(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]](#footnote-1)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

Abstract here

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

*(Up to five keywords are allowed and should be given in alphabetical order. Please capitalize the key words)*

Supporting Information for this article is available from the author or on the WWW under http://dx.doi.org/10.1022/bimj.XXXXXXX (please delete if not applicable)

# Introduction

Short paragraph about HMMs and problems with speed and uncertainty evaluation.

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 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, the constraints can be imposed by suitable transformations of the parameters (see e.g. Zucchini et al., 2016). The covariance matrix of the untransformed (original) parameters can then be obtained by the delta method, a feature implemented in TMB.

In this paper, we will show how to optimize a Poisson HMM using TMB in the language R. Afterwards, we will use an example to show how to make a nested model, efficiently compute confidence intervals, retrieve interesting probabilities, we will apply a Poisson HMM on a hospital dataset. Eventually, we will see that TMB can accelerate traditional optimizers in R by up to a few hundred times on a large dataset, simplifies evaluating confidence intervals, and gives similar parameter uncertainty to bootstrap and profile methods.

An aspect to remember is that we decided to use a direct maximization approach instead of the ExpectationMaximization (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. Selling points to include:

1. Two aspects of the paper: 1) Speedup of estimation 2) Esier and faster evaluation of parameter uncertainty
2. Speed important when *T* and *m* large?
3. Hessian based uncertainty earlier in the traditional sense not feasable (Visser et al., 2000), and finitedifferences must be employed since the computation of the Hessian not feasible.
4. Bootstrap methods requires speed
5. Profile methods requires speed

# Principles of using TMB for Maximum Likelihood Estimation (MLE)

2.1 Setup

In order to use TMB in R, it is necessary to install the R-package TMB and have the software Rtools installed. The latter is used for compiling the C++ code.

In this paper, we will use R version 3.6.0 and the code is run on Windows 10 Enterprise version 1809, with an Intel(R) Core(TM) i7-8700 processor.

The C++ code can be difficult to debug as it operates using a specific template, see Section 2.2 below for an example. TMB provides a debugging feature available with RStudiowhich can be useful to retrieve diagnostic error messages. Enabling it is optional. The command TMB:::setupRStudio() enables this feature and requires manual input to confirm.

2.2 Linear regression example

We start by illustrating how TMB can be used to fit a simple linear model. A more comprehensive tutorial on how to use TMB can be found at https://kaskr.github.io/adcomp/\_book/Tutorial.html.

Let *x* and *y* be datasets of size n, then the negative log-likehood corresponding to a simple linear model is given by

*n*

|  |  |
| --- | --- |
| −*l*(*a,b,σ*2) = −Xlog(*φ*(*yi*;*a* + *bxi,σ*2)))*,*  *i*=1 | (1) |

where *φ*(·;*µ,σ*2) is the density function of the univariate normal distribution with mean *µ* and variance *σ*2. TMB requires the (negative) log-likelihood function to be written in C++ under a specific template, and then loaded into R. Minimization, and other post-processing procedures are all done in R. The following C++ code called linreg.cpp produces the negative log-likelihood of the linear model:

1. .

We show here an example of a C++ objective function’s code of a linear regression in a file linreg.cpp.

|  |
| --- |
|  |
| #include *<*TMB. hpp*> //import the TMB template* |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| template*<*class Type*>* |
|  |
|  |
|  |
|  |
| Type objective function *<*Type *>*:: operator ( ) ( ) |
| { |
|  |
|  |
|  |
| // Data |
| DATA VECTOR( y ) ; // Data vector y transmitted from R |
| DATA VECTOR( x ) ; // Data vector x transmitted from R |
|  |
| // Parameters  PARAMETER( a ) ; // Parameter a transmitted from R |
| PARAMETER( b ) ; // Parameter b transmitted from R |
| PARAMETER( logSigma ) ; // Sigma on log-scale transmitted from R |
| // Transform logSigma back to natural scale  Type sigma = exp(logSigma); |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| // Declare negative log-likelihood |
| Type n l l = − sum( dnorm (y , |
| a + b \* x , |
| sigma , |
| true ) ) ; |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| */\* This l e t s the user r e t r i e v e any v a r i a b l e s along with t h e i r standard e r r o r s* |
|  |
|  |
| ADREPORT( a ) ; |
| ADREPORT( b ) ; |
| ADREPORT( sigma ) ; |
|  |
| return n l l ; |
| }  Note that in the above code we declare that the negative log-likelihood is a function of three parameters: *a,b* and logSigma. *Since the parameter* *σ*has to be non-negative the negative log-likelihood is parameterized in terms of log(*σ*) which can be freely estimated. By subsequently cthe s,can (Kass and Steffey (1989))to corresponding This feature is particularly useful when the likelihood has been reparameterized as above.  Once the C++ file with the negative log-likelihood is established it can be compiled and minimized in R as shown in the .R file below: |

|  |
| --- |
| *# Setting up TMB*  **library**(TMB) **compile**("code/linreg.cpp")  **dyn.load**(**dynlib**("code/linreg"))  *# Generate the data* **set.seed**(123) dat <- **list**(y = **rnorm**(20) + 1:20, x = 1:20) parameters <- **list**(a = 0, b = 0, logSigma = 0) *# Tell TMB to create the likelihood function* obj\_linreg <- **MakeADFun**(data, parameters, DLL = "linreg", silent = TRUE  *# Optimization of the objective function with TMB* mod\_linreg <- fit fit<- **nlminb**(obj\_linreg$par, obj\_linreg$fn) mod\_linreg  print(fit$par) |

The function sdreport will by default give the estimates and standard errors of the variables that the negative log-likelihood is parameterized in terms of:**sdreport**(obj\_linreg)

## sdreport(.) result

## Estimate Std. Error

## a 0.31009240 0.43829083

## b 0.98395535 0.03658781

## logSigma -0.05814659 0.15811381

## Maximum gradient component: 6.931683e-05

To get the estimates and standard errors of the variables called with ADREPORT() in the C++ file we can run a summary call with the select = “report” option:

**summary**(**sdreport**(obj\_linreg), select = "report")

## Estimate Std. Error

## a 0.3100924 0.43829083

## b 0.9839553 0.03658781

## sigma 0.9435116 0.14918222

And we can see the same estimates from a regular linear regression.

