Biometrical Journal **52** (2020) 61, zzz–zzz / DOI: 10.1002/bimj.200100000

Fast parameter and confidence interval estimation for Hidden Markov Models using Template Model Builder

Timothee Bacri[*1], **Jan Bulla** [1], and **Geir D. Berentsen** [2]

[1] Department of Statistics, University of Bergen, 5007 Bergen, Norway

[2] Department of Business and Management Science, Norwegian School of Economics, Helleveien 30, 5045 Bergen, Norway

Received zzz, revised zzz, accepted zzz

Hidden Markov Models (HMMs) are a class of models widely used in speech recognition (See e.g. Gales and Young, 2008; Fredkin and Rice, 1992). There are straightforward ways to compute Maximum Likelihood Estimates (MLE) of their parameters, but getting confidence intervals can be more difficult (See e.g. Zucchini et al., 2016; Lystig and Hughes, 2002). In addition, computing MLEs can be time-consuming when the dataset and the complexity become very large. We show in this paper a way to speed up computation by up to 50 times in the language R when compared to usual optimization approaches, and at the same time retrieve standard errors easily. In a first part, we see how to optimize HMMs with the `TMB` package in R and how to retrieve confidence intervals. In a second part, we compare different optimizers such as `nlminb`, and minimize the negative log-likelihood directly on different datasets: a small one (240 data points) from Leroux and Puterman (1992), a medium sized simulated one (2000 data points), and a large one (87648 data points) from a hospital.

Key words: Hidden Markov Model; TMB; Confidence intervals;

Supporting Information for this article is available from the author or on the WWW under https://github.com/timothee-bacri/hmm-tmb

1 Introduction

Hidden Markov models (HMMs) are a versatile type of model that have been employed in many different situations since their introduction by Baum and Petrie (1966). As an example, Fredkin and Rice (1992) applied them in speech recognition, Lystig and Hughes (2002) to rainfall occurence data, and Schadt et al. (1998) to phylogenetic trees. Fredkin and Rice (1992) and Zucchini et al. (2016) are references in the theory of HMMs. Evaluating uncertainty and getting confidence intervals in HMMs is uneasy and not necessarily reliable. Although Cappé et al. (2006, Chapter 12) showed it can be achieved using asymptotic normality of the MLEs of the parameters under certain conditions, Frühwirth-Schnatter (2006, p. 53) points out that in independent mixture models, "The regularity conditions are often violated". McLachlan and Peel (2004, p. 68) adds that "In particular for mixture models, it is well known that the sample size n has to be very large before the asymptotic theory of maximum likelihood applies." Lystig and Hughes (2002) shows a way to compute the exact Hessian, and Zucchini et al. (2016) shows another way to compute the approximate Hessian and thus confidence intervals but admits that "the use of the Hessian to compute standard errors (and thence confidence intervals) is unreliable if some of the parameters are on or near the boundary of their parameter space".

`TMB` (Template Model Builder) is an R package for efficient fitting of complex statistical random effect models to data, as described by Kristensen et al. (2015). It provides exact calculations of first and second

*Corresponding author: e-mail: timothee.bacri@uib.no, Phone: +33-636-775-063

order derivatives of the likelihood of a model by automatic differentiation, which allows for efficient gradient and/or Hessian based optimization of the likelihood as well as uncertainty estimates by means of the Hessian. The Hessian is not necessarily directly applicable for evaluating parameter uncertainty in HMMs as there are several parameter constraints in these models. This can be adressed by constraint optimization, and subsequently combining the Hessian with the Jacobian of the constraints to obtain the covariance matrix as shown by Visser et al. (2000). Alternatively, Zucchini et al. (2016) shows how the constraints can be imposed by suitable transformations of the parameters. The covariance matrix of the untransformed (original) parameters can then be retrieved by the delta method, a feature implemented in `TMB`.

In this paper, we first show how to optimize Poisson HMMs with the help of `TMB` in the language R. Afterwards, we explain how to make a nested model through an example, how to effortlessly compute confidence intervals, how to fetch interesting probabilities, and we apply a Poisson HMM on a real hospital dataset. Eventually, we see that `TMB` can accelerate traditional optimizers in R by up to approximately 50 times on a fairly large dataset, and can easily output confidence intervals similar to bootstrap and profile methods via its Hessian based approach.

Maximum likelihood estimation can be achieved either by a direct numerical maximization as introduced by Turner (2008) and later detailed with R code by Zucchini et al. (2016), by an Expectation-Maximization (EM) based algorithm as described by Baum et al. (1970), or by a mixture of the 2 as shown by Bulla and Berzel (2008). We decide to use a direct maximization approach instead of the EM algorithm. The reason is that the direct maximization approach is easier to adapt if one wants to fit different and more complex models. It also deals easily with missing observations, whereas the EM approach is more complex.

2 Principles of using `TMB` for Maximum Likelihood Estimation (MLE)

TBD: mention structure / concept: carry out functions available at GitHub in parallel while reading this tutorial to keep an acceptable length. One folder / file? for each section

2.1 Setup

Execution of our routines requires the installation of the R-package `TMB` and the software `Rtools`, where the latter serves for compiling C++ code. In order to ensure reproducibility of all results involving the generation of random numbers, the `set.seed` function requires R version number 3.6.0 or greater. Our scripts were tested on an Intel(R) Core(TM) i7-8700 processor running under Windows 10 Enterprise version 1809.

In particular for beginners, those parts of scripts involving C++ code can be difficult to debug because the code operates using a specific template. Therefore it is helpful to know that `TMB` provides a debugging feature, which can be useful to retrieve diagnostic error messages, in RStudio. Enabling this feature is optional and can be achieved by the command `TMB:::setupRStudio()` (requires manual confirmation and re-starting RStudio).

2.2 Linear regression example

We begin by demonstrating the principles of `TMB`, which we illustrate through the fitting procedure for a simple linear model. This permits, among other things, to show how to handle parameters subject to constraints, an aspect particularly relevant for HMMs. A more comprehensive tutorial on `TMB` presenting many technical details in more depths is available at
`https://kaskr.github.io/adcomp/_book/Tutorial.html`.

Let $\boldsymbol{x}$ and $\boldsymbol{y}$ the predictor and response vector, respectively, both of length n. For a simple linear regression model with intercept a and slope b, the negative log-likelihood equals

$$-l(a, b, \sigma^2) = -\sum_{i=1}^{n} \log(\phi(y_i; a + bx_i, \sigma^2))),$$

where $\phi(\cdot; \mu, \sigma^2)$ corresponds to the density function of the univariate normal distribution with mean μ and variance σ^2.

The use of `TMB` requires the (negative) log-likelihood function to be coded in C++ under a specific template, which is then loaded into R. The minimization of this function and other post-processing procedures are all carried out in R. Therefore, we require two files. The first file, named *linreg.cpp*, is written in C++ and defines the negative log-likelihood (nll) function of the linear model as follows.

```
#include <TMB.hpp> //import the TMB template

template<class Type>
Type objective_function<Type>::operator() ()
{
  DATA_VECTOR(y); // Data vector y passed from R
  DATA_VECTOR(x); // Data vector x passed from R

  PARAMETER(a);           // Parameter a passed from R
  PARAMETER(b);           // Parameter b passed from R
  PARAMETER(tsigma);      // Parameter sigma (transformed, on log-scale)
                          // passed from R

  // Transform tsigma back to natural scale
  Type sigma = exp(tsigma);

  // Declare negative log-likelihood
  Type nll = -sum(dnorm(y,
                        a + b * x,
                        sigma,
                        true));

  // Necessary for inference on sigma, not only tsigma
  ADREPORT(sigma);

  return nll;
}
```

Note that we define data inputs x and y using the `DATA_VECTOR()` declaration in the above code. Furthermore, we declare the nll as a function of the three parameters a, b and $\log(\sigma)$ using the `PARAMETER()` declaration. In order to be able to carry out a wide range of (unconstrained) optimization procedures in the following, the nll is parameterized in terms of $\log(\sigma)$. While the parameter σ is constrained to be non-negative, $\log(\sigma)$ can be freely estimated. Alternatively, constraint optimization methods could be carried out, but we won't investigate such procedures. The `ADREPORT()` function is optional but useful for parameter inference at the postprocessing stage.

The second file needed is written in R and serves for compiling the nll function defined above and carrying out the estimation procedure by numerical optimization of the nll function. The .R file (shown below) carries out the compilation of the C++ file and minimization of the objective function:

```
# Setting up TMB
library(TMB)
# Compilation. The compiler returns 0 if the compilation of
# the cpp file was successful
TMB::compile("code/linreg.cpp")

## [1] 0

# Dynamic loading of the compiled cpp file (TIMO: right?)
dyn.load(dynlib("code/linreg"))
# Generate the data for our test sample
set.seed(123)
data <- list(y = rnorm(20) + 1:20, x = 1:20)
parameters <- list(a = 0, b = 0, tsigma = 0)
# Instruct TMB to create the likelihood function
obj_linreg <- MakeADFun(data, parameters, DLL = "linreg",
                        silent = TRUE)
# Optimization of the objective function with nlminb
mod_linreg <- nlminb(obj_linreg$par, obj_linreg$fn)
mod_linreg$par

##           a           b      tsigma
##  0.31009240  0.98395535 -0.05814659
```

In addition to the core functionality presented above, different types of post-processing of the results is possible as well. For example, the function `sdreport` returns the estimates and standard errors of the variables in terms of which the nll is parameterized:

```
sdreport(obj_linreg)

## sdreport(.) result
##           Estimate Std. Error
## a       0.31009240 0.43829083
## b       0.98395535 0.03658781
## tsigma -0.05814659 0.15811381
## Maximum gradient component: 6.931683e-05
```

The standard errors result from the generalized delta method described by Kass and Steffey (1989), which is implemented within `TMB`. From a practical perspective, it is usually desirable to obtain standard errors for the constrained variables, in this case σ. To achieve this, one can run the `summary` function with argument `select = "report"`:

```
summary(sdreport(obj_linreg), select = "report")

##        Estimate Std. Error
## sigma 0.9435116  0.1491822
```


Full functionality of the `sdreport` function requires calling the function `ADREPORT` on the additional variables of interest (i.e. those including transformed parameters, in our example σ) in the C++ part. The `select` argument restricts the output to variables passed by `ADREPORT`. This feature is particularly useful when the likelihood has been reparameterized as above, and is especially relevant for HMMs. Following Zucchini et al. (2016), we refer to the original parameters as natural parameters, and to their transformed version as the working parameters.
Last, for comparison we display the estimation results from the `lm` function, which turn out to be identical.

```
lm(y ~ x, data = data)$coefficients

## (Intercept)           x
##   0.3100925   0.9839554
```

Note that minor deviations from the results of `lm` originate in the numerical methods involved in the selected optimization procedure, in our case `nlminb`.

3 Estimation `TMB`/HMM

3.1 Notation

A HMM is a model where the data is assumed to follow a mixture of distributions, going from one to the other as time progresses according to probabilities set by an underlying homogeneous stationary Markov chain. In this paper we focus on a Poisson-HMM, but only small changes to the code are needed to obtain models with other conditional distributions than the Poisson. The data denoted as $\{X_n : n = 1, \dots, N\}$ is assumed to follow a mixture of m Poisson distributions with parameters $\{\lambda_i : i = 1, \dots, m\}$.
Let $\{C_t : t = 1, \dots, N\}$ be the underlying Markov chain. The Markov chain is assumed irreducible and aperiodic. Grimmett et al. (2001, Lemma 6.3.5 on p. 225 and Theorem 6.4.3 on p. 227) has shown that irreducibility ensures the existence of the stationary distribution. As has been shown by Feller (1968, p. 394), aperiodicity implies that a unique limiting distribution exists and is the stationary distribution.
Let $p_i(x) = P(X_t = x|C_t = i), \forall i = 1, \dots, m$ and let $\mathbf{P}(x) = \begin{pmatrix} p_1(x) & & & 0 \\ & p_2(x) & & \\ & & \ddots & \\ 0 & & & p_m(x) \end{pmatrix}$.
We call $\begin{pmatrix} p_1(x_1) & p_2(x_1) & \dots & p_m(x_1) \\ p_1(x_2) & p_2(x_2) & \dots & p_m(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ p_1(x_n) & p_2(x_n) & \dots & p_m(x_n) \end{pmatrix}$ the emission probability matrix. If x is a missing data, we define $p_i(x) = 1$ and therefore the emission matrix of the dataset contain a line filled with ones. Furthermore, we let $\mathbf{\Gamma} = \{\gamma_{ij}\}$ denote the transition matrix of the Markov chain, and $\boldsymbol{\delta}$ be its stationary distribution. Finally, we let $X^{(t)} = \{X_1, \dots, X_t\}$ and $x^{(t)} = \{x_1, \dots, x_t\}$ denote the history up to time t.

