Olivier Gimenez

Bayesian Analysis of Capture-Recapture Data with Hidden Markov Models

Theory and Case Studies in R

Contents

List of Tables

List of Figures

Welcome

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

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

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

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

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

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

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

Olivier Gimenez, Montpellier, France
Last updated: January 20, 2022

License

[3] http://creativecommons.org/licenses/by-nc-nd/4.0/
[4] https://creativecommons.org/publicdomain/zero/1.0/

Preface

Why this book?

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

Who should read this book?

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

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

What will you learn?

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

What won't you learn?

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

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

Prerequisites

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

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

Acknowledgements

To be completed.

How this book was written

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

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

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

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

About the author

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

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

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

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

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

Part I

I. Fundations

Introduction

1

Bayesian statistics & MCMC

1.1 Introduction

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

1.2 Bayes' theorem

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

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

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

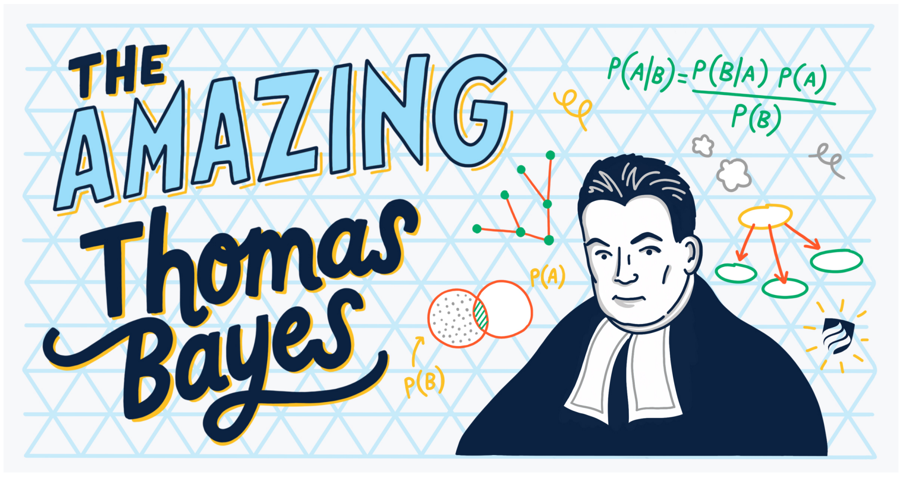


FIGURE 1.1: Cartoon of Thomas Bayes with Bayes' theorem in background. Source: [James Kulich](https://www.elmhurst.edu/blog/thomas-bayes/)

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

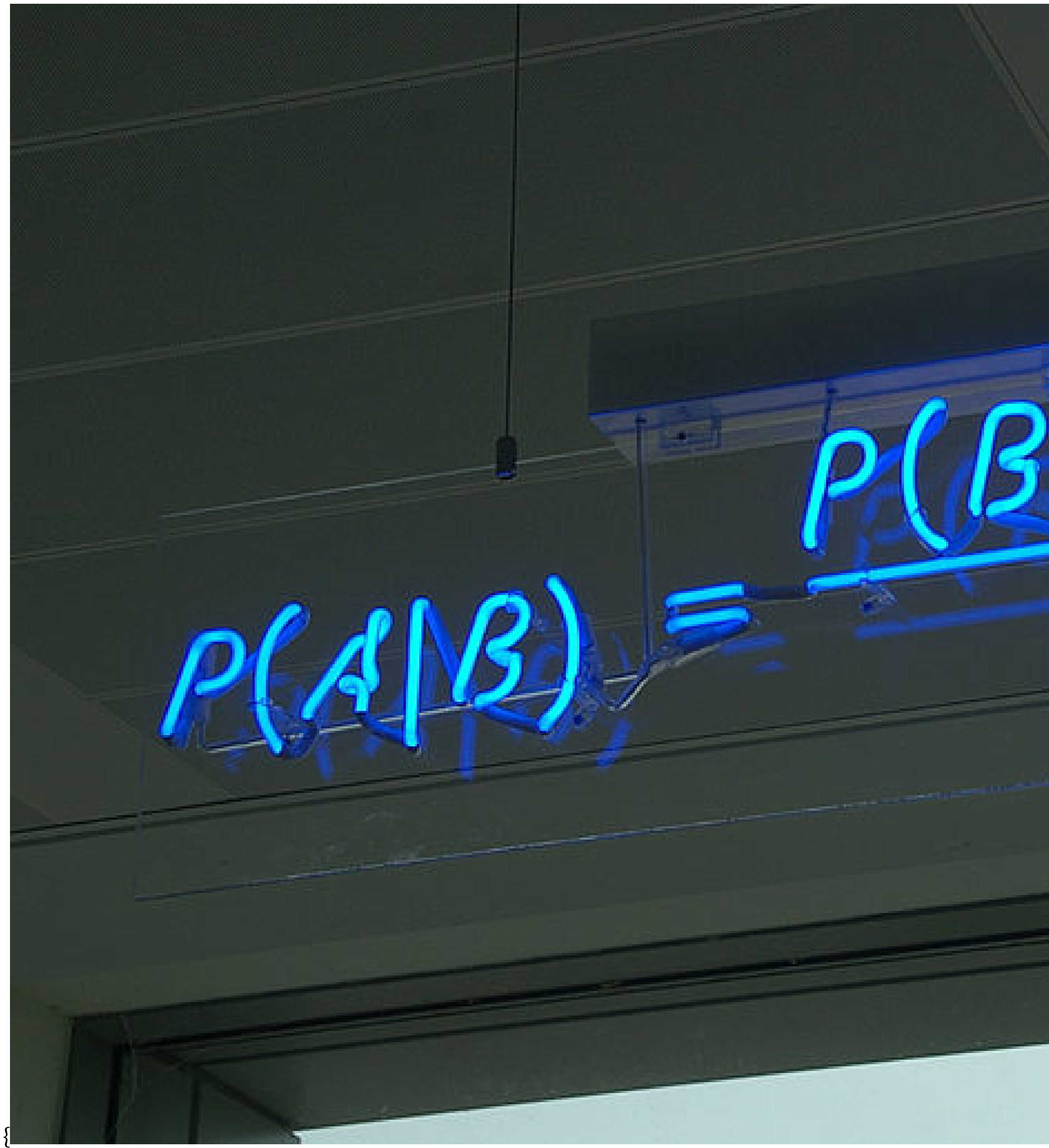
Bayes' theorem (Figure **??**) gives you $\Pr(A \mid B)$ using marginal probabilities $\Pr(A)$ and $\Pr(B)$ and $\Pr(B \mid A)$:

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

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

\begin{figure}

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

{
P(A|B) = P(B

}

\caption{Bayes' theorem spelt out in blue neon. Source: Wikipedia[2]}
\end{figure}

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

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

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

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

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

[2] https://en.wikipedia.org/wiki/Bayes%27_theorem

[3] When teaching Bayes' theorem, I am very much inspired by Tristan Mahr's slides from his introduction to Bayesian regression https://www.tjmahr.com/bayes-intro-lecture-slides-2017/

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[4].

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

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

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

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

On the right-hand side, there is the likelihood. This quantity is the same as in the MLE approach. Yes, the Bayesian and frequentist approaches have the same likelihood at their core, which mostly

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

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[5].

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

1.4 Approximating posteriors via numerical integration

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

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

We build our model first. Assuming all animals are independent of each other and have the same survival probability, then y the number

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

of alive animals at the end of the winter is a binomial distribution[7] with n trials and θ the probability of success:

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

This likelihood can be visualised in R:

```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)
```

banana-book_files/figure-latex/binlik-1.pdf

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

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

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

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

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

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

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

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

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

Then we get a numerical approximation of the posterior in Figure **??** 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)
```

banana-book_files/figure-latex/numapprox-1.pdf

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

How good is our numerical approximation of survival posterior distribution? Ideally, we would want to compare the approximation to the true posterior distribution. Although a closed-form expression for the posterior distribution is in general intractable, when you combine a binomial likelihood together with a beta distribution as a prior, then the posterior distribution is also a beta distribution, which makes it amenable to all sorts of exact calculations[8]. The beta distribution is continuous between 0 and 1, and extends the uniform distribution to situations where not all outcomes are equally likely. It has two parameters a and b that control its shape (Figure **??**).

banana-book_files/figure-latex/betadistribution-1.pdf

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

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

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