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

## ## Call:

## lm(formula = y ˜ x, data = data)

## ## Residuals:

## Min 1Q Median 3Q Max

## -1.98791 -0.60280 -0.03768 0.51942 1.73353

## ## Coefficients:

## Estimate Std. Error t value Pr(>|t|)

## (Intercept) 0.31009 0.46200 0.671 0.511

## x 0.98396 0.03857 25.513 1.39e-15 \*\*\*

## ---

## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

##

## Residual standard error: 0.9945 on 18 degrees of freedom

## Multiple R-squared: 0.9731,Adjusted R-squared: 0.9716

## F-statistic: 650.9 on 1 and 18 DF, p-value: 1.389e-15











# Estimation TMB/HMM

3.1 Notation

A Hidden Markov Model (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 will 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 {*Xn* : *n* = 1*,...,N*} is assumed to follow a mixture of *m* Poisson distributions with parameters {*λi* : *i* = 1*,...,m*}. Let {*Ct* : *t* = 1*,...,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.

*p*1(*x*) 0 

 *p*2(*x*) 

|  |  |  |  |
| --- | --- | --- | --- |
|  ..  | | | |
|  |  |  |    0 *pm*(*x*) |
| *p*1(*x*1)   1(*x*2) *p* We will call  ...      *p*1(*xn*) | *p*2(*x*1) *p*2(*x*2)  ...  *p*2(*xn*) | *...*  *...*  ...  *...* | *pm*(*x*1)  *pm*(*x*2   ...  the emission probability matrix. If *x* is a missing data, *pm*(*xn*) |

Let *pi*(*x*) = *P*(*Xt* = *x*|*Ct* = *i*)*,*∀*i* = 1*,...,m* and let **P**(*x*) =  . .

we will define *pi*(*x*) = 1 and therefore the emission matrix of the dataset will contain a line filled with ones.

Furthermore, we let **Γ** = {*γij*} denote the transition matrix of the Markov chain, a *δ* be its stationary distributionFinally, Let *X*(*t*) = {*X*1*,...,Xt*} and *x*(*t*) = {*x*1*,...,xt*} denote the history up to time *t*.

In Section 4, we will consider 3 different approaches of evaluating the confidence intervals for the uncertainty of the parameters: Hessian based, profile likelihood, and bootstrap.

3.2 Hmm Likelihood

Parameters subject to estimation: *θ* = (*γ*11*,...,γ*1*m,...,γm*1*,...,γmm,λ*1*,...,λm*)

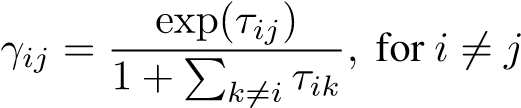
Likelihood formulation, forward algorithm, parameter transformations to avoid constraints

1. It can be shown (Zucchini reference) that the l likelihood of an HMM is given by

*LN* = P(X^(t)=x^(t))= *δ***P**(*x*1)**ΓP**(*x*2)**ΓP**(*x*3)*...***ΓP**(*xN*)**1**0

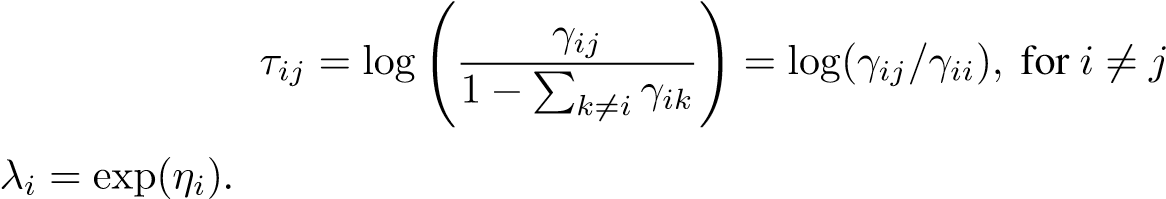
1. It is usual for HMMs to estimate the parameters *θ* by optimizing a reparametrized version of the log-likelihood log*L*(*ψ*), where *ψ* = *g*−1(*θ*) represent a set of unconstrained parameters. That transformation will be detailed below. As a result of the invariance principle, the maximum likelihood estimate of *θ* is given by *θ*ˆ = *g*(*ψ*ˆ) where *ψ*ˆ is the minimizer of −log*L*(*ψ*). To avoid constraints on the parameters, we transform the natural parameters {**Γ***,λ*} into working parameters {**T***,η*}.

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 **Γ** is done this way:



Each row must add to 1, so we can get the diagonal elements of **Γ** easily. *ηi* = log(*λi*).

The reverse transformation is

And

The relevant R function to do so is given by Zucchini et al. (2016, p. 52):

|  |
| --- |
| *# Transform Poisson natural parameters to working parameters* pois.HMM.pn2pw <- **function**(m, lambda, gamma, delta = **NULL**, stationary = TRUE) {  tlambda <- **log**(lambda) **if** (m == 1) **return**(tlambda) foo <- **log**(gamma / **diag**(gamma)) tgamma <- **as.vector**(foo[!**diag**(m)]) **if** (stationary) {  tdelta <- **NULL**  } **else** { tdelta <- **log**(delta[- 1] / delta[1])  }  Return(list(tlambda = tlambda, tgamma = tgamma, tdelta = tdelta))  } |

Those working parameters are fed into TMB, which will then turn them back to natural parameters before calculating the log-likelihood. That transformation from working parameters to natural parameters is done in the C++ likelihood function. We will now show a way to do that transformation:

The C++ function turning the working transition probability matrix into a natural parameter is the function Gammaw2n available in the file utils.cpp.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| */ / Function t r a n s f o r m i n g the working parameters in TPM to* | | | | |
| */ / n a t u r a l parameters ( w2n )* | | | | |
| template*<*class Type*>* | | | | |
| matrix*<*Type*>* Gamma w2n( int m, vector *<*Type*>* tgamma ) { | | | | |
|  | | | | |
| */ / Construct m x m i d e n t i t y matrix* | | | | |
| matrix*<*Type*>* gamma(m, m) ; | | | | |
| gamma . s e t I d e n t i t y ( ) ; | | | | |
|  | | | | |
| i f (m == 1) | | | | |
| return gamma; | | | | |
|  | | | | |
| */ / F i l l o f f d i a g o n a l elements with working parameters column*−*wise :* | | | | |
| int idx = 0; | | | | |
| for ( int i = 0; i *<* m; i ++){ | | | | |
| for ( int j = 0; j *<* m; j ++){ | | | | |
| i f ( j != i ){ | | | | |
| */ / F i l l gamma according to mapping and take e x p o n e n t i a l* | | | | |
| 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; |  |  |  |  |
| } |  |  |  |  |

The C++ function turning the working stationary distribution vector into a natural parameter is the function Deltaw2n available in the file utils.cpp.

|  |
| --- |
| */ / Function t r a n s f o r m i n g working parameters in i n i t i a l d i s t r i b u t i o n* |
| */ / to n a t u r a l parameters* |
| template*<*class Type*>* |
| vector *<*Type*>* Delta w2n ( int m, vector *<*Type*>* t d e l t a ) { |
|  |
| vector *<*Type*>* d e l t a (m) ; |
| vector *<*Type*>* foo (m) ; |
|  |
| i f (m == 1) |
| return Type ( 1 ) ; |
|  |
| */ / s e t f i r s t element to one .* |
| */ / F i l l in the l a s t m* − *1 elements with working parameters* |
| */ / and take e x p o n e n t i a l* |
| foo *<<* Type (1) , t d e l t a . exp ( ) ; |
|  |
| */ / normalize* |
| d e l t a = foo / foo . sum ( ) ; |
|  |
| return d e l t a ; |
| } |

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

|  |
| --- |
| vector *<*Type*>* lambda = log lambda . exp ( ) ; |

(iii) The “forward algorithm” allows for a recursive computation of the likelihood.