In this paper we focus on Poisson HMMs, and so $p_i(x) = \frac{e^{-\lambda_i}\lambda_i^x}{x!}$.
This can however be easily changed to other distributions.

3.2 HMM Likelihood

It can be shown (Zucchini et al., 2016, p. 36) that the likelihood of an HMM is given by

$$L_N = \mathbf{P}(X^{(t)} = x^{(t)}) = \boldsymbol{\delta}\mathbf{P}(x_1)\mathbf{\Gamma}\mathbf{P}(x_2)\mathbf{\Gamma}\mathbf{P}(x_3)\dots\mathbf{\Gamma}\mathbf{P}(x_N)\mathbf{1}'. \quad (1)$$

.

It is usual for HMMs to estimate the parameters $\boldsymbol{\theta}$ (where $\boldsymbol{\theta} = (\gamma_{11}, \ldots, \gamma_{1m}, \ldots, \gamma_{m1}, \ldots, \gamma_{mm}, \lambda_1, \ldots, \lambda_m)$) subject to estimation in the case of a Poisson HMM) by optimizing a reparametrized version of the log-likelihood $\log L(\boldsymbol{\psi})$, where $\boldsymbol{\psi} = g^{-1}(\boldsymbol{\theta})$ represent a set of unconstrained parameters. That transformation is detailed below. As a result of the invariance principle, the maximum likelihood estimate of $\boldsymbol{\theta}$ is given by $\hat{\boldsymbol{\theta}} = g(\hat{\boldsymbol{\psi}})$ where $\hat{\boldsymbol{\psi}}$ is the minimizer of $-\log L(\boldsymbol{\psi})$. To avoid constraints on the parameters, we transform the natural parameters $\{\boldsymbol{\Gamma}, \boldsymbol{\lambda}\}$ into working parameters $\{\mathbf{T}, \boldsymbol{\eta}\}$.

In practice, as explained in Section 2, we first create a set of natural parameters and turn them into working parameters before feeding those in an optimizer.
One possible transformation of $\boldsymbol{\Gamma}$ is done this way:

$$\gamma_{ij} = \frac{\exp(\tau_{ij})}{1 + \sum_{k \neq i} \tau_{ik}}, \text{ for } i \neq j$$

Each row must add to 1, so we can get the diagonal elements of $\boldsymbol{\Gamma}$ easily.
$\eta_i = \log(\lambda_i)$.

The reverse transformation is

$$\tau_{ij} = \log\left(\frac{\gamma_{ij}}{1 - \sum_{k \neq i} \gamma_{ik}}\right) = \log(\gamma_{ij}/\gamma_{ii}), \text{ for } i \neq j$$

And $\lambda_i = \exp(\eta_i)$.

The relevant R function to do so is given by Zucchini et al. (2016, p. 52), and an version adapted for use with `TMB` can be found in the supporting information, in the file *utils.R*.

Such a transformation can be adapted to other conditional distributions. Note that the parts concerning the Markov chain remain unchanged. The working parameters returned then serve as initial values for starting the optimization procedure of the likelihood via `TMB`. As illustrated in the previous section, we focus on unconstrained optimization, similar Zucchini et al. (2016, p. 50). Hence, we parameterize the likelihood function as a function of the working parameters. The computation of the likelihood then requires to transform the working parameters back to natural parameters before calculating the negative log-likelihood via Equation 1. This part is entirely set up in the C++ likelihood function.

Since the transformation of the transition probability Matrix (TPM) is identical for all HMMs, it is stored as a separate C++ function `Gamma_w2n` which is available in the file *utils.cpp* (Appendix A.2 item (iii)). This file also contains the C++ function `Delta_w2n`, which if necessary, we can define to transform the working stationary distribution vector into a natural parameter (Appendix A.2 item (ii)). It was unneeded for this paper.

Given a working parameter vector `log_lambda`, the C++ function turning the working vector of the Poisson means into a natural parameter is much simpler:

```
vector<Type> lambda = tlambda.exp();
```

The "forward algorithm" allows for a recursive computation of the likelihood. To state the forward algorithm let us define the vector $\boldsymbol{\alpha}_t$ for $t = 1, 2, \ldots, N$ so that

$$\begin{aligned}
\boldsymbol{\alpha}_t &= \boldsymbol{\delta}\mathbf{P}(x_1)\boldsymbol{\Gamma}\mathbf{P}(x_2)\boldsymbol{\Gamma}\mathbf{P}(x_3)\ldots\boldsymbol{\Gamma}\mathbf{P}(x_t) \\
&= \boldsymbol{\delta}\mathbf{P}(x_1)\prod_{s=2}^{t}\boldsymbol{\Gamma}\mathbf{P}(x_s) \\
&= (\alpha_t(1), \ldots, \alpha_t(m))
\end{aligned}$$

where $\boldsymbol{\delta}$ denotes the initial distribution of the Markov chain. By convention, the empty product is the identity matrix. It is called the forward algorithm because of the way the $\boldsymbol{\alpha}_t$ values are calculated

$$\begin{aligned}
&\boldsymbol{\alpha}_0 = \boldsymbol{\delta} \\
&\boldsymbol{\alpha}_t = \boldsymbol{\alpha}_{t-1}\boldsymbol{\Gamma}\mathbf{P}(x_t) \text{ for } t = 1, 2, \ldots, N \\
&L_T = \boldsymbol{\alpha}_T \mathbf{1}'.
\end{aligned}$$

In the same way, the "backward algorithm" allows for a recursive computation of the likelihood. To state the backward algorithm let us define the vector $\boldsymbol{\beta}_t$ for $t = 1, 2, \ldots, N$ so that

$$\begin{aligned}
\boldsymbol{\beta}_t' &= \boldsymbol{\Gamma}\mathbf{P}(x_{t+1})\boldsymbol{\Gamma}\mathbf{P}(x_{t+2}) \ldots \boldsymbol{\Gamma}\mathbf{P}(x_T) \ldots \mathbf{1}' \\
&= \left(\prod_{s=t+1}^{N} \boldsymbol{\Gamma}\mathbf{P}(x_s) \right) \mathbf{1}' \\
&= (\beta_t(1), \ldots, \beta_t(m))
\end{aligned}$$

It is called the backward algorithm because of the way the $\boldsymbol{\beta}_t$ values are calculated:

$$\begin{aligned}
&\boldsymbol{\beta}_T = \mathbf{1}' \\
&\boldsymbol{\beta}_t = \boldsymbol{\Gamma}\mathbf{P}(x_{t+1})\boldsymbol{\beta}_{t+1} \text{ for } t = N-1, N-2, \ldots, 1 \\
&L_T = \boldsymbol{\delta}\boldsymbol{\beta}_1.
\end{aligned}$$

Computing those probabilities directly can lead to underflow errors, it is therefore better to maximize the log-likelihood instead. We choose to minimize the negative log-likelihood for convenience purposes. In addition, in our case $\boldsymbol{\Gamma}$ and $\boldsymbol{\lambda}$ are subject to some constraints:

(i) The means λ_i of the state-dependent distributions must be non-negative, for $i = 1, \ldots, m$

(ii) The parameters $\gamma_{i,j}$ of the transition probability matrix $\boldsymbol{\Gamma}$ must be non-negative, and the rows of $\boldsymbol{\Gamma}$ must add to one

It should be noted that we use a scaled version of the forward algorithm in order to calculate the likelihood, as suggested by Zucchini et al. (2016, p. 48).

In summary, we need to define the state-dependent probabilities (from Poisson distributions in our case) and the transformations in R and C++ (depending on the distributions used, but can be easily adapted). We show below the next step: writing the likelihood function in C++ and using it to model a dataset.

3.3 Using TMB

3.3.1 Likelihood function

As we did with the linear regression example, we first need to define our objective function: the negative log-likelihood, in a C++ file. Let's name it *poi_hmm.cpp*. Its code can be found in the Appendix A.2 item (i).

3.3.2 Optimization

Once the objective function is written in C++, we can instruct R to optimize its parameters in a few steps.

(i) Loading packages


```
# Load TMB and optimization packages
library(TMB)
library(optimr)
# Run the C++ file containing the TMB code
TMB::compile("code/poi_hmm.cpp")

## [1] 0

# Load it
dyn.load(dynlib("code/poi_hmm"))
# Load the parameter transformation function
source("functions/utils.R")
```

(ii) Loading the dataset, see Section 7.1 for more details on the datasets.

```
load("data/fetal-lamb.RData")
lamb_data <- lamb
```

(iii) Creating natural parameters as initial values for the optimizer.
We use the optimal amount of hidden states (2) according to the BIC that Leroux and Puterman (1992) reported. Although the AIC selects 3 hidden states, our goal is to show a simple example.

```
# Model with 2 states
m <- 2
TMB_data <- list(x = lamb_data, m = m)

# Generate initial set of parameters for optimization
lambda <- c(1, 3)
gamma <- matrix(c(0.8, 0.2,
                  0.2, 0.8), byrow = TRUE, nrow = m)
```

(iv) Transforming them to working parameters

```
# Turn them into working parameters
parameters <- pois.HMM.pn2pw(m, lambda, gamma)
```

(v) Creating the `TMB` likelihood function object

```
obj_tmb <- MakeADFun(TMB_data, parameters, DLL = "poi_hmm",
                     silent = TRUE)
```

(vi) Optimizing

```
mod_tmb <- nlminb(start = obj_tmb$par, objective = obj_tmb$fn)
# Check that it converged successfully
mod_tmb$convergence == 0

## [1] TRUE
```


We are using `nlminb` because it is the fastest we have tested. Details are available in Section 7.2 item (iii).

(vii) Getting the ML estimates of the natural parameters and their standard errors, sent by `ADREPORT` It should be noted that the `gamma` parameter is shown column-wise below.

```
summary(sdreport(obj_tmb), "report")

##          Estimate Std. Error
## lambda 0.25636530 0.04016451
## lambda 3.11475069 1.02131434
## gamma  0.98872130 0.01063573
## gamma  0.31033874 0.18468645
## gamma  0.01127870 0.01063573
## gamma  0.68966126 0.18468645
## delta  0.96493132 0.03181447
## delta  0.03506868 0.03181447
```

(viii) It is possible to instruct the optimizer to run with the gradient and/or hessian computed by `TMB`:

```
# The negative log-likelihood is accessed by the objective
# attribute of the optimized object
mod_tmb$objective

## [1] 177.5188

mod_tmb <- nlminb(start = obj_tmb$par, objective = obj_tmb$fn,
                  gradient = obj_tmb$gr, hessian = obj_tmb$he)
mod_tmb$objective

## [1] 177.5188
```

Feeding to `nlminb` an exact gradient and hessian provided by `TMB` can make the model fit better, although in this case it make no difference.

The dataset used is provided by Leroux and Puterman (1992). Although our estimates are close to the paper's estimates, they're not exactly the same. The reason is that their likelihood is altered while ours isn't. This is clear when comparing the likelihoods with only 1 state.