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

Clearly, the two distributions are indistinguishable, suggesting that

banana-book_files/figure-latex/compar-1.pdf

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:

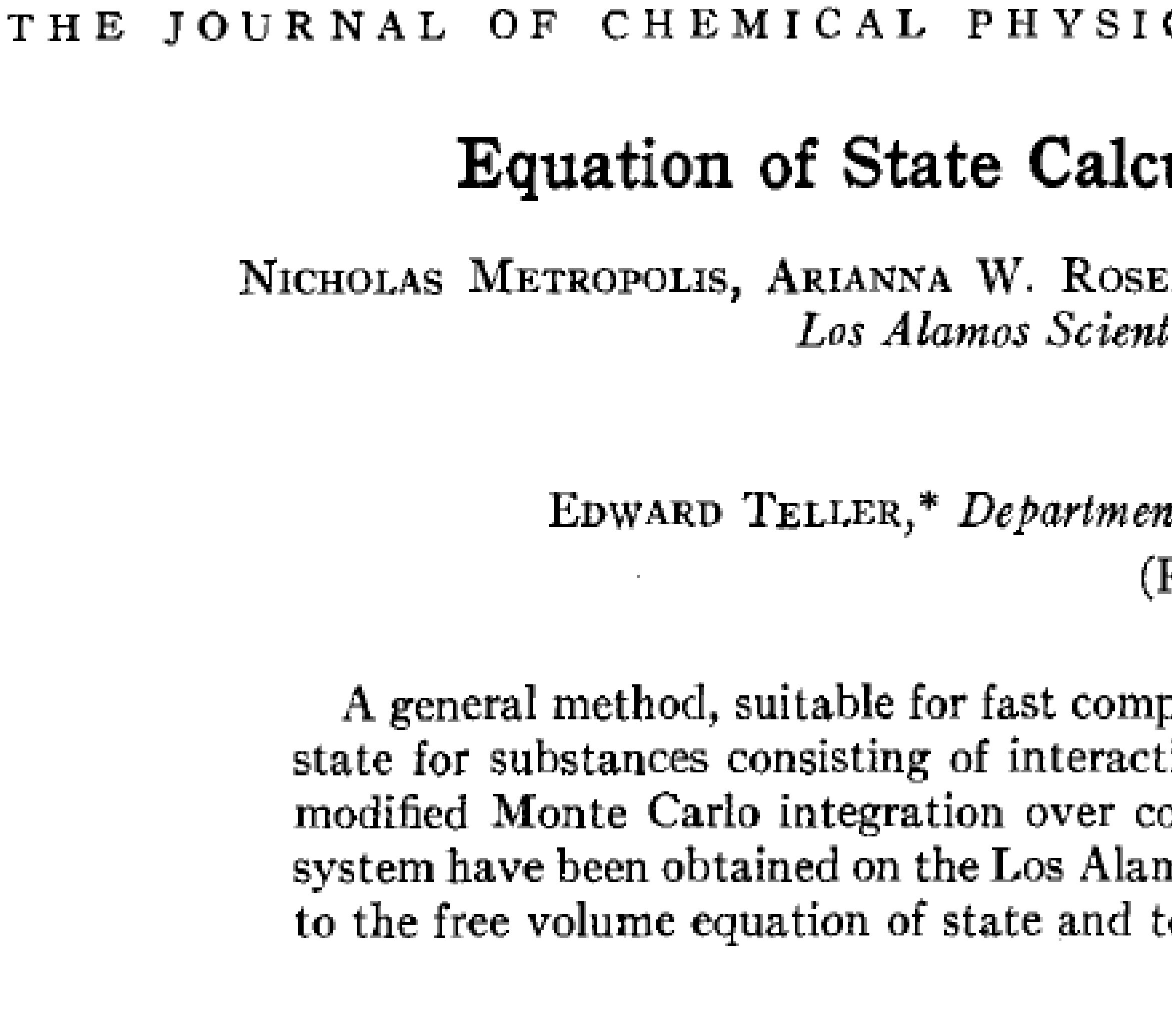
$$\Pr(\alpha, \beta, p \mid \text{data}) = \frac{\Pr(\text{data} \mid \alpha, \beta, p) \times \Pr(\alpha, \beta, p)}{\iiint \Pr(\text{data} \mid \alpha, \beta, p) \Pr(\alpha, \beta, p) d\alpha d\beta dp}$$

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

1.5 Markov chain Monte Carlo (MCMC)

In the early 1990s, statisticians rediscovered work from the 1950's in physics. In a famous paper that would lay the fundations of modern

Bayesian statistics (Figure **??**), 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.


THE JOURNAL OF CHEMICAL PHYSI

Equation of State Calc

NICHOLAS METROPOLIS, ARIANNA W. ROSE
Los Alamos Scient

EDWARD TELLER,* *Departmen*
(I

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

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

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 $\displaystyle\frac{1}{k}\sum_{i=1}^{k} f(X_i)$ 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:

```r
sample_from_posterior <- rbeta(1000, 20, 39) # draw 1000 values from posterior survival beta(2
mean(sample_from_posterior) # compute mean with Monte Carlo integration
## [1] 0.3396
```

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

```r
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

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

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%
## 0.2264 0.4572
```

1.5.2 Markov chains

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

$$\mathbf{\Gamma} = \begin{array}{cc} \text{sunny tomorrow} & \text{rainy tomorrow} \\ \left(\begin{array}{c} 0.8 \\ 0.9 \end{array} \right. & \left. \begin{array}{c} 0.2 \\ 0.1 \end{array} \right) \end{array} \begin{array}{l} \\ \text{sunny today} \\ \text{rainy today} \end{array}$$

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

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

[11]The Markov chain is irreducible and aperiodic.

```
## [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[12]. Here I illustrate the Metropolis algorithm and how to implement it in practice[13].

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

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

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

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

```r
# 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.
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 $R = \dfrac{\Pr(\text{candidate}|\text{data})}{\Pr(\text{current}|\text{data})}$. This is where the magic of MCMC happens, in that $\Pr(\text{data})$, the denominator in the Bayes' theorem, appears in both the numerator and the denominator in R therefore cancels out and does not need to be calculated.
4. If the posterior at the candidate location $\Pr(\text{candidate}|\text{data})$

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

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

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

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

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

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

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

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

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

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

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

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

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

We get the following values.