To state the forward algorithm let us define the vector *αt* for *t* = 1*,*2*,...,N* so that

*αt* = *δ***P**(*x*1)**ΓP**(*x*2)**ΓP**(*x*3)*...***ΓP**(*xt*)

*t*

= *δ***P**(*x*1) Y**ΓP**(*xs*)

*s*=2

## = (*αt*(1)*,...,αt*(*m*))

where *δ* 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 *αt* values are calculated

*α*0 = *δ*

*αt* = *αt*−1**ΓP**(*xt*) for *t* = 1*,*2*,...,N*

*LT* = *αT***1**0

(iv) 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 *βt* for *t* = 1*,*2*,...,N* so that *β*0*t* = **ΓP**(*xt*+1)**ΓP**(*xt*+2)*...***ΓP**(*xT*)*...***1**0

*N* !

= Y **ΓP**(*xs*) **1**0

*s*=*t*+1

## = (*βt*(1)*,...,βt*(*m*))

It is called the backward algorithm because of the way the *βt* values are calculated

*βT* = **1**0

*βt* = **ΓP**(*xt*+1)*βt*+1 for *t* = *N* − 1*,N* − 2*,...,*1

*LT* = *δβ*1

(v) Computing those probabilities directly can lead to underflow errors, it is therefore better to maximize the log-likelihood instead. We chose to minimize the negative log-likelihood for convenience purposes. In addition, in our case **Γ** and *λ* are subject to some constraints:

* + 1. The means *λi* of the state-dependent distributions must be non-negative, for *i* = 1*,...,m*
    2. The parameters *γi,j* of the transition probability matrix **Γ** must be non-negative, and the rows of

**Γ** must add to one

3.3 Using TMB

3.3.1 Computing the stationary distribution

Here, we show how to compute the stationary distribution of a Markov chain with *m* states and a transition probability matrix gamma. To do that, we find the solution *δ* of the system:

## *δ*(**I***m* − **Γ** + **U**) = **1**

where **I***m* is the *m* x *m* identity matrix, **U** is a *m* x *m* matrix of ones, and **1** is a row vector of ones. An implementation of this in R is

|  |
| --- |
| *# Compute the stationary distribution of a Markov chain*  *# with transition probability gamma* stat.dist <- **function**(gamma) {  *# m = dim(gamma)[1]*  *# A = diag(m) - gamma*  *# A[, m] = rep(1, m)*  *# b = c(rep(0, m - 1), 1) # return(solve(a = t(A), b = b))* m <- **dim**(gamma)[1]  **return**(**solve**(**t**(**diag**(m) - gamma + 1), **rep**(1, times = m)))  } |

More details are available in Zucchini et al. (2016).

In order to use it in TMB, we will now show a way to implement it in a separate file in C++ that we named utils.cpp.

*computing*

*/ /*

*Function*

*a*

*the*

*s t a t i o n a r y*

*d i s t r i b u t i o n*

*of*

*Markov*

*chain*

template

*<*

class

Type

*>*

vector

*<*

Type

*>*

t a t

S

d i s t (

int

matrix

m,

*<*

Type

*>*

gamma)

{

*d i s t r i b u t i o n*

*Construct*

*/ /*

*s t a t i o n a r y*

matrix

*<*

Type

*>*

I (m, m) ;

matrix

*<*

Type

*>*

U(m, m) ;

matrix

*<*

Type

*>*

row1vec (1 ,

m) ;

U = U. setOnes

( )

;

I

=

I . s e t I d e

n t i t y ( ) ;

nes ( ) ;

row1vec . setO

matrix

*<*

Type

*>*

A =

I

−

gamma + U;

matrix

*<*

Type

*>*

Ainv = A. inverse ( ) ;

matrix

*<*

Type

*>*

deltamat

=

row1vec

\*

Ainv ;

vector

*<*

Type

*>*

deltamat . row ( 0 ) ;