A 1 state Poisson HMM is the same as a Poisson regression model, for which the log-likelihood has the expression

$$\begin{aligned} l(\lambda) &= \log\left(\prod_{i=1}^{n} \frac{\lambda^{x_i} e^{-\lambda}}{x_i!}\right) \\ &= -n\lambda + \log(\lambda)\left(\sum_{i=1}^{n} x_i\right) - \sum_{i=1}^{n} \log(x_i!). \end{aligned}$$

The authors find a ML estimate $\lambda = 0.3583$ and a log-likelihood of -174.26. In contrast, calculating the log-likelihood explicitly shows a different result.


```
x <- lamb_data
n <- length(x)
l <- 0.3583
- n * l + log(l) * sum(x) - sum(log(factorial(x)))

## [1] -201.0436
```

The log-likelihood is different, but when the constant $-\sum_{i=1}^{n} \log(x_i!)$ is removed, it matches our result.

```
- n * l + log(l) * sum(x)

## [1] -174.2611
```

In this paper, we use the complete formula for the negative log-likelihood, which therefore differs sligthly from Leroux and Puterman (1992).

3.4 Computing the stationary distribution

Within the objective function in Section 3.3.1, the stationary distribution of the m state HMM's Markov chain with transition probability matrix $\mathbf{\Gamma}$ is calculated. In this section, we explain that calculation.

Zucchini et al. (2016) shows that calculating the stationary distribution can be achieved by solving Equation 2 for $\boldsymbol{\delta}$, where $\mathbf{I}_m$ is the $m * m$ identity matrix, $\mathbf{U}$ is a $m * m$ matrix of ones, and $\mathbf{1}$ is a row vector of ones.

$$\boldsymbol{\delta}(\mathbf{I}_m - \mathbf{\Gamma} + \mathbf{U}) = \mathbf{1} \tag{2}$$

An implementation of this in R is shown here

```
# Compute the stationary distribution of a Markov chain
# with transition probability gamma
stat.dist <- function(gamma) {
  m <- dim(gamma)[1]
  return(solve(t(diag(m) - gamma + 1), rep(1, m)))
}
```

and used in the supporting information.

In order to use it in `TMB`, an implementation in C++ is necessary under a specific template. The file *utils.cpp* (Appendix A.2 item (iv)) shows how to achieve this.

3.5 Nested model specification

Nested models can be useful for multiple reasons. For example, there can be a need to fix some parameters because of some biological or physical constraints.

Using a nested model solves this issue at the cost of having a worse fit. The reason is that some parameters are fixed and the others' estimates have different values than in the original model. Since the estimates can vary, the fit worsens.

To showcase the advantage of nested models, we re-use the 2 state model.

`TMB` can be instructed to treat some parameters as constants. However, only working parameters can be fixed. Nevertheless, in our situation we can fix a natural parameter (λ) for which we can easily identify the

corresponding working parameter ($\log(\lambda)$) to fix, hence we do that. Fixing a value in the transition probability matrix might be possible, but is clearly troublesome given the lack of a one-to-one correspondence with the natural parameters, so we fix a Poisson mean instead.

We arbitrarily fix λ_1 to 1.

```
# Get the previous values, and fix some
fixed_par_lambda <- lambda
fixed_par_lambda[1] <- 1
fixed_par_gamma <- gamma
```

Now that the nested model's natural parameters are constructed, we transform them into a set of working parameters.

```
# Transform them into working parameters
new_parameters <- pois.HMM.pn2pw(m = m,
                                 lambda = fixed_par_lambda,
                                 gamma = fixed_par_gamma)
```

In order for `TMB` to treat parameters as constants, the `map` argument of the `MakeADFun` function must be a list. This list must contain named vectors filled with `NA` for each fixed parameters, and unique factor levels for the regular parameters.

Equal factor levels are collected to common values, so we want a different factor level for each non-fixed parameter, and NA for each fixed parameter. We can use increasing numbers to make factors, and replace with NA where necessary.

```
map <- list(tlambda = as.factor(c(NA, 1)),
            tgamma = as.factor(c(2, 3)))

# The map is fed to the MakeADFun function
fixed_par_obj_tmb <- MakeADFun(TMB_data, new_parameters,
                               DLL = "poi_hmm",
                               silent = TRUE,
                               map = map)
fixed_par_mod_tmb <- nlminb(start = fixed_par_obj_tmb$par,
                            objective = fixed_par_obj_tmb$fn,
                            gradient = fixed_par_obj_tmb$gr,
                            hessian = fixed_par_obj_tmb$he)
```

The estimates and standard errors vary after fixing the parameters (Table 1) and the standard errors for the fixed parameters are zero.

Since the nested model isn't the optimal model, the likelihood becomes worse, going from

```
# Original model negative log-likelihood
mod_tmb$objective

## [1] 177.5188
```

to

	Original model		Nested model	
	Estimate	Std. Error	Estimate	Std. Error
λ_1	0.26	0.04	1.00	0.00
λ_2	3.11	1.02	3.44	0.86
$\gamma_{1,1}$	0.99	0.01	1.00	0.00
$\gamma_{2,1}$	0.31	0.18	0.19	0.18
$\gamma_{1,2}$	0.01	0.01	0.00	0.00
$\gamma_{2,2}$	0.69	0.18	0.81	0.18
δ_1	0.96	0.03	0.98	0.03
δ_2	0.04	0.03	0.02	0.03

Table 1 2 state Poisson HMM before and after fixing λ_1 to 1 using a nested model

```
# Nested model negative log-likelihood
fixed_par_mod_tmb$objective

## [1] 264.1636
```

Note that some inconsistencies can happen.

The stationary distribution is a vector of probabilities and should sum to 1. However, it doesn't behave as expected.

```
adrep <- summary(sdreport(obj_tmb), "report")
estimate_delta <- adrep[rownames(adrep) == "delta", "Estimate"]
sum(estimate_delta)

## [1] 1

sum(estimate_delta) == 1

## [1] FALSE
```

This is likely due to machine approximations when numbers far apart from each other interact together. In R, a small number is not 0 but is treated as 0 when added to a much larger number.

```
1e-100 == 0

## [1] FALSE

(1 + 1e-100) == 1

## [1] TRUE
```

This can result in incoherent findings when checking that 2 numbers are equal. Fortunately, no issue arose from this.

4 Confidence intervals

4.1 Hessian based confidence intervals

First, let us consider evaluation of parameter uncertainty via the Hessian. As the Hessian $\nabla^2 \log L(\hat{\psi})$ refers to the transformed parameters ψ, the delta-method must be used to obtain an estimate of the covariance matrix of $\hat{\theta}$:

$$\Sigma_{\hat{\theta}} = -\nabla g(\hat{\psi}) \left(\nabla^2 \log L(\hat{\psi}) \right)^{-1} \nabla g(\hat{\psi})' \tag{3}$$

With minimal effort from the user's perspective, `TMB` can be instructed to calculate $\Sigma_{\hat{\theta}}$ (by automatic differentiation). Standard errors of derived parameters, such as the stationary distribution $\boldsymbol{\delta}$ which is a function of $\boldsymbol{\Gamma}$, can be calculated by the delta-method similarly.

`TMB` provides an easy way to retrieve these. Following the example above, as we saw in Section 3.3.2,

```
# Get all standard errors
adrep <- summary(sdreport(obj_tmb), "report")
adrep

##          Estimate Std. Error
## lambda 0.25636541 0.04016445
## lambda 3.11475432 1.02131181
## gamma  0.98872128 0.01063571
## gamma  0.31033853 0.18468648
## gamma  0.01127872 0.01063571
## gamma  0.68966147 0.18468648
## delta  0.96493123 0.03181445
## delta  0.03506877 0.03181445

# More help with ?summary.sdreport
```

It should be noted that although the estimates can be found in the optimization variable `opt` in their working form, they are also sent to the `MakeADFun` object `obj` and can be retrieved as shown above. `sdreport` shows information about the working parameters' estimates, whereas the summary of that report also shows information about the variables reported by the function `ADREPORT`.

The next part shows how to get Wald confidence intervals as defined by Wald (1943), using `TMB`. With the previous 2 state Poisson HMM, the $100(1-\alpha)\%$ confidence interval for a is $a \pm z_{1-\alpha/2} * \sigma_a$ where z_x is the x-percentile of the standard normal distribution, and σ_a is the standard error of a.

```
adrep <- summary(sdreport(obj_tmb), "report")

# Get the 97.5 percentile of the standard normal distribution
q95_norm <- qnorm(1 - 0.05 / 2)

# Create the confidence interval
# Extract the values
estimates <- adrep[, "Estimate"]
std_errors <- adrep[, "Std. Error"]
# Create the bounds
lower_bound <- estimates - q95_norm * std_errors
upper_bound <- estimates + q95_norm * std_errors
```



```
# Show the CI
cbind(lower_bound, upper_bound)

##          lower_bound upper_bound
## lambda  0.177644535  0.33508628
## lambda  1.113019955  5.11648869
## gamma   0.967875672  1.00956689
## gamma  -0.051640322  0.67231737
## gamma  -0.009566886  0.03212433
## gamma   0.327682625  1.05164032
## delta   0.902576056  1.02728641
## delta  -0.027286406  0.09742394
```

Estimates of $\boldsymbol{\theta}$ and $\boldsymbol{\delta}$, with accompanying confidence intervals are displayed in Table 12 and Table 13.

As mentioned earlier, for a larger amount of hidden states, `TMB` may be unable to give some or any standard standard errors because some variables are close to their boundaries. In that situation, using a nested model might solve this issue. We refer the reader to Section 3.5 for information on how to specify a nested model using `TMB`.

4.2 Likelihood profile based confidence intervals

Next, we consider evaluating uncertainty using likelihood-profiles. Assuming normality of the maximum likelihood (ML) estimators, confidence intervals for the parameters can be obtained using the above estimates of standard deviations. For a fixed sample size, the validity of the normality assumption can often be violated and in these cases a likelihood-based CI is more robust (reference). Let η be single parameter in a model with parameters (η, θ) and likelihood $L(\eta, \theta)$, and let $L_p(\eta)$ be the profile likelihood defined by $L_p(\eta) = \max_\theta L(\eta, \theta)$. Then a likelihood-based CI for η is given by

$$\left\{ \eta : 2\log(\frac{L(\hat{\eta}, \hat{\theta})}{L_p(\eta)} < \chi^2_{1,(1-\alpha)} \right\} \tag{4}$$

where $\chi^2_{1,(1-\alpha)}$ is the $1-\alpha$ quantile of a χ^2 distribution with 1 degree of freedom. `TMB` allows for very efficient computation of both the profile likelihood $L(\eta)$ and the CI given by Equation 4, and this has been used to produce Table 12 and Table 13 which display the profile log-likelihood and the corresponding likelihood-based CI's in the model for the lamb and the simulated dataset. Note that the problem of transformed parameters is easier to deal with when making likelihood-based CI's, since $\{g(\eta), L_p(g(\eta))\} = \{g(\eta), L_p(\eta)\}$ for any one-to-one function g (invariance principle).

Again, `TMB` provides an easy way to obtain those intervals. The `name` argument allows to profile parameter. The `trace` argument indicates how much information on the optimization you want the function to show. Following the HMM example, if we wish to profile the 2^{nd} working parameter `log_lambda2`, we have to feed its position into the `name` argument:

```
profile <- tmbprofile(obj = obj_tmb,
                      name = 2,
                      trace = FALSE)

# Confidence interval of lambda
exp(confint(profile))

##             lower    upper
## tlambda 1.265339 4.947733
```


It should be noted that the argument `lincomb` allows to profile any linear combination of parameters by using a vector of the linear coefficients in the same order as the parameters.
This method can sometimes give NA values, usually when a ML estimate is close to a boundary, but not only. For example, the first working parameter `log_lambda` is pretty low (-4.47). It becomes too difficult to profile the likelihood so the function fails to provide a meaningful confidence interval.

Another important issue is about profiling with a univariate model. With only 1 (hidden) state, a Poisson HMM becomes a univariate Poisson regression model. Therefore, the profile becomes a plot of the likelihood when the Poisson mean varies. However, `tmbprofile` fails to provide a confidence interval. The reason is likely that it tries to optimize the likelihood despite the lack of parameters to change, and therefore fails. Visser et al. (2000), Meeker and Escobar (1995) and Venzon and Moolgavkar (1988) provide more details on profiling likelihoods.

