations subject to migration and death. *Researches on Population Ecology*, 13(2):97–113, 1972.
- N. A. Arnason. The estimation of population size, migration rates and survival in a stratified population. *Researches on Population Ecology*, 15(2):1–8, 1973.
- F. Bairlein. Population studies of white storks *ciconia ciconia* in europe, with reference to the western population. In C. Perrins, J.-D. Lebreton, and G. Hirons, editors, *Bird Population Studies: Relevance to Conservation and Management*, pages 207 – 229. Oxford: Oxford University Press., 1991.
- C. Barbraud, J. C. Barbraud, and M. Barbraud. Population dynamics of the White Stork *ciconia ciconia* in western France. *Ibis*, 141:469–479, 1999.
- S. J. Bonner and C. J. Schwarz. An extension of the Cormack–Jolly–Seber model for continuous covariates with application to *microtus pennsylvanicus*. *Biometrics*, 62(1):142–149, 2006.
- C. Brownie, J. E. Hines, J. D. Nichols, K. H. Pollock, and J. B. Hestbeck. Capture-recapture studies for multiple strata including non-Markovian transitions. *Biometrics*, 49:1173–1187, 1993.
- S. T. Buckland. A Conversation with Richard M. Cormack. *Statistical Science*, 31(2):142 – 150, 2016.

- R. I. Băncilă, R. Pradel, R. Choquet, R. Plăiasu, and O. Gimenez. Using temporary emigration to inform movement behaviour of cave-dwelling invertebrates: a case study of a cave harvestman species. *Ecological Entomology*, 43(5):551–559, 2018.
- T. Chambert, J. J. Rotella, and M. D. Higgs. Use of posterior predictive checks as an inferential tool for investigating individual heterogeneity in animal population vital rates. *Ecology and Evolution*, 4(8):1389–1397, 2014.
- Rémi Choquet, Anne Viallefont, Lauriane Rouan, Kamel Gaanoun, and Jean-Michel Gaillard. A semi-Markov model to assess reliably survival patterns from birth to death in free-ranging populations. *Methods in Ecology and Evolution*, 2(4):383–389, 2011.
- D. Cole. *Parameter Redundancy and Identifiability*. Chapman and Hall/CRC, 2020.
- D.J. Cole, B.J.T. Morgan, R.S. McCrea, R. Pradel, O. Gimenez, and R. Choquet. Does your species have memory? analyzing capture–recapture data with memory models. *Ecology and Evolution*, 4(11):2124–2133, 2014.
- P. B. Conn, D. S. Johnson, P. J. Williams, S. R. Melin, and M. B. Hooten. A guide to bayesian model checking for ecologists. *Ecological Monographs*, 88(4):526–542, 2018.
- P.B. Conn and E.G. Cooch. Multistate capture–recapture analysis under imperfect state observation: An application to disease models. *Journal of Applied Ecology*, 46(2):486–492, 2009.
- E.G. Cooch and G. White. *Program MARK: a gentle introduction*. 13th edition edition, 2017. URL <http://www.phidot.org/software/mark/docs/book/>.
- E.G. Cooch, P.B. Conn, S.P. Ellner, A.P. Dobson, and K.H. Pollock. Disease dynamics in wild populations: modeling and estimation: a review. *Journal of Ornithology*, 152(2):485–509, 2012.