d e l t a

=

return

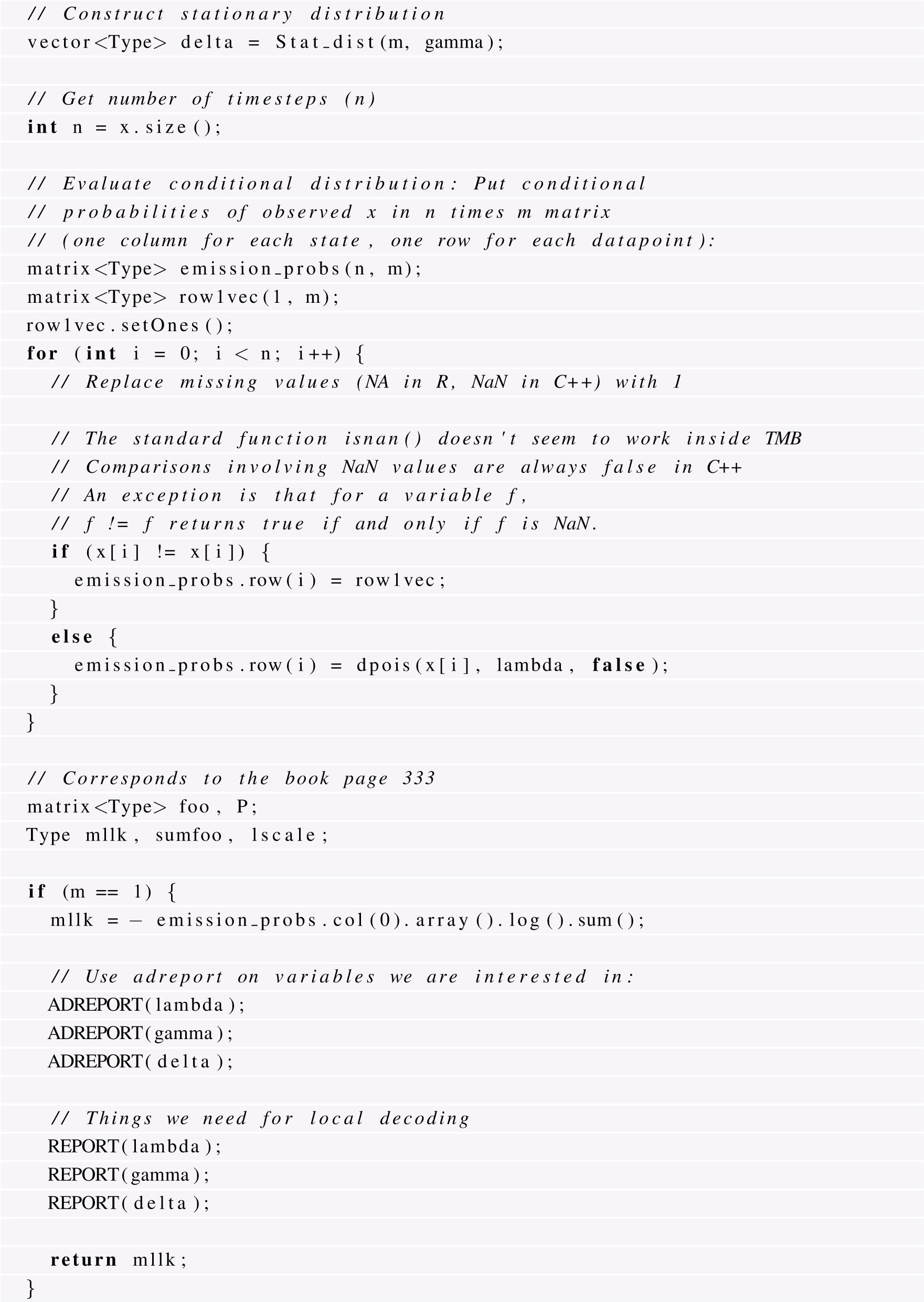
d e l t a ;

}

3.3.2 Likelihood function

Now we will show how to calculate the negative log-likelihood using TMB. C++ code in file poi hmm.cpp:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| #include *<*TMB. hpp*>* | | |  |  |
| #include ” . . / functions / u t i l s . cpp” | | |  |  |
|  | | |  |  |
| */ / L i k e l i h o o d f o r a poisson hidden markov model .* | | |  |  |
| template*<*class Type*>* | | |  |  |
| Type objective function *<*Type *>*:: operator ( ) ( ) | | |  |  |
| { |  | |  |  |
|  |  | |  |  |
| */ / Data* |  | |  |  |
| DATA VECTOR( x ) ; | */ / t i m e s e r i e s v e c t o r* | |  |  |
| DATA INTEGER(m) ; | */ / Number of s t a t e s m* | |  |  |
|  | |  |  |  |
| */ / Parameters* | |  |  |  |
| PARAMETER VECTOR( log lambda ) ; | | */ / c o n d i t i o n a l* | *l o g s d ' s* |  |
| PARAMETER VECTOR( tgamma ) ; | | */ / m(m*−*1) working parameters* | | *of TPM* |
|  | | | |  |
| */ / Transform working parameters to n a t u r a l parameters :* | | | |  |
| vector *<*Type*>* lambda = log lambda . exp ( ) ; | | | |  |
| matrix*<*Type*>* gamma = Gamma w2n(m, tgamma ) ; | | | |  |
|  | | | |  |



|  |
| --- |
| foo = ( d e l t a \* vector *<*Type*>*(emission probs . row ( 0 ) ) ) . matrix ( ) ; |
| sumfoo = foo . sum ( ) ; |
| l s c a l e = 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 ( ) ; |
| l s c a l e += log ( sumfoo ) ; |
| foo /= sumfoo ; |
| } |
| mllk = −l s c a l e ; |
|  |
| */ / Use adreport on v a r i a b l e s f o r which we want standard e r r o r s* |
| ADREPORT( lambda ) ; |
| ADREPORT(gamma ) ; |
| ADREPORT( d e l t a ) ; |
|  |
| */ / V a r i a b l e s we need f o r conveniency* |
| REPORT( lambda ) ; |
| REPORT(gamma ) ; |
| REPORT( d e l t a ) ; |
| REPORT( n ) ; |
| REPORT( emission probs ) ; |
| REPORT( mllk ) ; |
|  |
| return mllk ; |
| } |

It should be noted that within TMB, 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.

In C++, comparisons involving NaN values are always false, except when testing inequality between 2 NaN values. In particular, for a float f, the expression f != f will be true only if f is a NaN value. Similarly, f == f returns false if and only if f is a NaN value.

3.3.3 Optimization

Once the likelihood function is written in C++, we can tell R to optimize its parameters.

1. 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", "-O1 -g", DLLFLAGS = "") |

## [1] 0

*# Load it* **dyn.load**(**dynlib**("code/poi\_hmm")) *# Load the parameter transformation function* **source**("functions/utils\_no\_tmb.R")

1. Loading the dataset, see Section 6.1 for more details.

**load**("data/fetal-lamb.RData") lamb\_data <- lamb

1. Creating natural parameters as initial values for the optimizer.

We use the best 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) |

Below is a possible way to automate the creation of initial parameters given *m*.

|  |
| --- |
| **if** (m == 1) {  gamma <- **matrix**(1)  } **else** { gamma <- **matrix**(0.2 / (m - 1), nrow = m, ncol = m) **diag**(gamma) <- 0.8  }  lambda <- **seq**(0.1, 4, length.out = m) |

1. Transforming them to working parameters

*# Turn them into working parameters* parameters <- **pois.HMM.pn2pw**(m, lambda, gamma) parameters <- **list**(log\_lambda = parameters[1:m],

tgamma = parameters[- (1:m)])

1. Creating the TMB likelihood function object

obj\_tmb <- **MakeADFun**(TMB\_data, parameters, DLL = "poi\_hmm",

silent = TRUE)

1. Optimizing

mod\_tmb <- **nlminb**(start = obj\_tmb$par, objective = obj\_tmb$fn)

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

1. 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

1. Getting the ML estimates of the natural parameters, sent by REPORT, in a format easy to work with

ML\_working\_estimates <- obj\_tmb$env$last.par

ML\_natural\_estimates <- obj\_tmb$**report**(ML\_working\_estimates)

ML\_natural\_estimates$lambda

## [1] 0.2563653 3.1147507

ML\_natural\_estimates$gamma

## [,1] [,2]

## [1,] 0.9887324 0.01126755

## [2,] 0.3103387 0.68966126

ML\_natural\_estimates$delta

## [1] 0.96496476 0.03503524

1. It is possible to tell the optimizer to use 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

ML\_working\_estimates <- obj\_tmb$env$last.par

ML\_natural\_estimates <- obj\_tmb$**report**(ML\_working\_estimates)