4.3 Bootstrap based confidence intervals

Finally, we consider evaluating uncertainty using bootstrap. There are many ways to bootstrap. Non-parametric bootstraping of time series is possible, as Härdle et al. (2003) has shown. However our focus is not on bootstrapping techniques, so we choose a parametric approach. See Efron and Tibshirani (1993) for more details on bootstrapping.

From the parameters' ML estimates, we generate new data and re-estimate the parameters 500 times. From that list of new estimates we can get the 2.5th and 97.5th percentiles and get 95% confidence intervals for the parameters.

We show below how we get confidence intervals using bootstrap, based on the 3 state Poisson HMM estimates from above.

(i) First, we need a function to generate random data from a HMM.

```
# Generate a random sample from a HMM
pois.HMM.generate_sample  <- function(ns, mod) {
  mvect <- 1:mod$m
  state <- numeric(ns)
  state[1] <- sample(mvect, 1, prob = mod$delta)
  for (i in 2:ns) {
    state[i] <- sample(mvect, 1, prob = mod$gamma[state[i - 1], ])
  }
  x <- rpois(ns, lambda = mod$lambda[state])
  return(x)
}
```

(ii) Then, when the model is estimated each time, we don't impose an order for the states. This can lead to the label switching problem, where states aren't ordered the same way in each model. To address this, we re-ordered the states by ascending Poisson means.
Sorting the means is pretty straightforward. Re-ordering the TPM is a little trickier. To do so, we took the permutations of the states given by the sorted Poisson means, and permuted each row index and column index to its new value.
The function we used is

```
# Relabel states by increasing Poisson means
pois.HMM.label.order <- function(m, lambda, gamma, delta = NULL) {
  # Get the indexes of the sorted states
```



```
  # according to ascending lambda
  sorted_lambda <- sort(lambda, index.return = TRUE)$ix
  # Re-order the TPM according to the switched states
  # in the sorted lambda
  ordered_gamma <- matrix(0, nrow = m, ncol = m)
  for (col in 1:m) {
    new_col <- which(sorted_lambda == col)
    for (row in 1:m) {
      new_row <- which(sorted_lambda == row)
      ordered_gamma[row, col] <- gamma[new_row, new_col]
    }
  }
  # Re-order the stationary distribution if it was provided
  # Generate it otherwise
  if (is.null(delta)) {
    delta <- stat.dist(ordered_gamma)
  } else {
    delta <- delta[sorted_lambda]
  }
  return(list(lambda = sort(lambda),
              gamma = ordered_gamma,
              delta = delta))
}
```

Let's show an example to understand the process. For readability, the TPM is filled with row and column indexes instead of probabilities.

```
lambda <- c(30, 10, 20)
gamma <- matrix(c(11, 12, 13,
                  21, 22, 23,
                  31, 32, 33), byrow = TRUE, ncol = 3)
pois.HMM.label.order(m = 3, lambda, gamma)

## $lambda
## [1] 10 20 30
##
## $gamma
##      [,1] [,2] [,3]
## [1,]   33   31   32
## [2,]   13   11   12
## [3,]   23   21   22
##
## $delta
## [1] -0.032786885  0.016393443 -0.008196721
```

State 1 has been relabeled state 3, state 2 became state 1, and state 3 became state 2.

(iii) Bootstrap code


```
bootstrap_estimates <- data.frame()
DATA_SIZE <- length(lamb_data)
# Set how many parametric bootstrap samples we create
BOOTSTRAP_SAMPLES <- 10

# ML parameters
ML_working_estimates <- obj_tmb$env$last.par.best
ML_natural_estimates <- obj_tmb$report(ML_working_estimates)
gamma <- ML_natural_estimates$gamma
lambda <- ML_natural_estimates$lambda
delta <- ML_natural_estimates$delta

# Parameters for TMB
cols <- names(ML_working_estimates)
tgamma <- ML_working_estimates[cols == "tgamma"]
tlambda <- ML_working_estimates[cols == "tlambda"]
ML_TMB_parameters <- list(tlambda = tlambda,
                          tgamma = tgamma)

params_names <- c("lambda", "gamma", "delta")

for (idx_sample in 1:BOOTSTRAP_SAMPLES) {
  # Loop as long as there is an issue with nlminb
  repeat {
    #simulate the data
    bootstrap_data <- pois.HMM.generate_sample(DATA_SIZE,
                                               list(m = m,
                                                    lambda = lambda,
                                                    gamma = gamma,
                                                    delta = delta))

    # Parameters for TMB
    TMB_data_bootstrap <- list(x = bootstrap_data, m = m)

    # Estimate the parameters
    obj <- MakeADFun(TMB_data_bootstrap,
                     ML_TMB_parameters,
                     DLL = "poi_hmm",
                     silent = TRUE)

    # Using tryCatch to break the loop if an error happens
    # interrupts the outer (for) loop, not the inner one (repeat)
    mod_temp <- NULL
    try(mod_temp <- nlminb(start = obj$par, objective = obj$fn,
                           gradient = obj$gr, hessian = obj$he),
        silent = TRUE)

    # If nlminb doesn't reach any result, retry
    if (is.null(mod_temp)) {
```



```
      next
    }
    # If nlminb doesn't converge successfully, retry
    if (mod_temp$convergence != 0) {
      next
    }
    working_parameters <- obj$env$last.par.best
    natural_parameters <- obj$report(working_parameters)

    l <- natural_parameters$lambda
    g <- natural_parameters$gamma
    d <- natural_parameters$delta

    # Label switching
    natural_parameters <- pois.HMM.label.order(m, l, g, d)

    # If some parameters are NA for some reason, retry
    if (anyNA(natural_parameters[params_names], recursive = TRUE)) {
      next
    }

    # If everything went well, end the "repeat" loop
    break
  }
  # The values from gamma are taken columnwise
  natural_parameters <- unlist(natural_parameters[params_names])
  len_par <- length(natural_parameters)
  bootstrap_estimates[idx_sample, 1:len_par] <- natural_parameters
}

# Lower and upper 95% bounds
q <- apply(bootstrap_estimates, 2, function(par_estimate) {
  quantile(par_estimate, probs = c(0.025, 0.975))
})

params_names <- paste0(rep("lambda", m), 1:m)
# Get row and column indexes for gamma instead of the default
# columnwise index: the default indexes are 1:m for the 1st column,
# then (m + 1):(2 * m) for the 2nd, etc...
for (gamma_idx in 1:m ^ 2) {
  row <- (gamma_idx - 1) %% m + 1
  col <- (gamma_idx - 1) %/% m + 1
  row_col_idx <- c(row, col)
  params_names <- c(params_names,
                    paste0("gamma",
                           paste0(row_col_idx, collapse = "")))
}
params_names <- c(params_names,
                  paste0(rep("delta", m), 1:m))
```



```
bootstrap_CI <- data.frame("Parameter" = params_names,
                           "Estimate" = c(lambda, gamma, delta),
                           "Lower bound" = q[1, ],
                           "Upper bound" = q[2, ])
print(bootstrap_CI, row.names = FALSE)

##  Parameter   Estimate  Lower.bound Upper.bound
##    lambda1 0.25636541 7.117339e-09   0.2954041
##    lambda2 3.11475432 3.565353e-01   3.4379317
##    gamma11 0.98872128 4.444232e-01   0.9947810
##    gamma21 0.31033853 9.937752e-02   0.5693026
##    gamma12 0.01127872 5.219006e-03   0.5555768
##    gamma22 0.68966147 4.306974e-01   0.9006225
##     delta1 0.96493123 2.635814e-01   0.9830019
##     delta2 0.03506877 1.699805e-02   0.7364186
```

It should be noted that some estimates can be very large or small. This can happen when the randomly generated bootstrap sample contains long chains of the same values. However, a large number of bootstrap samples lowers that risk since we ”leave out” 5% of the extreme values in the CI. It could be useful to skip the bootstrap samples where the estimates are too far out of expectations.

5 State inference

5.1 Setup

Given an optimized `MakeADFun` object `obj`, we need to setup some variables to compute the probabilities detailed below.

```
# Retrieve the objects at ML value
adrep <- obj_tmb$report(obj_tmb$env$last.par.best)
delta <- adrep$delta
gamma <- adrep$gamma
emission_probs <- adrep$emission_probs
n <- adrep$n
m <- length(delta)
mllk <- adrep$mllk
```

5.2 Log-forward probabilities

The forward probabilities have been detailed in Section 3.2. We show here a way to compute the log of the forward probabilities, using a scaling scheme defined by Zucchini et al. (2016).

```
# Compute log-forward probabilities (scaling used)
lalpha <- matrix(NA, m, n)
foo <- delta * emission_probs[1, ]
sumfoo <- sum(foo)
lscale <- log(sumfoo)
foo <- foo / sumfoo
lalpha[, 1] <- log(foo) + lscale
```



```
for (i in 2:n) {
  foo <- foo %*% gamma * emission_probs[i, ]
  sumfoo <- sum(foo)
  lscale <- lscale + log(sumfoo)
  foo <- foo / sumfoo
  lalpha[, i] <- log(foo) + lscale
}
# lalpha contains n=240 columns, so we only display 5 for readability
lalpha[, 1:5]

##            [,1]       [,2]       [,3]      [,4]      [,5]
## [1,] -0.2920639 -0.5591179 -0.8265948 -1.094088 -1.361583
## [2,] -6.4651986 -7.7716769 -8.1146145 -8.385232 -8.652850
```

5.3 Log-backward probabilities

The backward probabilities have been defined in the same section.

```
# Compute log-backwards probabilities (scaling used)
lbeta <- matrix(NA, m, n)
lbeta[, n] <- rep(0, m)
foo <- rep (1 / m, m)
lscale <- log(m)
for (i in (n - 1):1) {
  foo <- gamma %*% (emission_probs[i + 1, ] * foo)
  lbeta[, i] <- log(foo) + lscale
  sumfoo <- sum(foo)
  foo <- foo / sumfoo
  lscale <- lscale + log(sumfoo)
}
# lbeta contains n=240 columns, so we only display 4 for readability
lbeta[, 1:4]

##           [,1]      [,2]      [,3]      [,4]
## [1,] -177.2275 -176.9600 -176.6925 -176.4250
## [2,] -178.3456 -178.0781 -177.8099 -177.5253
```

5.4 Smoothing probabilities

The smoothing probabilities are defined in Zucchini et al. (2016) as $P(C_t = i|X^{(N)} = x^{(N)}) = \frac{\alpha_t(i)\beta_t(i)}{L_T}$.

```
# Compute conditional state probabilities, smoothing probabilities
stateprobs <- matrix(NA, ncol = n, nrow = m)
llk <- - mllk
for(i in 1:n) {
  stateprobs[, i] <- exp(lalpha[, i] + lbeta[, i] - llk)
}
```



```
# Most probable states
ldecode <- rep(NA, n)
for (i in 1:n) {
  ldecode[i] <- which.max(stateprobs[, i])
}
ldecode
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [32] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [63] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 1 1 1
##  [94] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [125] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [156] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [187] 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [218] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

5.5 Forecast, h-step-ahead-probabilities

The forecast distribution or h-step-ahead-probabilities as well as its implementation in R is detailed in Zucchini et al. (2016).
Let $\boldsymbol{\varphi}_N = \frac{\boldsymbol{\alpha}_N}{\boldsymbol{\alpha}_N \mathbf{1}'}$.
Then,

$$P(X_{N+h} = x | X^{(N)} = x^{(N)}) = \frac{\boldsymbol{\alpha}_N \boldsymbol{\Gamma}^h \mathbf{P}(x) \mathbf{1}'}{\boldsymbol{\alpha}_N \mathbf{1}'} = \boldsymbol{\varphi}_N \boldsymbol{\Gamma}^h \mathbf{P}(x) \mathbf{1}'.$$

An implementation of this, using a scaling scheme is