```
head(theta.post) # first values
## [1] 0.5000 0.2302 0.2906 0.2906 0.2980 0.2980
tail(theta.post) # last values
## [1] 0.2622 0.2622 0.2622 0.3727 0.3232 0.3862
```

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

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

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

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

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

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

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

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

Once the stationary distribution is reached, you may regard the

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

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, we see from the trace plot in Figure **??** 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 banana-book_files/figure-latex/burnin-1.pdf

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

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. From Figure **??**, 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.

banana-book_files/figure-latex/bgr-1.pdf

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

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

1.6.2 Chain length

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

banana-book_files/figure-latex/tracechainlength-1.pdf

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

In addition to trace plots, autocorrelation function (ACF) plots are a convenient way of displaying the strength of autocorrelation in a given sample values. ACF plots provide the autocorrelation between

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

successively sampled values separated by an increasing number of iterations, or *lag* (Figure **??**).

banana-book_files/figure-latex/acfchainlength-1.pdf

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

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 ≥ 1000 independent steps to get reasonable Monte Carlo estimates of model parameters. In the animal survival example, `n.eff` can be calculated with the R `coda::effectiveSize()` function.

Proposal SD	n.eff
0.1	224
1.0	1934
10.0	230

As expected, `n.eff` is less than the number of MCMC iterations because of autocorrelation. Only when the standard deviation of the proposal distribution is 1 and mixing is good (Figures **??** 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[15]. A bug in the code? A typo somewhere? A mistake in your maths? As often when coding is involved, the issue can be identified by removing complexities, and start with a simpler model until you find what the problem is.

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

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

[15]The quote 'When you have computational problems, often there's a problem with your model' is the folk theorem of statistical computing stated by Andrew Gelman in 2008, see `https://statmodeling.stat.columbia.edu/2008/05/13/the_folk_theore/`

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

1.7 Summary

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

1.8 Suggested reading

- Gelman, A. and Hill, J. (2006). Data Analysis Using Regression and Multilevel/Hierarchical Models (Analytical Methods for Social Research)[17]. Cambridge: Cambridge University Press.
- Gelman, A. and colleagues (2020). Bayesian workflow[18]. arXiv preprint.
- McCarthy, M. (2007). Bayesian Methods for Ecology[19]. Cambridge: Cambridge University Press.

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

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

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

- McElreath, R. (2020). Statistical Rethinking: A Bayesian Course with Examples in R and Stan (2nd ed.)[20]. CRC Press.

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

2

NIMBLE tutorial

2.1 Introduction

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

2.2 What is NIMBLE?

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

So why use NIMBLE you may ask? The short answer is that NIMBLE is capable of so much more! First, you will work from within R, but in the background NIMBLE will translate your code in C++ for faster computation (in general). Second, NIMBLE extends the BUGS language for writing new functions and statistical distributions of your own, or grab 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, including sequential Monte Carlo (particle filtering) and Monte Carlo Expectation Maximization (maximum likelihood). Last but not least, the development team is friendly and helpful, and based on users' feedbacks, NIMBLE folks work constantly at improving the package capabilities.

2.3 NIMBLE workflow

To run NIMBLE, you will need to specify three things: (1) a model (likelihood and priors), (2) the data, (3) those parameters you want to say something about, (4) initial values and (5) MCMC details (number of chains, length of the burn-in period and number of iterations following burn-in).

But first things firt, and do not forget to load the `nimble` package.

```
library(nimble)
```

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

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

In the code above, the ~ means distributed as.

BUGS is a declarative language for graphical (or hierarchical) models.

Most programming languages are imperative, which means a series of commands will be executed in the order they are written. A declarative language like BUGS is more like building a machine before using it. Each line declares that a component should be plugged into the machine, but it doesn't matter in what order they are declared as long as all the right components are plugged in by the end of the code.

The machine in this case is a graphical model12. A node (sometimes called a vertex) holds one value, which may be a scalar or a vector. Edges define the relationships between nodes. A huge variety of statistical models can be thought of as graphs.

Here is the code to define and create a simple linear regression model with four observations

' and deterministic relationships are declared with '<-'. For example, each y[i] follows a normal distribution with mean predicted.y[i] and standard deviation sigma. Each predicted.y[i] is the result of intercept + slope * x[i]. The for-loop yields the equivalent of writing four lines of code, each with a different value of i. It does not matter in what order the nodes are declared. Imagine that each line of code draws part of Figure 5.1, and all that matters is that the everything gets drawn in the end. Available distributions, default and alternative parameterizations, and functions are listed in Section 5.2.4.

NIMBLE calls non-stochastic nodes 'deterministic', whereas BUGS ca

The model definition consists of a series of relations inside a block delimited by curly brackets { and } and preceded by the keyword model. Here is a simple linear regression example:

Each relation defines a node in the model. The node on the left of a relation is defined in terms of other nodes – referred to as parent nodes – that appear on the right hand side. Taken together, the nodes in the model form a directed acyclic graph (with the parent/child relationships represented as directed edges). The very top-level nodes in the graph, with no parents, are constant nodes, which are defined either in the model definition (e.g. 1.0E-3), or in the data when the model is compiled (e.g. x[1]). Relations can be of two types. A stochastic relation (~) defines a stochastic node, repre- senting a random variable in the model. A deterministic relation (<-) defines a deterministic node, the value of which is determined exactly by the values of its parents. The equals sign (=) can be used for a deterministic relation in place of the left arrow (<-).

R2jags is that we can specify the model by creating a special kind of function.6 The avoids the need to create temporary files (as rjags requires) and keeps things tidier in our R markdown documents.

Describe distributions. And also nodes. Stochastic. Deterministic. And distributed as. Provide list of built-in distributions? You can provide your own, see e.g.

Read in data.

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

Distinguish constants and data. To Nimble, not all "data" is data…

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

Constants: + Can never be changed + Must be provided when a model is defined (part of the model structure) + E.g. vector of known index values, variables used to define for-loops, etc.

After defining the model code, we should define the constants, initial values and data list. Compared to WinBUGS and JAGS, data and initial values can be defined in the same way, while 'constants' is a new list that contains the values that would not change, including the variables that define for-loop indices. In our settings, the lists of data, constants and initial values are given as follows:

Data: + Can be changed without re-building the model + Can be (re-)simulated within a model + E.g. stuff that *only* appears to the left of a "~"

For computational efficiency, better to specify as much as possible as constants. NIMBLE will help you with this!

We can also control the starting point for the chains. Starting different chains and quite different parameter values can help

verify that the MCMC algorithm is not overly sensitive to where we are starting from, and ensure that the MCMC algorithm has explored the posterior distribution sufficiently.

On the other hand, if we start a chain too far from the peak of the posterior distribution, the chain may have trouble converging.

We can provide either specific starting points for each chain or a function that generates random starting points.

Specify initial values.