ML\_natural\_estimates$lambda

## [1] 0.2563654 3.1147543

ML\_natural\_estimates$gamma

## [,1] [,2]

## [1,] 0.9887213 0.01127872

## [2,] 0.3103385 0.68966147

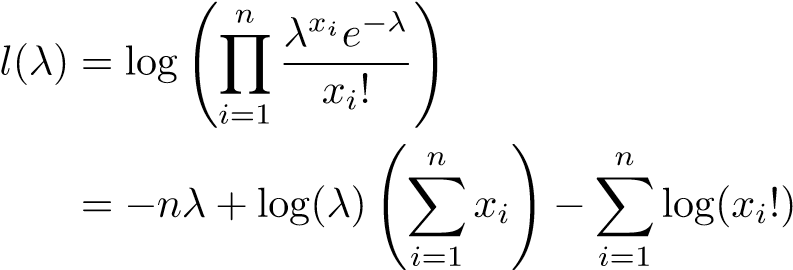
ML\_natural\_estimates$delta

## [1] 0.96493123 0.03506877

The use of an exact gradient and hessian provided by TMB can make the model fit better, although in our case it doesn’t make a 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 they use a modified likelihood and we don’t. This is clear when comparing the likelihoods with only 1 hidden state.

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

*.*

They find a ML estimate *λ* = 0*.*3583 and a log-likelihood of -174.26.

|  |
| --- |
| x <- lamb\_data n <- **length**(x) lambda <- 0.3583  - n \* lambda + **log**(lambda) \* **sum**(x) - **sum**(**log**(**factorial**(x)))  ## [1] -201.0436 |

We can see that the log-likelihood is different, but when we remove the constant , it becomes

- n \* lambda + **log**(lambda) \* **sum**(x)

## [1] -174.2611

In our estimation procedures, we use the complete formula for the log-likelihood. In addition we don’t use the Expectation-Maximization (EM) algorithm like they do. Therefore our likelihood differs sligthly from the paper.

3.4 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.

In our situation, 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 be different, the fit worsens.

To showcase the advantage of nested models, we will make a model with 2 hidden states.

|  |
| --- |
| *# 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) **if** (m == 1) {  gamma <- **matrix**(1)  } **else** { gamma <- **matrix**(0.2 / (m - 1), nrow = m, ncol = m) **diag**(gamma) <- 0.8  }  *# Turn them into working parameters* parameters <- **pois.HMM.pn2pw**(m, lambda, gamma) parameters <- **list**(log\_lambda = parameters[1:m], tgamma = parameters[- (1:m)])  obj\_tmb <- **MakeADFun**(TMB\_data, parameters, DLL = "poi\_hmm", silent = TRUE)  mod\_tmb <- **nlminb**(start = obj\_tmb$par, objective = obj\_tmb$fn,  gradient = obj\_tmb$gr, hessian = obj\_tmb$he)  ML\_working\_estimates <- obj\_tmb$env$last.par.best  ML\_natural\_estimates <- obj\_tmb$**report**(ML\_working\_estimates)  ML\_gamma <- ML\_natural\_estimates$gamma  ML\_lambda <- ML\_natural\_estimates$lambda ML\_delta <- ML\_natural\_estimates$delta |

TMB can be instructed to treat some parameters as constants. We will arbitrarily fix *λ*1 to 1.

|  |
| --- |
| *# # Replace low values with 0*  *# threshold <- 0.02*  *# fixed\_par\_gamma <- replace(ML\_gamma,*  *# list = which(ML\_gamma < threshold),*  *# values = 0)*  *# fixed\_par\_lambda <- replace(ML\_lambda,*  *# list = which(ML\_lambda < threshold),*  *# values = 0)*  *#*  *# # Each row of gamma needs to sum to 1, so we distribute*  *# # the difference on each value that is not fixed to 0*  *# for (row in 1:nrow(fixed\_par\_gamma))* {  *# idx\_replacement <- which(ML\_gamma[row, ] < threshold)*  *# difference <- 1 - sum(fixed\_par\_gamma[row, ])*  *# difference <- difference / (m - length(idx\_replacement)) #*  *# old\_value <- fixed\_par\_gamma[row, -idx\_replacement]*  *# fixed\_par\_gamma[row, -idx\_replacement] <- old\_value + difference #* }  *# Get the previous values, and fix some* fixed\_par\_lambda <- lambda fixed\_par\_lambda[1] <- 1 fixed\_par\_gamma <- gamma  *# # Each row of gamma needs to sum to 1, so we distribute*  *# # the difference on each value that is not fixed*  *# for (row in c(1, 3))* {  *# idx\_replacement <- row*  *# difference <- 1 - sum(fixed\_par\_gamma[row, ])*  *# difference <- difference / (m - length(idx\_replacement)) #*  *# old\_value <- fixed\_par\_gamma[row, -idx\_replacement]*  *# fixed\_par\_gamma[row, -idx\_replacement] <- old\_value + difference #* } |

Since we can only fix working parameters, we chose to fix a natural parameter for which we can easily identify the corresponding working parameter to fix, for simplicity. Fixing a value in the transition probability matrix would be tricky, so we chose a Poisson mean instead.

Now that the nested model parameters have been made, we need to 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)  fixed\_wpar\_lambda <- new\_parameters[1:m] fixed\_wpar\_gamma <- new\_parameters[-(1:m)] new\_parameters <- **list**(log\_lambda = fixed\_wpar\_lambda, |

tgamma = fixed\_wpar\_gamma)

In order to have TMB treat parameters as constants, the map argument of the MakeADFun function must be a list. This list must contain named vectors filled with NA if the parameter is to be treated as a constant, or factor levels.