- L.M.M Cook, L. P. Brower, and H. J. Croze. The accuracy of a population estimation from multiple recapture data. *Journal of Animal Ecology*, 36(1):57–60, 1967.
- M. Cruz-Flores, R. Pradel, J. Bried, J. González-Solís, and R. Ramos. Sex-specific costs of reproduction on survival in a long-lived seabird. *Biology Letters*, 17(3):2021.
- S. Cubaynes, R. Pradel, R. Choquet, C. Duchamp, J.-M. Gaillard, J.-D. Lebreton, E. Marboutin, C. Miquel, A.-M. Reboulet, C. Poillot, P. Taberlet, and O. Gimenez. Importance of accounting for detection heterogeneity when estimating abundance: the case of French wolves. *Conservation Biology*, 24(2):621–626, 2010.
- S. Cubaynes, C. Lavergne, E. Marboutin, and O. Gimenez. Assessing individual heterogeneity using model selection criteria: how many mixture components in capture–recapture models? *Methods in Ecology and Evolution*, 3(3):564–573, 2012.
- P. de Valpine, D. Turek, C. J. Paciorek, C. Anderson-Bergman, D. T. Lang, and R. Bodik. Programming with models: writing statistical algorithms for general model structures with NIMBLE. *Journal of Computational and Graphical Statistics*, 26(2):403–413, 2017.
- M. Desprez, R. Pradel, E. Cam, J.-Y. Monnat, and O. Gimenez. Now you see him, now you don't: experience, not age, is related to reproduction in kittiwakes. *Proceedings of the Royal Society B: Biological Sciences*, 278(1721):3060–3066, 2011.
- M. Desprez, C. R. McMahon, M. A. Hindell, R. Harcourt, and O. Gimenez. Known unknowns in an imperfect world: incorporating uncertainty in recruitment estimates using multi-event capture-recapture models. *Ecology and Evolution*, 3(14):4658–4668, 2013.
- J. A. Dupuis. Bayesian estimation of movement and survival probabilities from capture-recapture data. *Biometrika*, 82(4):761–772, 1995.
- C. R. Faustino, C. S. Jennelle, V. Connolly, A. K. Davis, E. C. Swarthout, A. A. Dhondt, and E. G. Cooch. Infection dynamics in

- a house finch population: seasonal variation in survival, encounter, and transmission rate. *Journal of Animal Ecology*, 73:651–669, 2004.
- X. Fernández-i Marín. ggmcmc: Analysis of mcmc samples and bayesian inference. *Journal of Statistical Software*, 70(9):1–20, 2016.
- A. Fernández-Chacón, E. Moland, S. H. Espeland, A. R. Kleiven, and E. M. Olsen. Causes of mortality in depleted populations of Atlantic cod estimated from multi-event modelling of mark--recapture and recovery data. *Canadian Journal of Fisheries and Aquatic Sciences*, 74(1):116–126, 2016.
- M. Frederiksen, J.-D. Lebreton, R. Pradel, R. Choquet, and O. Gimenez. Identifying links between vital rates and environment: A toolbox for the applied ecologist. *Journal of Applied Ecology*, 51(1):71–81, 2014.
- J. Gabry and T. Mahr. bayesplot: Plotting for bayesian models, 2022. URL <https://mc-stan.org/bayesplot/>. R package version 1.10.0.
- A. Gelman and J. Hill. *Data Analysis Using Regression and Multi-level/Hierarchical Models*. Cambridge University Press, 2006.
- A. Gelman, J. Hwang, and A. Vehtari. Understanding predictive information criteria for bayesian models. *Statistics and Computing*, 24(6): 997–1016, 2014.
- A. Gelman, A. Vehtari, D. Simpson, C. C. Margossian, B. Carpenter, Y. Yao, L. Kennedy, J. Gabry, P.-C. Bürkner, and M. Modrák. Bayesian workflow, 2020.
- M. Genovart and R. Pradel. Transience effect in capture-recapture studies: The importance of its biological meaning. *Plos One*, 14(9): e0222241, 2019.
- M. Genovart, R. Pradel, and D. Oro. Exploiting uncertain ecological fieldwork data with multi-event capture–recapture modelling: an example with bird sex assignment. *Journal of Animal Ecology*, 81(5): 970–977, 2012.