```
initial.values <- function() list(theta = runif(1,0,1))
```

```
initial.values()
## $theta
## [1] 0.9331
```

Which parameters to save? Define parameters to keep track of (i.e., parameters of interest).

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

MCMC details

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

Number of posterior samples per chain:

$$n.posterior = \frac{n.iter - n.burnin}{n.thin}$$

Run model, tadaa!

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

Details on messages received.

Say we do not thin. But ok if you'd like, just think of $n.posterior = \frac{n.iter - n.burnin}{n.thin}$.

Proposer le même modèle avec la bernoulli pour montrer une boucle. The binomial is just a sum of Bernoulli outcomes. Like flipping a coin for each individual and get a survivor with prob phi. Comme dans annexe Hobbs. Vectorize also.

2.3.1 Post-process MCMC outputs by hand

```
str(mcmc.output)
## List of 2
##  $ chain1: num [1:4000, 1] 0.39 0.39 0.39 0.374 0.328 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr "theta"
##  $ chain2: num [1:4000, 1] 0.321 0.374 0.334 0.334 0.334 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr "theta"
```

```
head(mcmc.output$chain1)
##        theta
## [1,] 0.3900
## [2,] 0.3900
## [3,] 0.3900
## [4,] 0.3738
## [5,] 0.3282
## [6,] 0.3203
```

banana-book_files/figure-latex/unnamed-chunk-41-1.pdf

2.3.2 Post-process MCMC outputs without pain

We use MCMCvis, but there are other perfectly valid options out there like ggmcmc and basicMCMCplots.

Numerical summaries.

```
library(MCMCvis)
MCMCsummary(mcmc.output, round = 2)
##       mean   sd 2.5%  50% 97.5% Rhat n.eff
## theta 0.34 0.06 0.22 0.34  0.47    1  1754
```

Trace and posterior density

```
MCMCtrace(mcmc.output,
          pdf = FALSE)
```

banana-book_files/figure-latex/unnamed-chunk-43-1.pdf

```
MCMCtrace(mcmc.output,
          pdf = FALSE,
          ind = TRUE,
          Rhat = TRUE,
          n.eff = TRUE)
```

banana-book_files/figure-latex/unnamed-chunk-44-1.pdf

2.4 Syntax: what's new/better/different?

basculer des trucs de speed up ici

- Vectorization

```
# JAGS (& Nimble)
for(t in 1:Tmax){
  x[t] <- Mu.x + epsilon[t]
}

# Nimble
x[1:Tmax] <- Mu.x + epsilon[1:Tmax]
```

- More flexible specification of distributions

```
# JAGS (& Nimble)
for(t in 1:Tmax){
  epsilon[t] ~ dnorm(0, tau)
}
tau <- pow(sigma, -2)
sigma ~ dunif(0, 5)
```

```
# Nimble
for(t in 1:Tmax){
  epsilon[t] ~ dnorm(0, sd = sigma)
}
sigma ~ dunif(0, 5)
```

- Your own functions and distributions

```
x[1:Tmax] <- myNimbleFunction(a = Mu.x, b = epsilon[1:Tmax])
```

```
sigma ~ dCustomDistr(c = 0.5, z = 10)
```

- The end of empty indices

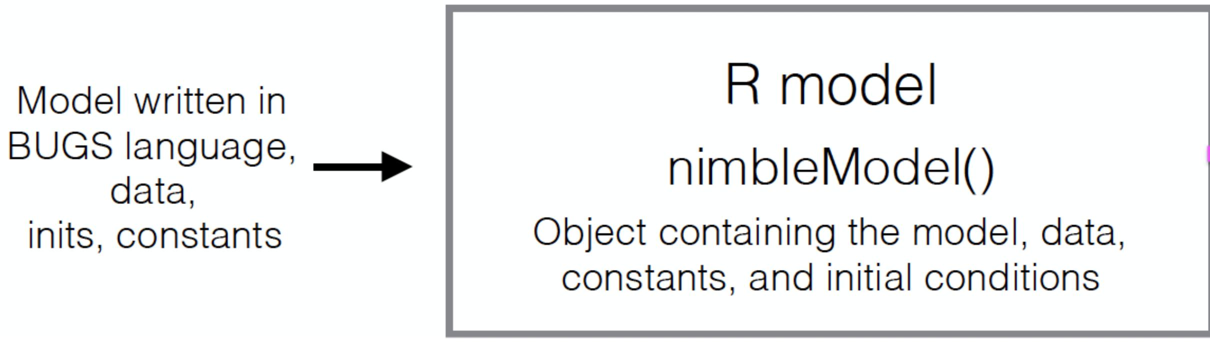
```
# JAGS
sum.x <- sum(x[])