Equal factor level is collected to a common value, so we want 1 factor level for each non-fixed parameter, and NA for each fixed parameter. We take increasing numbers to make factors, and replace with NA where necessary.

|  |
| --- |
| *# map\_tgamma <- 1:(m ˆ 2 - m)*  *# replacement\_idx <- which(new\_parameters$tgamma == -Inf)*  *# map\_tgamma[replacement\_idx] <- NA*  *# map\_tgamma <- as.factor(map\_tgamma)*  *#*  *# map\_log\_lambda <- (m ˆ 2 - m) + 1:m*  *# replacement\_idx <- which(new\_parameters$log\_lambda == -Inf)*  *# map\_log\_lambda[replacement\_idx] <- NA*  *# map\_log\_lambda <- as.factor(map\_log\_lambda)*  replacement\_idx <- **c**(TRUE, **rep**(FALSE, m - 1))  map\_tgamma <- 1:(m ˆ 2 - m)  map\_tgamma <- **as.factor**(map\_tgamma)  replacement\_idx <- **head**(replacement\_idx, m) map\_log\_lambda <- (m ˆ 2 - m) + 1:m  map\_log\_lambda[replacement\_idx] <- NA map\_log\_lambda <- **as.factor**(map\_log\_lambda)  map <- **list**(log\_lambda = map\_log\_lambda, tgamma = map\_tgamma)  *# 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) |

Old estimates and standard errors, and the new ones

We can see that both approaches converged properly:

mod\_tmb$convergence ## [1] 0 fixed\_par\_mod\_tmb$convergence

## [1] 0

Before After

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 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 |
| *γ*1*,*1 | 0.99 | 0.01 | 1.00 | 0.00 |
| *γ*2*,*1 | 0.31 | 0.18 | 0.19 | 0.18 |
| *γ*1*,*2 | 0.01 | 0.01 | 0.00 | 0.00 |
| *γ*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 Before and after fixing low parameters to 0

and the likelihood got worse as expected.

|  |
| --- |
| *# The negative log-likelihood is accessed by the attribute*  *# "objective" of the optimized object* mod\_tmb$objective ## [1] 177.5188 fixed\_par\_mod\_tmb$objective  ## [1] 264.1636 |

It should be noted that some inconsistencies can happen. For example, the stationary distribution is a vector of probabilities and should sum to 1. However, we can see that it doesn’t exactly sum to 1.

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 most likely due to machine approximations when making numbers far apart from each other interact together. For example, in R a small number is not 0

1e-100 == 0

## [1] FALSE

but is treated as 0 when added to a much larger number.

(1 + 1e-100 - 1) == 0

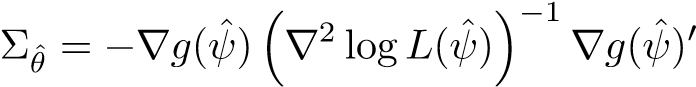
## [1] TRUE

This can result in inconsistencies when numbers far apart from each other are involved. Fortunately, this doesn’t seem to cause any issues.

# Confidence intervals

4.1 Hessian based confidence intervals

First, let us consider evaluation of parameter uncertainty via the Hessian. As the Hessian ∇2 log*L*(*ψ*ˆ) refers to the transformed parameters *ψ* the delta-method must be used to obtain an estimate of the covariance matrix of *θ*ˆ:

 (2)

With minimal effort from the users perspective, TMB can be instructed to calculate Σ*θ*ˆ (by automatic differentiation). Standard errors of derived parameters, such as the stationary distribution *δ* which is a function of **Γ**, can be calculated by the delta-method similarly.

TMB provides an easy way to retrieve these. Following the example above, as we saw in subsubsec-

tion 3.3.3,

*# 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 will also show information about the variables reported by the function ADREPORT.

We will now see how to get Wald confidence intervals using TMB. NEED CITATION With a 3 state Poisson HMM, the 100(1 − *α*)% confidence interval for *a* is *a* ± *z*1−*α/*2 ∗ *σa* where *zx* is the *x*-percentile of the standard normal distribution, and *σa* is the standard error of *a*.

|  |
| --- |
| **set.seed**(123)  *# Load TMB and optimization packages* **library**(TMB) **library**(optimr)  *# Run the C++ file containing the TMB code*  TMB::**compile**("code/poi\_hmm.cpp", "-O1 -g", DLLFLAGS = "")  ## [1] 0  *# Load it*  **dyn.load**(**dynlib**("code/poi\_hmm")) |

|  |
| --- |
| ## Warning: 8 external pointers will be removed  *# Load the data*  **load**("data/fetal-lamb.RData") lamb\_data <- lamb  *# Same model as before, 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)  parameters <- **pois.HMM.pn2pw**(m, lambda, gamma) parameters <- **list**(log\_lambda = parameters[1:m], tgamma = parameters[- (1:m)])  obj\_tmb <- **MakeADFun**(TMB\_data, parameters,  DLL = "poi\_hmm", silent = TRUE)  mod\_tmb <- **nlminb**(start = obj\_tmb$par, objective = obj\_tmb$fn,  gradient = obj\_tmb$gr, hessian = obj\_tmb$he)  *# sdreport gives the estimates of the parameters and their standard errors* 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 CI # 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 |

We will show how to retrieve the ML estimates sent by ADREPORT, along with their standard errors

|  |
| --- |
| *# Get the estimates of the transformed variables* adrep <- **summary**(**sdreport**(obj\_tmb), "report") estimates <- adrep[, "Estimate"] std\_errors <- adrep[, "Std. Error"]  names <- **names**(estimates) == "lambda"  ML\_lambda <- estimates[names]  ML\_lambda\_std\_error <- std\_errors[names]  names <- **names**(estimates) == "gamma"  ML\_gamma <- estimates[names]  ML\_gamma <- **matrix**(ML\_gamma, ncol = m)  ML\_gamma\_std\_error <- std\_errors[names]  ML\_gamma\_std\_error <- **matrix**(ML\_gamma\_std\_error, ncol = m)  names <- **names**(estimates) == "delta"  ML\_delta <- estimates[names]  ML\_delta\_std\_error <- std\_errors[names]  ML\_lambda  ## lambda lambda  ## 0.2563654 3.1147543  ML\_lambda\_std\_error  ## lambda lambda  ## 0.04016445 1.02131181  ML\_gamma  ## [,1] [,2]  ## [1,] 0.9887213 0.01127872  ## [2,] 0.3103385 0.68966147  ML\_gamma\_std\_error  ## [,1] [,2]  ## [1,] 0.01063571 0.01063571  ## [2,] 0.18468648 0.18468648  ML\_delta  ## delta delta  ## 0.96493123 0.03506877  ML\_delta\_std\_error  ## delta delta  ## 0.03181445 0.03181445 |

Estimates of *θ* and *δ*, with accompanying confidence intervals are displayed in Table 11 and Table 12. For comparison, we have included the corresponding standard deviations resulting from replacing ∇2 log*L*(*ψ*ˆ) in (Equation 2) with traditional numerical approximations using the R functions nlm (nlmb) and fdHess (nlme). The difference when using these approximations are of order 1*e* − 05 or less, however one would still have to calculate ∇*g*(*ψ*ˆ) for these approximations to be useful.