- V. Gervasi, L. Boitani, D. Paetkau, M. Posillico, E. Randi, and P. Ciucci. Estimating survival in the Apennine brown bear accounting for uncertainty in age classification. *Population Ecology*, 59(2):119–130, 2017.
- O. Gimenez and R. Choquet. Individual heterogeneity in studies on marked animals using numerical integration: capture–recapture mixed models. *Ecology*, 91(4):951–957, 2010.
- O. Gimenez, C. Crainiceanu, C. Barbraud, S. Jenouvrier, and B. J. T. Morgan. Semiparametric Regression in Capture–Recapture Modeling. *Biometrics*, 62(3):691–698, 2006.
- O. Gimenez, V. Rossi, R. Choquet, C. Dehais, B. Doris, H. Varella, J.-P. Vila, and R. Pradel. State-space modelling of data on marked individuals. *Ecological Modelling*, 206(3):431–438, 2007.
- O. Gimenez, S. J. Bonner, R. King, R.A. Parker, S.P. Brooks, L.E. Jamieson, V. Grosbois, B.J.T. Morgan, and L. Thomas. WinBUGS for population ecologists: Bayesian modeling using Markov chain Monte Carlo methods. In D. L. Thomson, E. G. Cooch, and M. J. Conroy, editors, *Modeling Demographic Processes In Marked Populations*, pages 883–915. Springer, 2009a.
- O. Gimenez, A. Grégoire, and T. Lenormand. Estimating and visualizing fitness surfaces using mark–recapture data. *Evolution*, 63(12): 3097–3105, 2009b.
- O. Gimenez, B. J. T. Morgan, and S. P. Brooks. Weak identifiability in models for mark-recapture-recovery data. In D. L. Thomson, E. G. Cooch, and M. J. Conroy, editors, *Modeling demographic processes in marked populations*, pages 8–48. Springer, 2009c.
- O. Gimenez, J.-D. Lebreton, J.-M. Gaillard, R. Choquet, and R. Pradel. Estimating demographic parameters using hidden process dynamic models. *Theoretical Population Biology*, 82(4):307–316, 2012.
- O. Gimenez, E. Cam, and J.-M. Gaillard. Individual heterogeneity and capture–recapture models: what, why and how? *Oikos*, 127(5):664–686, 2018a.

- O. Gimenez, Lebreton, J.-D., Choquet, R., Pradel, and R. R2ucare: An R package to perform goodness-of-fit tests for capture–recapture models. *Methods in Ecology and Evolution*, 9(7):1749–1754, 2018b.
- B.R. Goldstein, D. Turek, L. Ponisio, and P. de Valpine. nimbleEcology: Distributions for ecological models in ‘nimble’, 2021. URL <https://CRAN.R-project.org/package=nimbleEcology>. R package version 0.4.1.
- T. A. Gowan, M. D. Tringali, J. A. Hostetler, J. Martin, L. I. Ward-Geiger, and J. M. Johnson. A hidden markov model for estimating age-specific survival when age and size are uncertain. *Ecology*, 102(8):e03426, 2021.
- V. Grosbois, O. Gimenez, J. M. Gaillard, R. Pradel, C. Barbraud, J. Clobert, A. P. Møller, and H. Weimerskirch. Assessing the impact of climate variation on survival in vertebrate populations. *Biological Reviews*, 83(3):357–399, 2008.
- S. Guérin, D. Picard, R. Choquet, and A. Besnard. Advances in methods for estimating stopover duration for migratory species using capture-recapture data. *Ecological Applications*, 27(5):1594–1604, 2017.
- J. B. Hestbeck, J. D. Nichols, and R. A. Malecki. Estimates of movement and site fidelity using mark-resight data of wintering canada geese. *Ecology*, 72(2):523–533, 1991.
- D. Jurafsky and J. H. Martin. Appendix on Hidden Markov models. In D. Jurafsky and J. H. Martin, editors, *Speech and Language Processing*. Prentice Hall, 3rd edition edition, 2023. URL <https://web.stanford.edu/~jurafsky/slp3/>.
- S. Kanyamibwa, A. Schierer, R. Pradel, and J.-D. Lebreton. Changes in adult survival rates in a western European population of the White Stork *ciconia ciconia*. *Ibis*, 132:27–35, 1990.
- K. Khan and C.A. Calder. Restricted spatial regression methods: Implications for inference. *Journal of the American Statistical Association*, 117(537):482–494, 2022.