# Nimble
sum.x <- sum(x[1:Tmax])
```

- & more…

2.5 Our `nimble` workflow so far

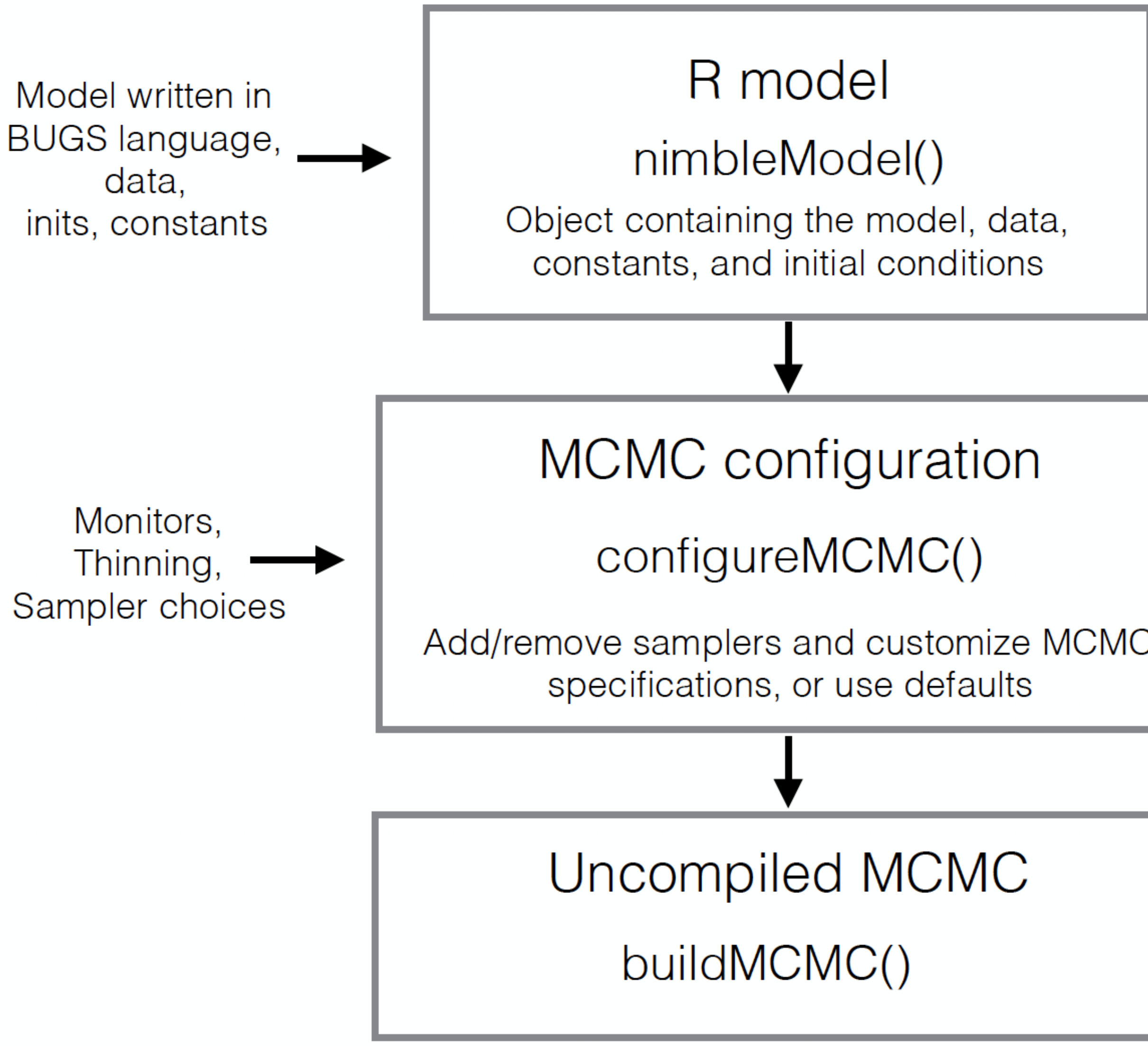
```
knitr::include_graphics("images/nimble_workflow_sofar.png")
```


Model written in BUGS language, data, inits, constants
R model
nimbleModel()
Object containing the model, data, constants, and initial conditions

Adapted from L. Ponisio

But `nimble` gives full access to the MCMC engine

```r
knitr::include_graphics("images/nimble_workflow.png")
```


Model written in BUGS language, data, inits, constants
R model
nimbleModel()
Object containing the model, data, constants, and initial conditions
Monitors, Thinning, Sampler choices
MCMC configuration
configureMCMC()
Add/remove samplers and customize MCMC specifications, or use defaults
Uncompiled MCMC
buildMCMC()

Credit: L. Ponisio

```
knitr::include_graphics("images/I1bIY06.gif")
```

2.6 Functions

Say we want an R function that adds 2 to every value in a vector.