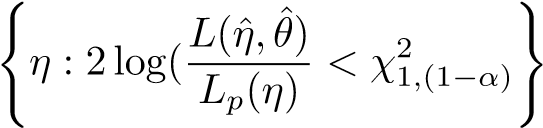
As mentioned earlier, for a larger amount of hidden states, TMB may be unable to give some or any standard standard errors because some variables are close to their boundaries.

In that situation, using a nested model might solve this issue at the cost of the model usually having a worse fit.

Refer to Section 3.4 for information on how to make 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*(*η,θ*), and let *Lp*(*η*) be the profile likelihood defined by *Lp*(*η*) = max*θ L*(*η,θ*). Then a likelihood-based CI for *η* is given by

 (3)

where is the 1 − *α* quantile of a *χ*2 distribution with 1 degree of freedom. TMB allows for very efficient computation of both the profile likelihood *L*(*η*) and the CI given by (Equation 3), and this has been used to produce Table 11 and Table 12 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*(*η*)*,Lp*(*g*(*η*))} = {*g*(*η*)*,Lp*(*η*)} 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 loglambda2, we have to feed its position into the name argument:

profile <- **tmbprofile**(obj = obj\_tmb, name = 2, trace = FALSE)

*# Working parameter log\_lambda* **plot**(profile)

**confint**(profile)

## lower upper

## log\_lambda 0.2353399 1.598929

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

## lower upper

## log\_lambda 1.265339 4.947733

0.2

0.4

0.6

0.8

1.0

1.2

1.4

1.6

177.5

178.0

178.5

179.0

179.5

value

log\_lambda

Figure 1 Profile likelihood

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 loglambda 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. Nonparametric bootstraping of time series is possible, as Hardle et al.¨ (2003) has shown. But since our focus is not on bootstrapping techniques, we chose 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.

1. 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)  } |

1. 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. In short, 1 → 3*,*2 → 1*,and*3 → 2.

1. 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"] log\_lambda <- ML\_working\_estimates[cols == "log\_lambda"] ML\_TMB\_parameters <- **list**(log\_lambda = log\_lambda, tgamma = tgamma)  params\_names <- **c**("lambda", "gamma", "delta") |

{

|  |
| --- |
| **for** (idx\_sample **in** 1:BOOTSTRAP\_SAMPLES) {  mod\_temp <- **NULL** natural\_parameters <- **NULL**  *# This while loop is in case tmb doesn't converge for some reason*  *# And in case some parameter estimates are NA values* **while** (**is.null**(mod\_temp) | **anyNA**(natural\_parameters[params\_names], recursive = TRUE))  *#simulate the data* bootstrap\_data <- **pois.HMM.generate\_sample**(ns = DATA\_SIZE,  mod = **list**(m = m, lambda = gamma = 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)  *# Sometimes TMB doesn't converge, maybe the sample*  *# has too many extremes*  *# In those cases, the data is regenerated through the "while"*  *# The error messages are useless in those cases* **try**(mod\_temp <- **nlminb**(start = obj$par, objective = obj$fn, gradient = obj$gr, hessian = obj$he),  silent = TRUE)  **if** (!**is.null**(mod\_temp)) { working\_parameters <- obj$env$last.par.best natural\_parameters <- obj$**report**(working\_parameters)  *# Label switching*  natural\_parameters <- **pois.HMM.label.order**(m,  natural\_parameters natural\_parameters natural\_parameters  }  }  *# The values from gamma are taken columnwise*  natural\_parameters <- **unlist**(natural\_parameters[params\_names]) len\_par <- **length**(natural\_parameters) |

lambda,

gamma, delta))

$lambda,

$gamma,

$delta)

|  |
| --- |
| bootstrap\_estimates[idx\_sample, 1:len\_par] <- natural\_parameters  }  q <- **apply**(bootstrap\_estimates, 2, **function**(par\_estimate) { **quantile**(par\_estimate, probs = **c**(0.025, 0.975))  })  *# Get row column indexes for gamma instead of the default column* params\_names <- **paste0**(**rep**("lambda", m), 1:m) **for** (gamma\_idx **in** 1:m ˆ 2) { row\_col\_idx <- **matrix.col.idx.to.rowcol**(gamma\_idx, m) 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, right = FALSE, row.names = FALSE)  ## Parameter Estimate Lower.bound Upper.bound  ## lambda1 0.25636541 0.116387922 0.32108168  ## lambda2 3.11475432 0.747766566 4.15080417  ## gamma11 0.98872128 0.929839268 0.99379528  ## gamma21 0.31033853 0.063464711 0.62099744  ## gamma12 0.01127872 0.006204724 0.07016073  ## gamma22 0.68966147 0.379002556 0.93653529  ## delta1 0.96493123 0.339074033 0.98651955  ## delta2 0.03506877 0.013480450 0.66092597 |

1. 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.

# State inference

5.1 Setup

Given an optimized MakeADFun object obj, we need to setup some variables 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  }  *# Since lalpha contains n=240 columns, 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)  } |

*# Since lbeta contains n=240 columns, we only display 5 for readability* lbeta[, 1:5]

## [,1] [,2] [,3] [,4] [,5]

## [1,] -177.2275 -176.9600 -176.6925 -176.4250 -176.1576

## [2,] -178.3456 -178.0781 -177.8099 -177.5253 -176.9034

5.4 Smoothing probabilities

The smoothing probabilities are defined in Zucchini et al. .

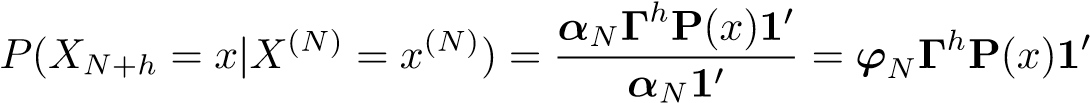
|  |
| --- |
| *# Compute conditional state probabilities, smoothing probabilities* stateprobs <- **matrix**(NA, ncol = n, nrow = m) llk <- - mllk **for**(i **in** 1:n) {  stateprobs[, i] <- **exp**(lalpha[, i] + lbeta[, i] - llk)  }  *# Most probable states* ldecode <- **rep**(NA, n) **for** (i **in** 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 1 1 1 1 1 1  ## [38] 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 1 1 1 1  ## [75] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  ## [112] 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 1 1 1 1  ## [149] 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 1 1 1 1  ## [186] 1 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 1 1 1 1  ## [223] 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 .