- R. King, B. J. T. Morgan, O. Gimenez, and S. P. Brooks. *Bayesian Analysis for Population Ecology*. Chapman and Hall/CRC, 2009.
- D. N. Koons, M. Gamelon, J.-M. Gaillard, L. M. Aubry, R. F. Rockwell, F. Klein, R. Choquet, and O. Gimenez. Methods for studying cause-specific senescence in the wild. *Methods in Ecology and Evolution*, 5(9):924–933, 2014.
- R. Langrock and R. King. Maximum likelihood estimation of mark-recapture–recovery models in the presence of continuous covariates. *The Annals of Applied Statistics*, 7(3):1709–1732, 2013.
- J.-D. Lebreton, K. P. Burnham, J. Clobert, and D. R. Anderson. Modeling survival and testing biological hypotheses using marked animals: A unified approach with case studies. *Ecological Monographs*, 62:67–118, 1992.
- J.-D. Lebreton, T. Almeras, and R. Pradel. Competing events, mixtures of information and multistratum recapture models. *Bird Study*, 46:39–46, 1999.
- J.-D. Lebreton, J. D. Nichols, R. J. Barker, R. Pradel, and J. A. Spendelow. Modeling individual animal histories with multistate capture–recapture models. *Advances in Ecological Research*, 41:87–173, 2009.
- W. A. Link and M. J. Eaton. On thinning of chains in mcmc. *Methods in Ecology and Evolution*, 3(1):112–115, 2012.
- L. Marescot, S. Benhaiem, O. Gimenez, H. Hofer, J.-D. Lebreton, X. A. Olarte-Castillo, S. Kramer-Schadt, and M. L. East. Social status mediates the fitness costs of infection with canine distemper virus in Serengeti spotted hyenas. *Functional Ecology*, 32(5):1237–1250, 2018.
- G. Marzolin, A. Charmantier, and O. Gimenez. Frailty in state-space models: application to actuarial senescence in the dipper. *Ecology*, 92(3):562–567, 2011.
- M. A. McCarthy. *Bayesian Methods for Ecology*. Cambridge University Press, 2007.

- M. A. McCarthy and P. Masters. Profiting from prior information in bayesian analyses of ecological data. *Journal of Applied Ecology*, 42(6):1012–1019, 2005.
- B. T. McClintock, R. Langrock, O. Gimenez, E. Cam, D. L. Borchers, R. Glennie, and T. A. Patterson. Uncovering ecological state dynamics with hidden Markov models. *Ecology Letters*, 23(12):1878–1903, 2020.
- R. McElreath. *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. Chapman and Hall/CRC, 2nd edition edition, 2020.
- S. B. MacGrayne. *The Theory That Would Not Die: How Bayes' Rule Cracked the Enigma Code, Hunted Down Russian Submarines, and Emerged Triumphant from Two Centuries of Controversy*. Yale University Press, 2011.
- S. Morano, K. M. Stewart, J. S. Sedinger, C. A. Nicolai, and M. Vavra. Life-history strategies of North American elk: trade-offs associated with reproduction and survival. *Journal of Mammalogy*, 94(1):162–172, 2013.
- C. R. Nater, Y. Vindenes, P. Aass, D. Cole, O. Langangen, S. J. Moe, A. Rustadbakken, D. Turek, L. A. Vøllestad, and T. Ergon. Size- and stage-dependence in cause-specific mortality of migratory brown trout. *Journal of Animal Ecology*, 89(9):2122–2133, 2020.
- J. D. Nichols, J. E. Hines, K. H. Pollock, R. L. Hinz, and W. A. Link. Estimating breeding proportions and testing hypotheses about costs of reproduction with capture-recapture data. *Ecology*, 75(7):2052–2065, 1994.
- J.D. Nichols, J.R. Sauer, K. H. Pollock, and J. B. Hestbeck. Estimating transition probabilities for stage-based population projection matrices using capture–recapture data. *Ecology*, 73:306–312, 1992.
- V. Ollivier, R. Choquet, A. Gamble, M. Bastien, B. Combes, E. Gilot-Fromont, M. Pellerin, J.-M. Gaillard, J.-F. Lemaître, H. Verheyden, and T. Boulinier. Temporal dynamics of antibody level against lyme disease bacteria in roe deer: Tale of a sentinel? *Ecology and Evolution*, 13(8):e10414, 2023.