```
add2 <- function(x) {
   x + 2
}
Radd2 <- nimbleRcall(function(x = double(0)){},
                     Rfun = 'add2',
                     returnType = double(0))
demoCode <- nimbleCode({
  mu ~ dnorm(0,1)
  for(i in 1:n) {
    x[i] ~ dnorm(mu, sd = 1)
    z[i] <- Radd2(x[i])
    }
})

param_names <- c("mu", "z")
mcmc.out <- nimbleMCMC(code = demoCode,
                      constants = list(n = 4),
                      data = list(x = c(-1, -2, 1, 2)),
                      inits = list(mu = rnorm(1)),
                      monitors = param_names,
                      nchains = 2,
                      niter = 1000,
                      nburnin = 500)
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
library(MCMCvis)
MCMCsummary(object = mcmc.out, round = 2)
##       mean   sd 2.5% 50% 97.5% Rhat n.eff
## mu   -0.01 0.46 -0.9   0  0.87    1   993
## z[1]  1.00 0.00  1.0   1  1.00  NaN     0
## z[2]  0.00 0.00  0.0   0  0.00  NaN     0
## z[3]  3.00 0.00  3.0   3  3.00  NaN     0
## z[4]  4.00 0.00  4.0   4  4.00  NaN     0
```

Change format to vectorise.

```
add2 <- function(x) {
   x + 2
}
Radd2 <- nimbleRcall(function(x = double(1)){},
                     Rfun = 'add2',
                     returnType = double(1))
demoCode <- nimbleCode({
  mu ~ dnorm(0,1)
  for(i in 1:n) {
    x[i] ~ dnorm(mu, sd = 1)
    }
    z[1:4] <- Radd2(x[1:4])
})

param_names <- c("mu", "z")
mcmc.out <- nimbleMCMC(code = demoCode,
                      constants = list(n = 4),
                      data = list(x = c(-1, -2, 1, 2)),
                      inits = list(mu = rnorm(1)),
                      monitors = param_names,
                      nchains = 2,
                      niter = 1000,
                      nburnin = 500)
```

```
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
library(MCMCvis)
MCMCsummary(object = mcmc.out, round = 2)
##       mean   sd  2.5% 50% 97.5% Rhat n.eff
## mu       0 0.45 -0.86   0  0.85    1  1150
## z[1]     1 0.00  1.00   1  1.00  NaN     0
## z[2]     0 0.00  0.00   0  0.00  NaN     0
## z[3]     3 0.00  3.00   3  3.00  NaN     0
## z[4]     4 0.00  4.00   4  4.00  NaN     0
```

Now have paramater to estimate as parameter of your R function.

```
add2 <- function(x) {
   x + 2
}
Radd2 <- nimbleRcall(function(x = double(0)){},
                     Rfun = 'add2',
                     returnType = double(0))
demoCode <- nimbleCode({
  mu ~ dnorm(0,1)
  for(i in 1:n) {x[i] ~ dnorm(mu, sd = 1)}
  z <- Radd2(mu)
})

param_names <- c("mu", "z")
mcmc.out <- nimbleMCMC(code = demoCode,
                      constants = list(n = 4),
                      data = list(x = c(-1, -2, 1, 2)),
                      inits = list(mu = rnorm(1)),
                      monitors = param_names,
                      nchains = 2,
                      niter = 1000,
                      nburnin = 500)
```

```
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
## |-------------|-------------|-------------|-------------|
## |-------------------------------------------------------|
library(MCMCvis)
MCMCsummary(object = mcmc.out, round = 2)
##      mean   sd  2.5% 50% 97.5% Rhat n.eff
## mu      0 0.46 -0.92   0  0.87    1  1179
## z       2 0.46  1.08   2  2.87    1  1179
```

In general if you do need a nimbleRcall like this, there are a couple of considerations. It is common to need to write a wrapper function, i.e. a function you access via nimbleRcall that calls your actual function of interest with arguments and then return value rearranged as needed. For example, if you just need the eigenvectors, a wrapper function could pick those out and return them. The bigger issue is the returnType declaration: nimble type declarations do not include an R list of type declarations as a nimble type. I think you could use a nimbleList data structure for this purpose. You would have to create a nimbleList type and then use that as the declared returnType. But you would still need to write a wrapper, so that you could convert the list returned from base::eigen into a nimbleList object to return from your wrapper. I hope that makes sense.

https://kenkellner.com/blog/models-with-integrals.html

Same thing w/ global environment.

```
library(nimble)
add2 <- function(x) {
   x + 2 + globvar
}
add2(2)
globvar <- 2020
add2(2)
Radd2 <- nimbleRcall(function(x = double(0)){},
                     Rfun = 'add2',
```

```r
                     returnType = double(0))
demoCode <- nimbleCode({
  mu ~ dnorm(0,1)
  for(i in 1:n) {x[i] ~ dnorm(mu, sd = 1)}
  z <- Radd2(mu)
})

param_names <- c("mu", "z")
mcmc.out <- nimbleMCMC(code = demoCode,
                       constants = list(n = 4),
                       data = list(x = c(-1, -2, 1, 2)),
                       inits = list(mu = rnorm(1)),
                       monitors = param_names,
                       nchains = 2,
                       niter = 1000,
                       nburnin = 500)
#printErrors()
# pb is y is not recognized
#ls()
# assign y to global env
# https://stackoverflow.com/questions/9726705/assign-multiple-objects-to-globalenv-from-within-
#assign("globvar", 20, envir = .GlobalEnv)

library(MCMCvis)
MCMCsummary(object = mcmc.out, round = 2)
```

2.7 Code your own sampler