```
# Number of steps
h <- 1
# Values for which we want the forecast probabilities
xf <- 0:50

nxf <- length(xf)
dxf <- matrix(0, nrow = h, ncol = nxf)
foo <- delta * emission_probs[1, ]
sumfoo <- sum(foo)
lscale <- log(sumfoo)
foo <- foo / sumfoo
for (i in 2:n) {
  foo <- foo %*% gamma * emission_probs[i, ]
  sumfoo <- sum( foo)
  lscale <- lscale + log(sumfoo)
  foo <- foo / sumfoo
}
emission_probs_xf <- get.emission.probs(xf, lambda)
for (i in 1:h) {
  foo <- foo %*% gamma
  for (j in 1:m) {
```



```
    dxf[i, ] <- dxf[i, ] + foo[j] * emission_probs_xf[, j]
  }
}
# dxf contains n=240 columns, so we only display 4 for readability
dxf[, 1:4]

## [1] 0.765294660 0.197684763 0.027659770 0.004772415
```

5.6 Global decoding using the Viterbi algorithm

The Viterbi algorithm is detailed in Zucchini et al. (2016). It calculates the sequence of states $(i_1^*, \ldots, i_N^*)$ which maximizes the conditional probability of all states simultaneously, i.e.

$$(i_1^*, \ldots, i_N^*) = \underset{i_1,\ldots,i_N \in \{1,\ldots,m\}}{\arg\max} P(C_1 = i_1, \ldots, C_N = i_N | X^{(N)} = x^{(N)}).$$

An implementation of it is

```
xi <- matrix(0, n, m)
foo <- delta * emission_probs[1, ]
xi[1, ] <- foo / sum(foo)
for (i in 2:n) {
  foo <- apply(xi[i - 1, ] * gamma, 2, max) * emission_probs[i, ]
  xi[i, ] <- foo / sum(foo)
}
iv <- numeric(n)
iv[n] <- which.max(xi[n, ])
for (i in (n - 1):1){
  iv[i] <- which.max(gamma[, iv[i + 1]] * xi[i, ])
}
iv

##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [32] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [63] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 1 1 1 1 1
##  [94] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [125] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [156] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [187] 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [218] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

6 Application to hospital data

6.1 Hospital data description

We timed m states HMM parameter estimations using different parameters on a dataset provided by a hospital from Assistance Publique – Hôpitaux de Paris, a french hospital trust, with $m = 1, 2, 3, 4$. Other datasets are considered in Section 7.

The dataset contains the number of patients who entered a hospital each hour between the first hour of 2010 and the last hour of 2019. The data was anonymized and grouped using the R package `dplyr`. That

data contains 87648 data points.
The column PATIENTS represents the number of people arriving at the hospital between the hour mentioned and the next. The zeroes there indicate that no patient arrived at the hospital during the hour.
The column DATE is a datetime item useful for sorting and plotting the data.
The column WDAY represents the number of the day in the week: 1 is Monday and 7 is Sunday.
The columns YEAR is an integer taking values 2010, 2011, ..., 2019.
For example, the dataset starts on Januray 1^{st} 2010 on the hour 0 with an amount of patients of 6, indicating that 6 people arrived between 00:00 and 00:59.

See the supporting information for details on importing the data.

6.2 Speed comparison

Different numbers of hidden states m were used: ($(1, 2)$ for the lamb dataset, $(1, 2, 3)$ for the simulated dataset, and $(1, 2, 3, 4)$ for the hospital dataset) to compare speeds. Each estimation was timed 5 times, allowing for graphical comparisons through boxplots. The approaches with `TMB` were much faster than without, thus making it necessary to display the figures on a log scale for time. The lamb dataset and the simulated one are used in Section 7, whereas the current section focuses on the dataset about hospital patient arrivals.

For readability, DM denotes direct maximization without using `TMB`, whereas TMB1, TMB2, TMB3, and TMB4 denote direct maximization using `TMB` with and without making use of the exact gradient and hessian that `TMB` can provide. Table 2 summarizes the notation.

Table 2 Naming of TMB parameters

	TMB1	TMB2	TMB3	TMB4
Exact gradient is used	No	No	Yes	Yes
Exact hessian is used	No	Yes	No	Yes

We timed the parameter estimation of m states Poisson HMMs using the approaches DM, TMB1, TMB2, TMB3, and TMB4, with $m = 1, 2, 3, 4$ and found that using `TMB` accelerates the estimation in all cases as can be seen in Figure 1 by the drop in time from DM to any estimation using `TMB`. The times can be found in the supporting information. As could have been expected, estimating a model's parameters takes a longer time as the complexity of the model (m) increases.

Interestingly, for $m > 1$, TMB3 shows a clear acceleration when compared to other estimations using `TMB`. The reason is unclear.

Each box of the boxplots summarizes 5 data points but shows little variation. A percentage mean speed increase of TMB1, ..., TMB4 when compared to the mean times of DM summarizes these graphs nicely (see Table 3). For example, the average 4 state Poisson HMM estimation speed when providing `TMB`'s exact gradient to `nlminb` and not the hessian (TMB3) is 5066% faster than without making use of `TMB` (DM).

TMB also accelerates the likelihood computation time. The percentage speed gains from using `TMB` (Table 4) are all positive, thus showing that on large datasets, the objective function computation time can be accelerated.

6.3 Interpretation

Looking at a boxplot of the data (Figure 2) lets us discover the hourly distribution of the data. We choose to use hours instead of days or other time durations because the difference in arrivals between night and day was already obvious to the doctors themselves, and we would likely find at least two distinct regimes there.

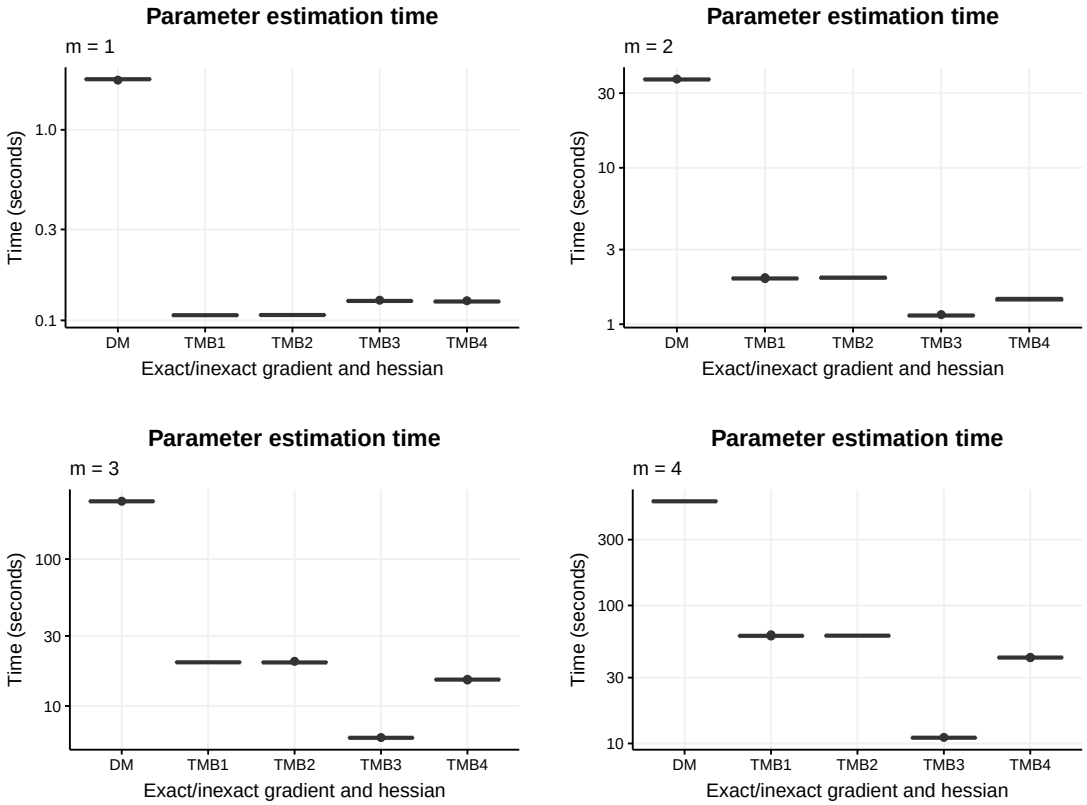

Figure 1 m state Poisson HMM estimation time with and without using TMB, estimated on the hospital dataset. Each procedure was repeated and timed. Boxplots are shown for visual comparison. The naming convention is found in Table 2.

m	TMB1	TMB2	TMB3	TMB4
1	1630	1624	1353	1364
2	1769	1754	3126	2442
3	1145	1144	3963	1534
4	841	842	5066	1256

Table 3 Speed percentage increase of using TMB for m state Poisson HMM parameter estimation, estimated on the hospital dataset. Each procedure was repeated and timed. Their mean times are compared in this table. When m=1, the mean estimation time with TMB by providing the exact gradient but not the exact hessian (TMB3) was 1353% lower than the mean estimation time with direct maximization without TMB (DM). The naming convention is found in Table 2.

More patients arrive in the day rather than during the night as was guessed, but more patients arrive in the afternoon rather than in the evening. Also, most patients show up in the early part of the day and around noon. Given the amount of data (87648/24 = 3652 data points for each hour), we can apply a z-test to see if the means are different.

We find a p-value approximated to 0, so the test shows that the difference between the average amount of patient arrival between 6 and 6:59am and the one between 10 and 10:59am is statistically significant. This

m	TMB1	TMB2	TMB3	TMB4
1	1720	1705	1723	1708
2	1784	1799	1800	1801
3	1145	1150	1147	1143
4	856	856	853	855

Table 4 Speed percentage increase of using TMB for m state Poisson HMM negative log-likelihood calculation, estimated on the hospital dataset. Each procedure was repeated and timed. Their mean times are compared in this table. When m=1, the mean estimation time with TMB by providing the exact gradient but not the exact hessian (TMB3) was 1723% lower than the mean estimation time with direct maximization without TMB (DM). Naming convention is found in Table 2

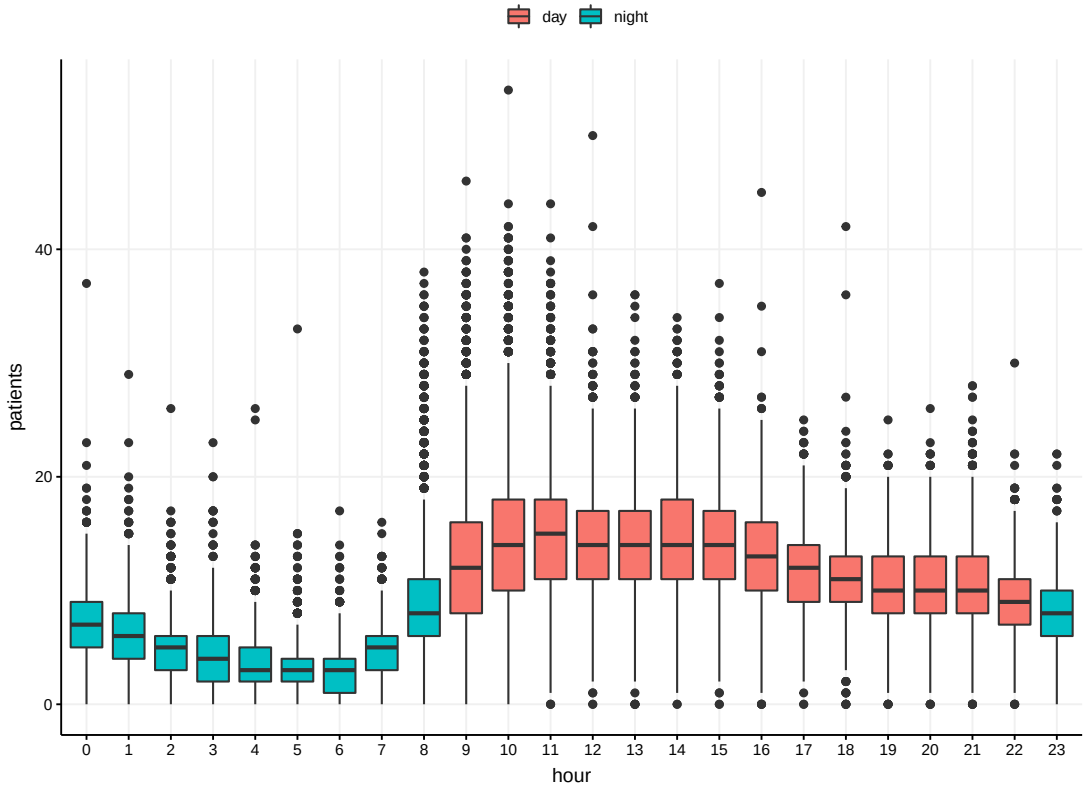

Figure 2 Hourly amount of patient arrival in the hospital

agrees with the conclusion of the doctors and shows that there are likely at least 2 different distributions in the hourly arrivals of patients, justifying an educated guess of 2 or more hidden states.