- N. Pacourea, M. Authier, K. Delord, and C. Barbraud. Population response of an apex Antarctic consumer to its prey and climate fluctuations. *Oecologia*, 189(2):279–291, 2019.
- S. Paganin and P. de Valpine. Computational methods for fast bayesian model assessment via calibrated posterior p-values, 2023.
- G. Péron, Y. Ferrand, F. Gossman, C. Bastat, M. Guenezan, and O. Gimenez. Nonparametric spatial regression of survival probability: visualization of population sinks in Eurasian woodcock. *Ecology*, 92(8):1672–1679, 2011.
- L. C. Ponisio, P. de Valpine, N. Michaud, and D. Turek. One size does not fit all: Customizing mcmc methods for hierarchical models using nimble. *Ecology and Evolution*, 10(5):2385–2416, 2020.
- R. Pradel. Multievent: An Extension of Multistate Capture–Recapture Models to Uncertain States. *Biometrics*, 61(2):442–447, 2005.
- R. Pradel. The stakes of capture–recapture models with state uncertainty. In D. L. Thomson, E. G. Cooch, and M. J. Conroy, editors, *Modeling demographic processes in marked populations*, pages 781–795. Springer, 2009.
- R. Pradel and A. Sanz-Aguilar. Modeling trap-awareness and related phenomena in capture-recapture studies. *Plos One*, 7(3):e32666, 2012.
- R. Pradel, A.R. Johnson, A. Viallefont, R.G. Nager, and F. Césilly. Local recruitment in the Greater flamingo: A new approach using capture–mark–recapture data. *Ecology*, 78:1431–1445, 1997.
- R. Pradel, C.M.A. Wintrebert, and O. Gimenez. A proposal for a goodness-of-fit test to the arnason–schwarz multisite capture–recapture model. *Biometrics*, 59:43–53, 2003.
- R. Pradel, , O. Gimenez, and J.-D. Lebreton. Principles and interest of gof tests for multistate capture–recapture models. *Animal Biodiversity and Conservation*, 28.2:189–204, 2005.

- R. Pradel, L. Maurin-Bernier, O. Gimenez, M. Genovart, R. Choquet, and D. Oro. Estimation of sex-specific survival with uncertainty in sex assessment. *Canadian Journal of Statistics*, 36(1):29–42, 2008.
- R. Pradel, R. Choquet, and A. Béchet. Breeding Experience Might Be a Major Determinant of Breeding Probability in Long-Lived Species: The Case of the Greater Flamingo. *Plos One*, 7(12):e51016, 2012.
- G. Péron, J.-M. Gaillard, C. Barbraud, C. Bonenfant, A. Charmantier, R. Choquet, T. Coulson, V. Grosbois, A. Loison, G. Marzolin, N. Owen-Smith, D. Pardo, F. Plard, R. Pradel, C. Toïgo, and O. Gimenez. Evidence of reduced individual heterogeneity in adult survival of long-lived species. *Evolution*, 70(12):2909–2914, 2016.
- L.R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.
- C.P. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer, 2nd edition edition, 2004.
- C.P. Robert and G. Casella. *Introducing Monte Carlo Methods with R*. Springer, 2010.
- J. P. Rose, G. D. Wylie, M. L. Casazza, and B. J. Halstead. Integrating growth and capture–mark–recapture models reveals size-dependent survival in an elusive species. *Ecosphere*, 9(8):e02384, 2018.
- L. Rouan, R. Choquet, and R. Pradel. A general framework for modeling memory in capture–recapture data. *Journal of Agricultural, Biological, and Environmental Statistics*, 14(3):338–355, 2009.
- J. A. Royle. Modeling individual effects in the Cormack–Jolly–Seber model: A state–space formulation. *Biometrics*, 64(2):364–370, 2008.
- J.A. Royle, R. B. Chandler, R. Sollmann, and B. Gardner. *Spatial Capture-Recapture*. Academic Press, 2013.