```r
library(nimble)
load('matos/ressources-chapters/nimble/dipper_data.Rdata')

dipperCode <- nimbleCode({
    logit.p ~ dnorm(0, 0.001)
    logit.phi ~ dnorm(0, 0.001)
    p <- expit(logit.p)
    phi <- expit(logit.phi)
    ##phi ~ dunif(0, 1)
    ##p ~ dunif(0, 1)
    for(i in 1:N) {
        x[i, first[i]] <- 1
        y[i, first[i]] <- 1
        for(t in (first[i]+1):T) {
            x[i, t] ~ dbern(phi * x[i, t-1])
            y[i, t] ~ dbern(p * x[i, t])
        }
    }
})

N <- dim(sightings)[1]
T <- dim(sightings)[2]
dipperConsts <- list(N = N, T = T, first = first)
dipperData <- list(y = sightings)
xInit <- ifelse(!is.na(sightings), 1, 0)
dipperInits <- list(logit.phi = 0, logit.p = 0, x = xInit)

samples <- nimbleMCMC(dipperCode, dipperConsts, dipperData, dipperInits,
                      niter = 10000, nburnin = 5000,
                      monitors = c('p', 'phi'))

my_MH <- nimbleFunction(
    name = 'my_MH',
    contains = sampler_BASE,
    setup = function(model, mvSaved, target, control) {
        calcNodes <- model$getDependencies(target)
        scale <- control$scale
    },
    run = function() {
        initialLP <- model$getLogProb(calcNodes)
        current <- model[[target]]
        proposal <- rnorm(1, current, scale)
        model[[target]] <<- proposal
        proposalLP <- model$calculate(calcNodes)
        lMHR <- proposalLP - initialLP
        if(runif(1,0,1) < exp(lMHR)) {
            ## accept
            copy(from = model, to = mvSaved, nodes = calcNodes, logProb = TRUE, row = 1)
        } else {
            ## reject
            copy(from = mvSaved, to = model, nodes = calcNodes, logProb = TRUE, row = 1)
        }
    },
    methods = list(
        reset = function() {}
    )
)

scale <- 0.05

Rmodel <- nimbleModel(dipperCode, dipperConsts, dipperData, dipperInits)
conf <- configureMCMC(Rmodel, monitors = c('p', 'phi'))
conf$printSamplers()
conf$printSamplers(byType = TRUE)
conf$removeSamplers(c('logit.p', 'logit.phi'))
conf$addSampler(target = 'logit.p', type = 'my_MH', control = list(scale = scale))
conf$addSampler(target = 'logit.phi', type = 'my_MH', control = list(scale = scale))
conf$printSamplers()
conf$printMonitors()
Rmcmc <- buildMCMC(conf)

out <- compileNimble(list(model=Rmodel, mcmc=Rmcmc))
Cmcmc <- out$mcmc

samples2 <- runMCMC(Cmcmc, niter = 10000, nburnin = 5000)
```

```
samplesSummary(samples2)

basicMCMCplots::chainsPlot(samples2)
```

2.8 A dire quelque part?

Pourquoi Nimble plutôt que Stan? Syntaxe BUGS, also discrete latent states easier to deal with, no need to marginalise. In Stan you marginalise (ref forward to relevant section of the book), but difficult endeavour, and you do not need to do that with NIMBLE, it has algorithms that work fine with discrete latent states.

2.9 When things go wrong: Tip and tricks

2.10 Summary

- Blabla.
- Reblabla.

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

2.11 Suggested reading

- Official website `https://r-nimble.org`
- User Manual `https://r-nimble.org/html_manual/cha-welcome-nimble.html` and cheatsheet[1].
- Users mailing list `https://groups.google.com/forum/#!forum/nimble-users`
- Training material `https://github.com/nimble-training`
- Reference to cite 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[2]. *Journal of Computational and Graphical Statistics* **26** (2): 403–13.

[1] `https://r-nimble.org/cheatsheets/NimbleCheatSheet.pdf`
[2] `https://arxiv.org/pdf/1505.05093.pdf`

3

Hidden Markov models

–> –>

–> –> –> –> –>

Part II

II. Transitions

Introduction

4

Survival

–> –>

–>

–>

–>

–> –> –>

–>

–> –> –> –>

5 *Covariates*

6

Dispersal

7

Model selection and validation

Part III

III. States

Introduction

8 State uncertainty

9 Hidden semi-Markov models

Part IV

IV. Case studies

Introduction

10

Life history theory

10.1 Tradeoffs

?, ?, and ?

10.2 Breeding dynamics

?, ?, ?, and ?

10.3 Actuarial senescence

?, ?

10.4 Cause-specific mortalities

? and ?

10.5 Disease dynamics

? and ?

10.6 Sex uncertainty

? and **?**

11
Abundance

11.1 Horvitz-Thompson

?

11.2 Jolly-Seber

11.3 Robust design

?, **?**, **?**, and **?**

12

Stopover duration

?

13

Individual dependence

13.1 Dependence among individuals

? and **?**

13.2 Individual heterogeneity

?, **?**, and **?**

Part V

V. Conclusion

Take-home messages

–> –>

–>

–>

–>

FAQ