To choose more accurately the number of hidden states, we tried a few, and looked at the quality of each fit (Table 5). According to the AIC, the BIC, and the negative log-likelihood, the model with $m = 4$ is the most appropriate and we therefore pick it.

Once the 4 state HMM is estimated, we can look at the state distribution of a 4 state Poisson HMM fitted on the hourly arrivals (see Figure 3) across the entire hospital dataset. This HMM seems to agree with our

m	nll	AIC	BIC
1	324288	648578	648587
2	259295	518598	518636
3	247403	494824	494908
4	242587	485206	485356

Table 5 Model selection measures of m state Poisson HMMs estimated on the hospital dataset

previous guess. It should be noted that for the hours 6 and 23, state 4 appeared 0 times and explains why the number 4 is missing in those columns.

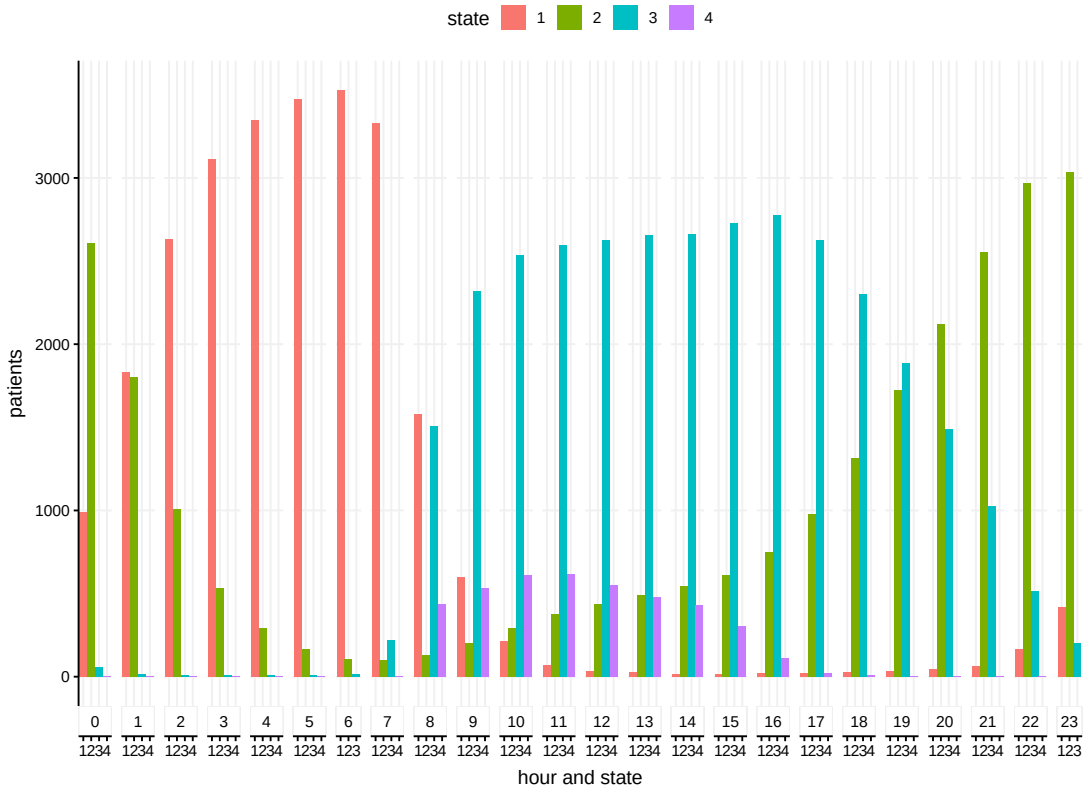

Figure 3 Hourly state distribution of a 4 state Poisson HMM estimated and fitted on the hospital dataset. The 1^{st} row in the horizontal axis denotes the hour, the 2^{nd} row denotes the state. The state is also color coded for clarity.

States (1, 2, 3, 4) have Poisson means (3.816, 8.64, 13.186, 22.503) respectively.

7 Speed evaluation

7.1 Lamb and simulation dataset description

We timed estimations of HMMs using different parameters on a dataset provided by Leroux and Puterman (1992) and on a simulated dataset.

The first dataset is the number of movements by a fetal lamb in 240 consecutive 5-second intervals. The counts are listed here in Table 6 (read from left to right and top to bottom) and are available in the supporting information.

0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 2 2 0 0 0 0 1 0 0 1 1 0 0 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 7 3 2 3 2 4 0 0 0 0 1 0 0 0 0 0 0 0 1 0 2 0 0 0 0 1 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 2 1 0 0 1 0 0 0 1 0 1 1 0 0 0 1 0 0 1 0 0 0 1 2 0 0 0 1 0 1 1 0 1 0 0 2 0 1 2
1 1 2 1 0 1 1 0 0 1 1 0 0 0 1 1 1 0 4 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

Table 6 Numbers of movements by a fetal lamb in 240 consecutive 5-second intervals. Read from left to right, and from top to bottom.

The second dataset is a sequence of random numbers generated by a m-state Poisson HMM. We used the code below to generate that simulated dataset, with $m = 1, 2, 3$.

```
DATA_SIZE_SIMUL <- 2000
m <- 2
# Generate parameters
lambda <- seq(1, 7, length.out = m)
# Create the transition probability matrix with 0.8 on its diagonal
gamma <- matrix(0.2 / (m - 1), nrow = m, ncol = m)
diag(gamma) <- 0.8
delta <- stat.dist(gamma)

#simulate the data
simul_data <- pois.HMM.generate_sample(ns = DATA_SIZE_SIMUL,
                                       mod = list(m = m,
                                                  lambda = lambda,
                                                  gamma = gamma,
                                                  delta = delta))
```

7.2 Speed comparison

(i) Speed of `TMB`

To compare speeds with and without using `TMB`, it is important to test multiple sets of parameters. Hence, we use the R package `microbenchmark`.

Figure 4 shows the time acceleration when using `TMB` on the lamb dataset from Leroux and Puterman (1992) through boxplots. `TMB` accelerates the estimation time for the HMMs, as with the hospital dataset.

The percentage average speed increases from using `TMB` on HMMs estimated on the lamb dataset are all positive (Table 7), thus showing that estimation using `TMB` is faster than a regular estimation procedure without using `TMB` even on a small dataset.

Figure 5 shows the time acceleration when using `TMB` on a simulated dataset through boxplots. Similarly to the other datasets, it shows that using `TMB` results in faster estimations of Poisson HMMs' parameters.

The average speed increases using a simulated dataset (Table 8) shows similar results to the speed gains from the other datasets, showing that `TMB` is useful for estimating Poisson HMMs on medium sized datasets.

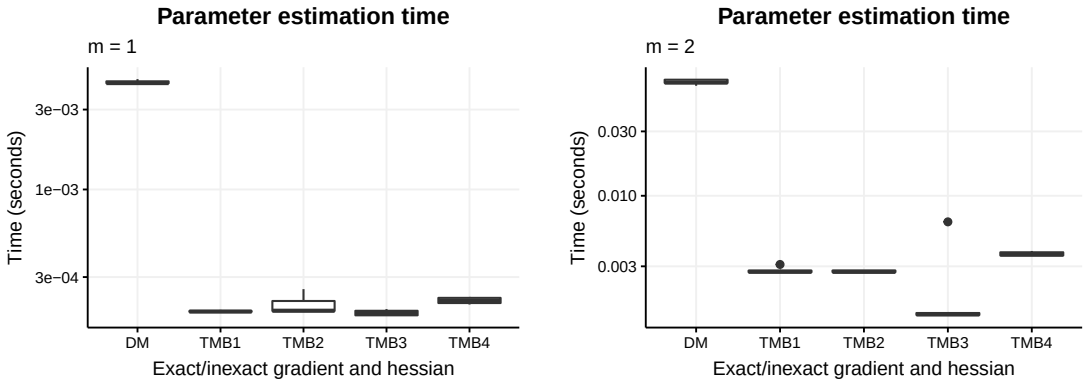

Figure 4 m state Poisson HMMs estimation time with and without using TMB, estimated on the lamb dataset. Each procedure was repeated and timed. Boxplots are shown for visual comparison. The naming convention is found in Table 2.

m	TMB1	TMB2	TMB3	TMB4
1	2236	2018	2284	1922
2	2385	2437	2885	1778

Table 7 Speed percentage increase of using TMB for m state Poisson HMM parameter estimation, estimated on the lamb dataset. Each procedure was repeated and timed. Their mean times are compared in this table. When m=1, the mean estimation time with TMB by providing the exact gradient but not the exact hessian (TMB3) was 2284% lower than the mean estimation time with direct maximization without TMB (DM). The naming convention is found in Table 2.

m	TMB1	TMB2	TMB3	TMB4
1	2437	2480	2290	2026
2	1959	1922	3727	3688
3	1210	1227	4753	2994

Table 8 Speed percentage increase of using TMB for m state Poisson HMM parameter estimation, estimated on the simulated dataset. Each procedure was repeated and timed. Their mean times are compared in this table. When m=1, the mean estimation time with TMB by providing the exact gradient but not the exact hessian (TMB3) was 2290% lower than the mean estimation time with direct maximization without TMB (DM). The naming convention is found in Table 2.

(ii) Log-likelihoods
Optimizing with `TMB` gives the same estimates as optimizing without (see Table 9) when estimating a 2 state Poisson HMM on the lamb dataset. Similar conclusions are obtained for Poisson HMMs estimated on the hospital dataset and simulated datasets.

Moreover, the percentage speed gains from using `TMB` (Table 10 and Table 11) are all positive, thus showing that on small and medium sized datasets, the objective function computation time can be accelerated.