- S. Ruette, J.-M. Vandel, M. Albaret, and S. Devillard. Comparative survival pattern of the syntopic pine and stone martens in a trapped rural area in France. *Journal of Zoology*, 295(3):214–222, 2015.
- C. S. Rushing. An ecologist’s introduction to continuous-time multi-state models for capture–recapture data. *Journal of Animal Ecology*, 92(4):936–944, 2023.
- S. Santoro, I. Pacios, S. Moreno, A. Bertó-Moran, and C. Rouco. Multi–event capture–recapture modeling of host–pathogen dynamics among European rabbit populations exposed to myxoma and Rabbit Hemorrhagic Disease Viruses: Common and heterogeneous patterns. *Veterinary Research*, 45(1):39, 2014.
- J. F. Saracco, J. A. Royle, D. F. DeSante, and B. Gardner. Modeling spatial variation in avian survival and residency probabilities. *Ecology*, 91(7):1885–1891, 2010.
- M. Schaub, O. Gimenez, B.R. Schmidt, and R. Pradel. Estimating survival and temporary emigration in the multistate capture-recapture framework. *Ecology*, 85:2107–2113, 2004.
- M. Schaub, W. Kania, and Koppen U. Variation of primary production during winter induces synchrony in survival rates in migratory white storks *ciconia ciconia*. *Journal of Animal Ecology*, 74:656–666, 2005.
- C. J. Schwarz, J. F. Schweigert, and A. N. Arnason. Estimating migration rates using tag-recovery data. *Biometrics*, 49:177–193, 1993.
- R. P. Scofield, D. J. Fletcher, and C. J. R. Robertson. Titi (sooty shearwaters) on whero island: Analysis of historic data using modern techniques. *Journal of Agricultural, Biological, and Environmental Statistics*, 6(2):268–280, 2001.
- R. P. Shefferson, J. Proper, S. R. Beissinger, and E. L. Simms. Life History Trade-Offs in a Rare Orchid: The Costs of Flowering, Dormancy, and Sprouting. *Ecology*, 84(5):1199–1206, 2003.

- D. Turek. basicmcmcplots: Trace plots, density plots and chain comparisons for mcmc samples, 2022. URL <https://mc-stan.org/bayesplot/>. R package version 0.2.7.
- D. Turek, P. de Valpine, and C. J. Paciorek. Efficient Markov chain Monte Carlo sampling for hierarchical hidden Markov models. *Environmental and Ecological Statistics*, 23(4):549–564, 2016.
- D. Turek, C. Wehrhahn, and O. Gimenez. Bayesian non-parametric detection heterogeneity in ecological models. *Environmental and Ecological Statistics*, 2021.
- B. K. Williams, J. D. Nichols, and M. J. Conroy. *Analysis and Management of Animal Populations*. Academic Press, 2002.
- H. Worthington, R. King, and S. T. Buckland. Analysing mark-recapture–recovery data in the presence of missing covariate data via multiple imputation. *Journal of Agricultural, Biological, and Environmental Statistics*, 20(1):28–46, 2015.
- C. Youngflesh. MCMCvis: Tools to visualize, manipulate, and summarize mcmc output. *Journal of Open Source Software*, 3(24):640, 2018.
- W. Zucchini, I.L. MacDonald, and R. Langrock. *Hidden Markov models for time series: An introduction using R*. Chapman and Hall/CRC, 2nd edition edition, 2016.