(iii) All optimization methods
Finally, we compare different optimization methods. The ones we retain are BFGS, Nelder-Mead, L-BFGS-B, nlm, nlminb, and hjn, because the others don't converge in our case. `marqLevAlg`

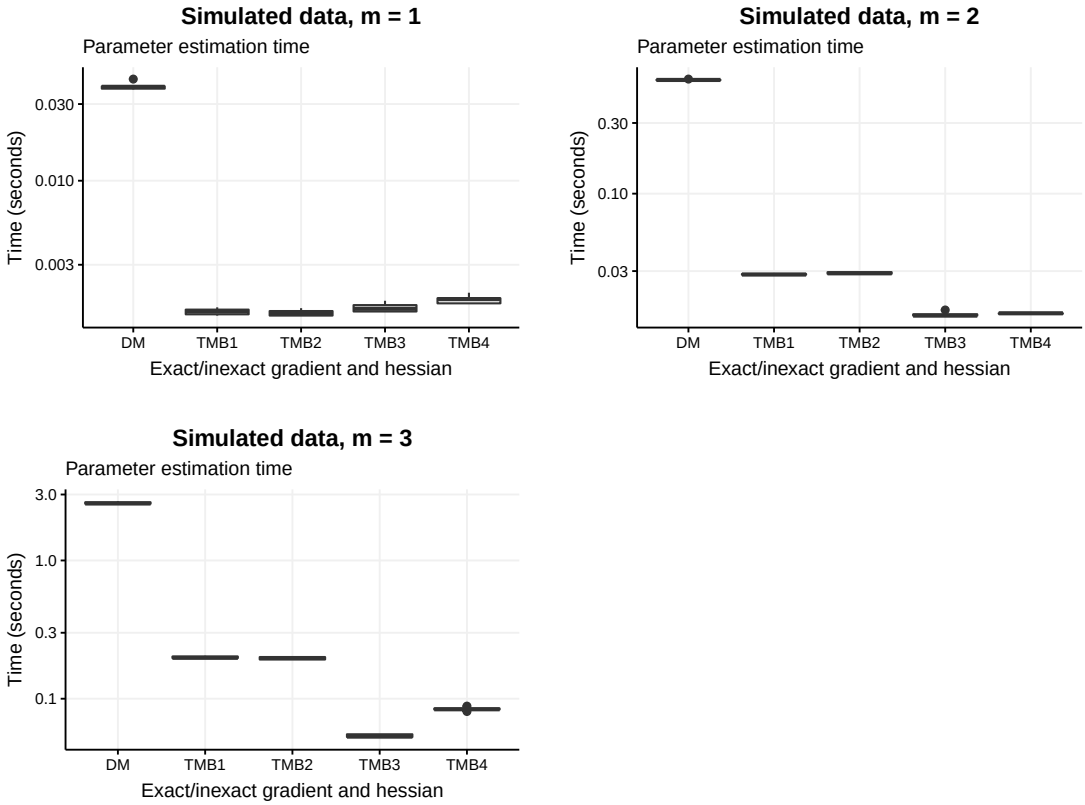

Figure 5 m state Poisson HMM estimation time with and without using TMB, simulated data. Naming convention is found in Table 2

	DM	TMB1	TMB2	TMB3	TMB4
λ_1	0.26	0.26	0.26	0.26	0.26
λ_2	3.11	3.11	3.11	3.11	3.11
$\gamma_{1,1}$	0.99	0.99	0.99	0.99	0.99
$\gamma_{2,1}$	0.31	0.31	0.31	0.31	0.31
$\gamma_{1,2}$	0.01	0.01	0.01	0.01	0.01
$\gamma_{2,2}$	0.69	0.69	0.69	0.69	0.69
δ_1	0.96	0.96	0.96	0.96	0.96
δ_2	0.04	0.04	0.04	0.04	0.04

Table 9 Estimates of 2 state Poisson HMM with and without using TMB, estimated on the lamb dataset. Naming convention is found in Table 2

provides an algorithm for least-squares curve fitting, and is therefore included in the comparison. Exact gradients and hessians are provided by `TMB` and fed to each algorithm. The speed comparisons are in the following Figure 6.

The following tables summarize the estimates and their confidence intervals, for the lamb dataset and for the simulated dataset. Instead of showing the standard error, we display the lower and upper

m	TMB1	TMB2	TMB3	TMB4
1	2297	2283	2306	2268
2	2179	2178	2214	2165

Table 10 Speed percentage increase of using TMB for m state Poisson HMM negative log-likelihood calculation, estimated on the lamb dataset. Each procedure was repeated and timed. Their mean times are compared in this table. When m=1, the mean estimation time with TMB by providing the exact gradient but not the exact hessian (TMB3) was 2306% lower than the mean estimation time with direct maximization without TMB (DM). Naming convention is found in Table 2

m	TMB1	TMB2	TMB3	TMB4
1	2348	2423	2385	2350
2	1904	1914	1822	1872
3	1238	1229	1240	1177

Table 11 Speed percentage increase of using TMB for m state Poisson HMM negative log-likelihood calculation, estimated on the simulated dataset. Each procedure was repeated and timed. Their mean times are compared in this table. When m=1, the mean estimation time with TMB by providing the exact gradient but not the exact hessian (TMB3) was 2385% lower than the mean estimation time with direct maximization without TMB (DM). Naming convention is found in Table 2

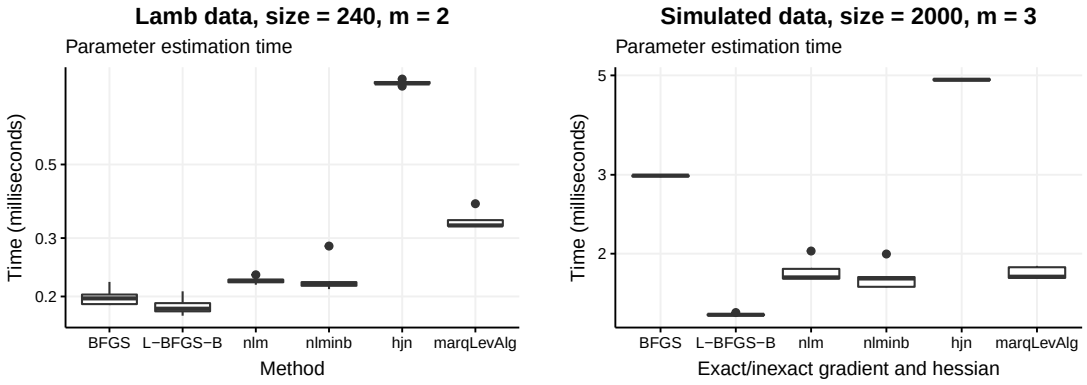

Figure 6 Poisson HMM parameter estimation time per optimization method

bounds of the confidence intervals. The reason is that sometimes, only one bound of the interval is known when using the profile likelihood method and hence doesn't obtain a standard error.

m	Parameter	Parameter.estimate	Profile.L	Profile.U	Bootstrap.L	Bootstrap.U	TMB.L	TMB.U
1	λ_1	0.36			0.29	0.44	0.28	0.43
2	λ_1	0.26	0.15	0.33	0.00	0.33	0.18	0.34
2	λ_2	3.11	1.27	4.95	0.39	5.21	1.11	5.12
2	$\gamma_{1,1}$	0.99	0.93	1.00	0.56	1.00	0.97	1.01
2	$\gamma_{2,1}$	0.31	0.04	0.68	0.04	1.00	-0.05	0.67
2	$\gamma_{1,2}$	0.01	0.00	0.07	0.00	0.44	-0.01	0.03
2	$\gamma_{2,2}$	0.69	0.32	0.96	0.00	0.96	0.33	1.05
2	δ_1	0.96			0.96	0.96	0.90	1.03
2	δ_2	0.04			0.04	0.04	-0.03	0.10

Table 12 Estimates and standard errors on the lamb dataset

Error?

m	Parameter	Parameter.value	Parameter.estimate	Profile.L	Profile.U	Bootstrap.L	Bootstrap.U	TMB.L	TMB.U
1	λ_1	1.00	0.97			0.93	1.01	0.93	1.02
2	λ_1	1.00	0.99	0.93	1.05	0.93	1.05	0.93	1.05
2	λ_2	20.00	20.12	19.83	20.40	19.83	20.43	19.83	20.40
2	$\gamma_{1,1}$	0.80	0.79	0.77	0.82	0.77	0.82	0.77	0.82
2	$\gamma_{2,1}$	0.20	0.22	0.20	0.25	0.20	0.25	0.20	0.25
2	$\gamma_{1,2}$	0.20	0.21	0.18	0.23	0.18	0.23	0.18	0.23
2	$\gamma_{2,2}$	0.80	0.78	0.75	0.80	0.75	0.80	0.75	0.80
2	δ_1	0.50	0.52			0.52	0.52	0.48	0.56
2	δ_2	0.50	0.48			0.48	0.48	0.44	0.52
3	λ_1	1.00	0.96	0.88	1.04	0.88	1.03	0.88	1.04
3	λ_2	10.50	10.30	10.03	10.58	10.04	10.58	10.03	10.58
3	λ_3	20.00	20.16	19.77	20.56	19.75	20.55	19.77	20.56
3	$\gamma_{1,1}$	0.80	0.81	0.77	0.85	0.78	0.84	0.78	0.84
3	$\gamma_{2,1}$	0.10	0.09	0.07	0.11	0.07	0.11	0.06	0.11
3	$\gamma_{3,1}$	0.10	0.10	0.08	0.12	0.07	0.12	0.07	0.12
3	$\gamma_{1,2}$	0.10	0.09	0.07	0.11	0.06	0.11	0.06	0.11
3	$\gamma_{2,2}$	0.80	0.83	0.79	0.87	0.80	0.86	0.80	0.86
3	$\gamma_{3,2}$	0.10	0.10	0.08	0.13	0.08	0.13	0.08	0.13
3	$\gamma_{1,3}$	0.10	0.10	0.08	0.13	0.08	0.13	0.08	0.13
3	$\gamma_{2,3}$	0.10	0.09	0.07	0.11	0.06	0.11	0.06	0.11
3	$\gamma_{3,3}$	0.80	0.80	0.75	0.84	0.77	0.83	0.77	0.83
3	δ_1	0.33	0.33			0.33	0.33	0.27	0.38
3	δ_2	0.33	0.35			0.35	0.35	0.30	0.41
3	δ_3	0.33	0.32			0.32	0.32	0.27	0.37

Table 13 Estimates and standard errors on the simulated dataset

All the code used to produce those results is available in the supporting information.

8 Discussion

In this paper, an alternative method of computing Poisson HMM parameters and their standard errors is examined. Using the language R, the usual method can take a long time, particularly if the standard errors are computed by bootstrapping. The approach with `TMB` was timed on 3 datasets with different amounts of data, and compared with the times of estimations using R without making use of `TMB`. For computational reasons, this paper doesn't estimate HMMs on millions or billions of data points, but the performance gains are expected to be large even in these cases. As can be seen in Table 13 and Table 12, the standard errors obtained through this method are very similar to the standard errors obtained through profiling the likelihood and bootstrapping while being less computationally intensive. The method used in this paper can easily be extended to other distributions.

It is recommended to use both `TMB`'s gradient and hessian since it ensures a more reliable result, although it doesn't make any difference in our computations. However, including the hessian can make the optimization slower because of its size, as demonstrated by the difference in time between TMB3 and TMB4 when dealing with a multiple state Poisson HMM on a large amount of data (hospital dataset).

Although we have not tried, it should be possible to extend this approach to HMMs with random effects for panel data for example. Introduced by Altman (2007), Mixed HMM (MHMM) is a class of HMM that combines fixed and random effects, by using the framework of Generalized Linear Mixed Models. `TMB` should be useable for MHMMs since the likelihood can be optimized through direct maximization as shown by Altman (2007), or through the EM algorithm as shown by Maruotti (2011). Both authors list possible approaches to estimate MHMMs. Altman (2007, p . 7) reports a numerical integration taking from 1 second with 1 random effect to several days when 4 random effects are included, and a Monte Carlo method requiring approximately 3 days to estimate a MHMM with 3 random effects. In addition, both assume the observations (conditional on the random effects and the hidden states), to be distributed in the exponential family. By default, `TMB` integrates out the random effects and then uses a Laplace approximation on the negative joint log-likelihood. However, if needed (for example because the minimizer with respect to the random effect is not unique) one can set this approximation explicitly in C++.

(i) To our surprise, TMB3 was the fastest in most cases although we don't understand why. It operates only with an exact gradient provided by `TMB` but without the exact hessian. This happened when we used hidden states ($m > 1$), and also when using other optimizers than `nlminb` such as `nlm`. Although adding the exact hessian on top of the gradient to the optimizer seems to slow down the computation, it might help the optimizer converge in some cases, and could be worth investigating.

(ii) When providing only `TMB`'s hessian (TMB2), optimizers sometimes fail to converge.

(iii) We have timed this approach on different volumes of data, but we haven't tried very large datasets of sizes in the millions or billions. However, we expect the increase in speed to be smaller since the time necessary becomes longer independently of the approach chosen.

(iv) Mention the prospect of using `TMB` for panel data with random effects, Laplace approximation.

Acknowledgements We gratefully thank Dr. Bertrand GALICHON and Dr. Anthony CHAUVIN for their patience and their efforts to provide the hospital dataset along with the necessary authorizations.

Conflict of Interest

The authors have declared no conflict of interest.

Appendix

A.1. R Code

(i) Code to setup global parameters and declare functions used

(ii) Packages

(iii) Functions

(iv) Code to run estimations and comparisons using the lamb dataset

(v) Code to run estimations and comparisons using a simulated dataset

(vi) Code to run estimations and comparisons using the hospital dataset

A.2. C++ Code

(i) Poisson HMM negative log-likelihood calculation This is the file poi_hmm.cpp which contains the negative likelihood function.

It should be noted that in the C++ file, testing if the value is missing (i.e. testing for NaN (Not A Number) values) requires a little trick. The reason is that the standard test function `std::isnan()` doesn't currently work on a single data value inside `TMB`. Cuurently in C++, comparisons involving NaN values are always false, except when testing inequality between 2 NaN values. In other words, for a float f, the expression f != f is true if and only if f is a NaN value. Similarly, f == f returns false if and only if f is a NaN value.

```
#include <TMB.hpp>
#include "../functions/utils.cpp"

// Likelihood for a poisson hidden markov model.
template<class Type>
Type objective_function<Type>::operator() ()
{

  // Data
  DATA_VECTOR(x);           // timeseries vector
  DATA_INTEGER(m);          // Number of states m

  // Parameters
  PARAMETER_VECTOR(tlambda);      // conditional log_sd's
  PARAMETER_VECTOR(tgamma);       // m(m-1) working parameters of TPM

  // Uncomment only using a non stationary distribution
  //PARAMETER_VECTOR(tdelta);     // transformed stationary distribution,

  // Transform working parameters to natural parameters:
  vector<Type> lambda = tlambda.exp();
  matrix<Type> gamma = Gamma_w2n(m, tgamma);

  // Construct stationary distribution
```



```
vector<Type> delta = Stat_dist(m, gamma);
// If using a non stationary distribution, use this instead
//vector<Type> delta = Delta_w2n(m, tdelta);

// Get number of timesteps (n)
int n = x.size();

// Evaluate conditional distribution: Put conditional
// probabilities of observed x in n times m matrix
// (one column for each state, one row for each datapoint):
matrix<Type> emission_probs(n, m);
matrix<Type> row1vec(1, m);
row1vec.setOnes();
for (int i = 0; i < n; i++) {
  if (x[i] != x[i]) { // f != f returns true if and only if f is NaN.
    // Replace missing values (NA in R, NaN in C++) with 1
    emission_probs.row(i) = row1vec;
  }
  else {
    emission_probs.row(i) = dpois(x[i], lambda, false);
  }
}

// Corresponds to the book page 333
matrix<Type> foo, P;
Type mllk, sumfoo, lscale;

if (m == 1) {
  mllk = -emission_probs.col(0).array().log().sum();

  // Use adreport on variables we are interested in:
  ADREPORT(lambda);
  ADREPORT(gamma);
  ADREPORT(delta);

  // Things we need for local decoding
  REPORT(lambda);
  REPORT(gamma);
  REPORT(delta);

  return mllk;
}
foo = (delta * vector<Type>(emission_probs.row(0))).matrix();
sumfoo = foo.sum();
lscale = log(sumfoo);
foo.transposeInPlace();
```



```
  foo /= sumfoo;
  for (int i = 2; i <= n; i++) {
    P = emission_probs.row(i - 1);
    foo = ((foo * gamma).array() * P.array()).matrix();
    sumfoo = foo.sum();
    lscale += log(sumfoo);
    foo /= sumfoo;
  }
  mllk = -lscale;

  // Use adreport on variables for which we want standard errors
  ADREPORT(lambda);
  ADREPORT(gamma);
  ADREPORT(delta);

  // Variables we need for local decoding and conveniency
  REPORT(lambda);
  REPORT(gamma);
  REPORT(delta);
  REPORT(n);
  REPORT(emission_probs);
  REPORT(mllk);

  return mllk;
}
```

(ii) This optional function is in the file utils.cpp. It is only necessary if a stationary distribution is not assumed, and $\boldsymbol{\delta}$ is used as a parameter instead of being derived from the transistion probability matrix $\boldsymbol{\Gamma}$. It codes the function to convert the working parameter `tdelta` into its natural format.

```
// Function transforming working parameters in initial distribution
// to natural parameters
template<class Type>
vector<Type> Delta_w2n(int m, vector<Type> tdelta) {

  vector<Type> delta(m);
  vector<Type> foo(m);

  if (m == 1)
    return Type(1);

  // set first element to one.
  // Fill in the last m - 1 elements with working parameters
  // and take exponential
  foo << Type(1), tdelta.exp();

  // normalize
```



```
  delta = foo / foo.sum();

  return delta;
}
```

(iii) This function is in the file utils.cpp. It transforms the transition probability matrix $\mathbf{\Gamma}$ from its working format to its natural format.

```
// Function transforming the working parameters in TPM to
// natural parameters (w2n)
template<class Type>
matrix<Type> Gamma_w2n(int m, vector<Type> tgamma) {

  // Construct m x m identity matrix
  matrix<Type> gamma(m, m);
  gamma.setIdentity();

  if (m == 1)
    return gamma;

  // Fill offdiagonal elements with working parameters column-wise:
  int idx = 0;
  for (int i = 0; i < m; i++){
    for (int j = 0; j < m; j++){
      if (j != i){
        // Fill gamma according to mapping and take exponential
        gamma(j, i) = tgamma.exp()(idx);
        idx++;
      }
    }
  }
  // Normalize each row:
  vector<Type> cs = gamma.rowwise().sum();
  for (int i = 0; i < m; i++) gamma.row(i) /= cs[i];

  return gamma;
}
```

(iv) This function is in the file utils.cpp. It derives the stationary distribution from the transition probability matrix $\mathbf{\Gamma}$.

```
// Function computing the stationary distribution of a Markov chain
template<class Type>
vector<Type> Stat_dist(int m, matrix<Type> gamma) {

  // Construct stationary distribution
  matrix<Type> I(m, m);
  matrix<Type> U(m, m);
```



```
  matrix<Type> row1vec(1, m);
  U = U.setOnes();
  I = I.setIdentity();
  row1vec.setOnes();
  matrix<Type> A =  I - gamma + U;
  matrix<Type> Ainv = A.inverse();
  matrix<Type> deltamat = row1vec * Ainv;
  vector<Type> delta = deltamat.row(0);

  return delta;
}
```

(v) Functions used in C++ Poisson HMM code

(vi) Linear model negative log-likelihood calculation

(vii) Functions used in C++ linear model code

References

Rachel MacKay Altman. Mixed Hidden Markov Models. *Journal of the American Statistical Association*, 102(477): 201–210, March 2007. ISSN 0162-1459. doi: 10.1198/016214506000001086.

Leonard E. Baum and Ted Petrie. Statistical Inference for Probabilistic Functions of Finite State Markov Chains. *The Annals of Mathematical Statistics*, 37(6):1554–1563, December 1966. ISSN 0003-4851. doi: 10.1214/aoms/1177699147.

Leonard E. Baum, Ted Petrie, George Soules, and Norman Weiss. A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains. *The Annals of Mathematical Statistics*, 41(1): 164–171, 1970. ISSN 00034851.

Jan Bulla and Andreas Berzel. Computational issues in parameter estimation for stationary hidden Markov models. *Computational Statistics*, 23(1):1–18, January 2008. ISSN 1613-9658. doi: 10.1007/s00180-007-0063-y.

Olivier Cappé, Eric Moulines, and Tobias Ryden. *Inference in Hidden Markov Models*. Springer Science & Business Media, April 2006. ISBN 978-0-387-28982-3.

Bradley Efron and Robert J Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall, New York, N.Y.; London, 1993. ISBN 978-0-412-04231-7.

William Feller. *An Introduction to Probability Theory and Its Applications*. Wiley, 1968. ISBN 978-0-471-25708-0.

Donald R. Fredkin and John A. Rice. Bayesian Restoration of Single-Channel Patch Clamp Recordings. *Biometrics*, 48(2):427–448, 1992. ISSN 0006341X, 15410420. doi: 10.2307/2532301.

Sylvia Frühwirth-Schnatter. *Finite Mixture and Markov Switching Models*. Springer Science & Business Media, November 2006. ISBN 978-0-387-35768-3.

Mark Gales and Steve Young. The Application of Hidden Markov Models in Speech Recognition. *Foundations and Trends® in Signal Processing*, 1(3):195–304, February 2008. ISSN 1932-8346, 1932-8354. doi: 10.1561/2000000004.

Geoffrey Grimmett, Geoffrey R. Grimmett, Professor of Mathematical Statistics Geoffrey Grimmett, David Stirzaker, and Mathematical Institute David R. Stirzaker. *Probability and Random Processes*. OUP Oxford, May 2001. ISBN 978-0-19-857222-0.

Wolfgang Härdle, Joel Horowitz, and Jens-Peter Kreiss. Bootstrap Methods for Time Series. *International Statistical Review*, 71(2):435–459, 2003. ISSN 1751-5823. doi: 10.1111/j.1751-5823.2003.tb00485.x.

Robert E. Kass and Duane Steffey. Approximate Bayesian Inference in Conditionally Independent Hierarchical Models (Parametric Empirical Bayes Models). *Journal of the American Statistical Association*, 84(407):717–726, September 1989. ISSN 0162-1459. doi: 10.1080/01621459.1989.10478825.

Kasper Kristensen, Anders Nielsen, Casper W. Berg, Hans Skaug, and Brad Bell. TMB: Automatic differentiation and Laplace approximation. *arXiv preprint arXiv:1509.00660*, 2015.

Brian G. Leroux and Martin L. Puterman. Maximum-Penalized-Likelihood Estimation for Independent and Markov-Dependent Mixture Models. *Biometrics*, 48(2):545–558, 1992. ISSN 0006-341X. doi: 10.2307/2532308.

Theodore C Lystig and James P Hughes. Exact Computation of the Observed Information Matrix for Hidden Markov Models. *Journal of Computational and Graphical Statistics*, 11(3):678–689, September 2002. ISSN 1061-8600. doi: 10.1198/106186002402.

Antonello Maruotti. Mixed hidden markov models for longitudinal data: An overview. *International Statistical Review*, 79(3):427–454, 2011. ISSN 1751-5823. doi: 10.1111/j.1751-5823.2011.00160.x.

Geoffrey J. McLachlan and David Peel. *Finite Mixture Models*. John Wiley & Sons, March 2004. ISBN 978-0-471-65406-3.

William Q. Meeker and Luis A. Escobar. Teaching about Approximate Confidence Regions Based on Maximum Likelihood Estimation. *The American Statistician*, 49(1):48–53, February 1995. ISSN 0003-1305. doi: 10.1080/00031305.1995.10476112.

Eric E. Schadt, Janet S. Sinsheimer, and Kenneth Lange. Computational Advances in Maximum Likelihood Methods for Molecular Phylogeny. *Genome Research*, 8(3):222–233, January 1998. ISSN 1088-9051, 1549-5469. doi: 10.1101/gr.8.3.222.

Rolf Turner. Direct maximization of the likelihood of a hidden Markov model. *Computational Statistics & Data Analysis*, 52(9):4147–4160, May 2008. ISSN 0167-9473. doi: 10.1016/j.csda.2008.01.029.

D. J. Venzon and S. H. Moolgavkar. A Method for Computing Profile-Likelihood-Based Confidence Intervals. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 37(1):87–94, 1988. ISSN 1467-9876. doi: 10.230

7/2347496.

Ingmar Visser, Maartje E. J. Raijmakers, and Peter C. M. Molenaar. Confidence intervals for hidden Markov model parameters. *British Journal of Mathematical and Statistical Psychology*, 53(2):317–327, 2000. ISSN 2044-8317. doi: 10.1348/000711000159240.

Abraham Wald. Tests of Statistical Hypotheses Concerning Several Parameters When the Number of Observations is Large. *Transactions of the American Mathematical Society*, 54(3):426–482, 1943. ISSN 0002-9947. doi: 10.2307/1990256.

W. Zucchini, I.L. MacDonald, and R. Langrock. *Hidden Markov Models for Time Series: An Introduction Using r, Second Edition*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. CRC Press, 2016. ISBN 978-1-4822-5384-9.