Then,

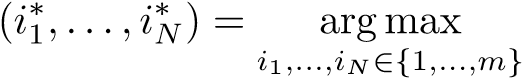


An implementation of this, using a scaling scheme is

|  |
| --- |
| *# 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]  }  }  *# Since lbeta contains n=240 columns, we only display 5 for readability* dxf[, 5]  ## [1] 0.002181533 |

5.6 Global decoding using the Viterbi algorithm

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

*P*(*C*1 = *i*1*,...,CN* = *iN*|*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 1 1 1 1 1

## [38] 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 1 1 1 1

## [75] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

## [112] 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 1 1 1 1

## [149] 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 1 1 1 1

## [186] 1 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 1 1 1 1

## [223] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

# Application to hospital data

6.1 Data description

We timed different parameters on a dataset provided by Leroux and Puterman (1992), on a simulated dataset, and on a larger dataset provided by a hospital from Assistance publique – Hopitaux de Paris, aˆ french hospital trust.

The first dataset is the number of movements by a fetal lamb in 240 consecutive 5-second intervals, and is:

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

1. 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
2. 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 2 Numbers of movements by a fetal lamb in 240 consecutive 5-second intervals

The second is a sequence of random numbers generated by a *m*-state Poisson HMM. The code we used is

|  |
| --- |
| DATA\_SIZE\_SIMULATION <- 2000  m <- 2  *# Generate parameters* lambda <- **seq**(1, 7, length.out = m)  *# Create the transition probability matrix with 0.8 on its diagonal* **if** (m == 1) {  gamma <- **diag**(1, m)  } **else** { 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\_SIMULATION,  mod = **list**(m = m, lambda = lambda, gamma = gamma, delta = delta)) |

The last is the number of patients who entered a hospital each hour between the first hour of 2010 and the last hour of 2019. The dataset is available at GITHUB REPO DATASET WEBSITE. The zeroes in the amount are not missing data but indicate that no one arrived at the hospital. The data was grouped using the package dplyr. The column DATE is a datetime item useful for sorting and plotting the data. The columns YEAR is an integer taking values 2010, 2011, ..., 2019. The column PATIENTS represents the number of people arriving at the hospital between the hour mentioned and the next. For example, the dataset starts on the hour 0 with an amount of 6, and indicates that 6 people arrived between 00:00 and 00:59. The column WDAY represents the number of the day in the week: 1 is Monday and 7 is Sunday.

6.2 Speed comparison

Different numbers of hidden states *m* were used: (1*,*2 for the lamb dataset, and 1*,*2*,*3 for the simulated dataset) to compare speeds.

Overall, the acceleration grows with the amount of hidden states *m* and the size of the dataset. For readability, DM denotes the estimation time without using TMB.

TMB1, ..., TMB4 will be used depending on whether TMB gives the gradient and hessian. The following table will summarize the notation

Table 3 Naming of TMB parameters

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

The following Figure 2 below shows the time acceleration when using TMB on the hospital dataset through boxplots, for multiple amounts of hidden states *m*.

**Hospital data, size = 87648**

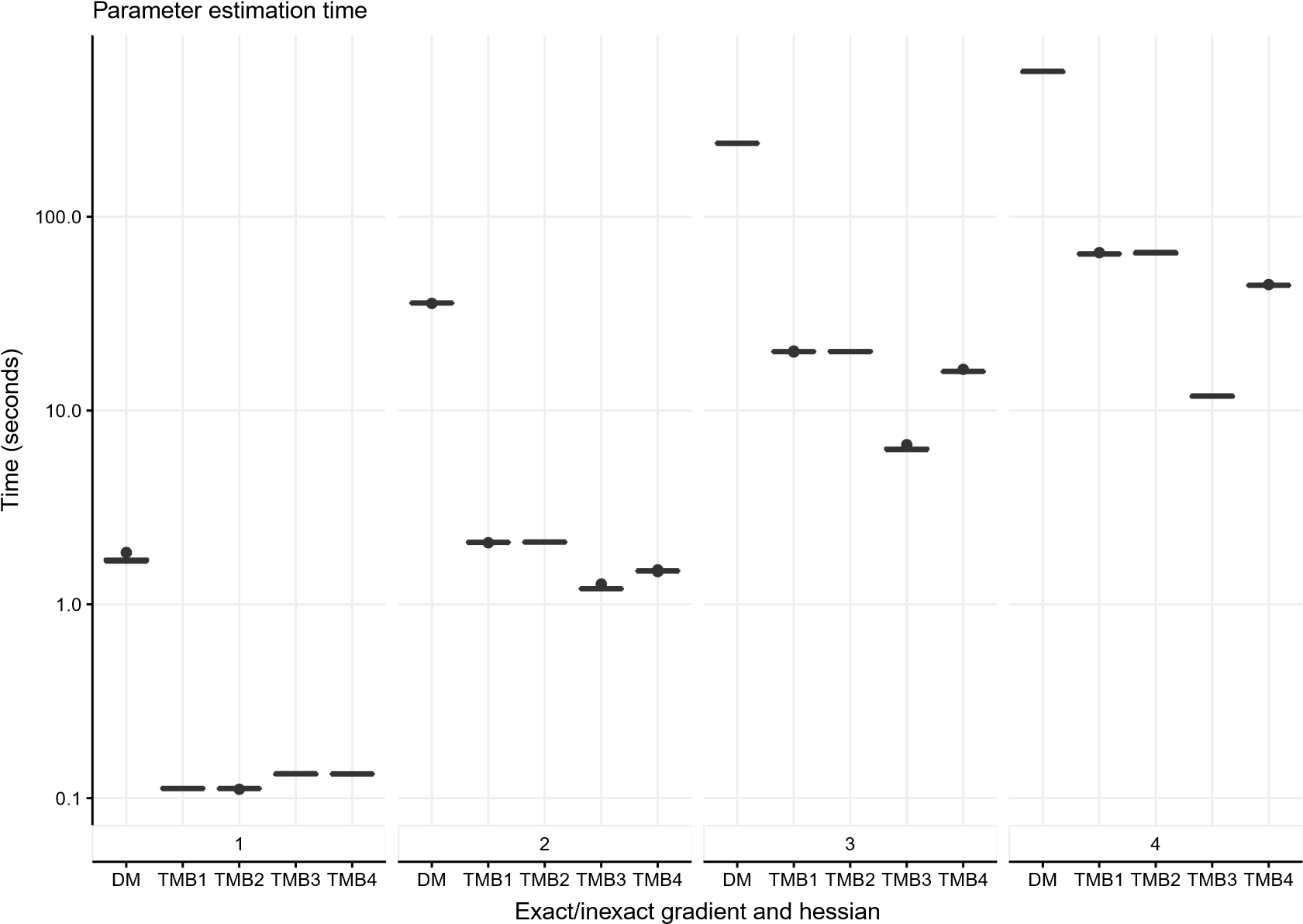


Figure 2 Time w/o using TMB with hospital data The average speed increases from using TMB on the hospital dataset are summarized here:

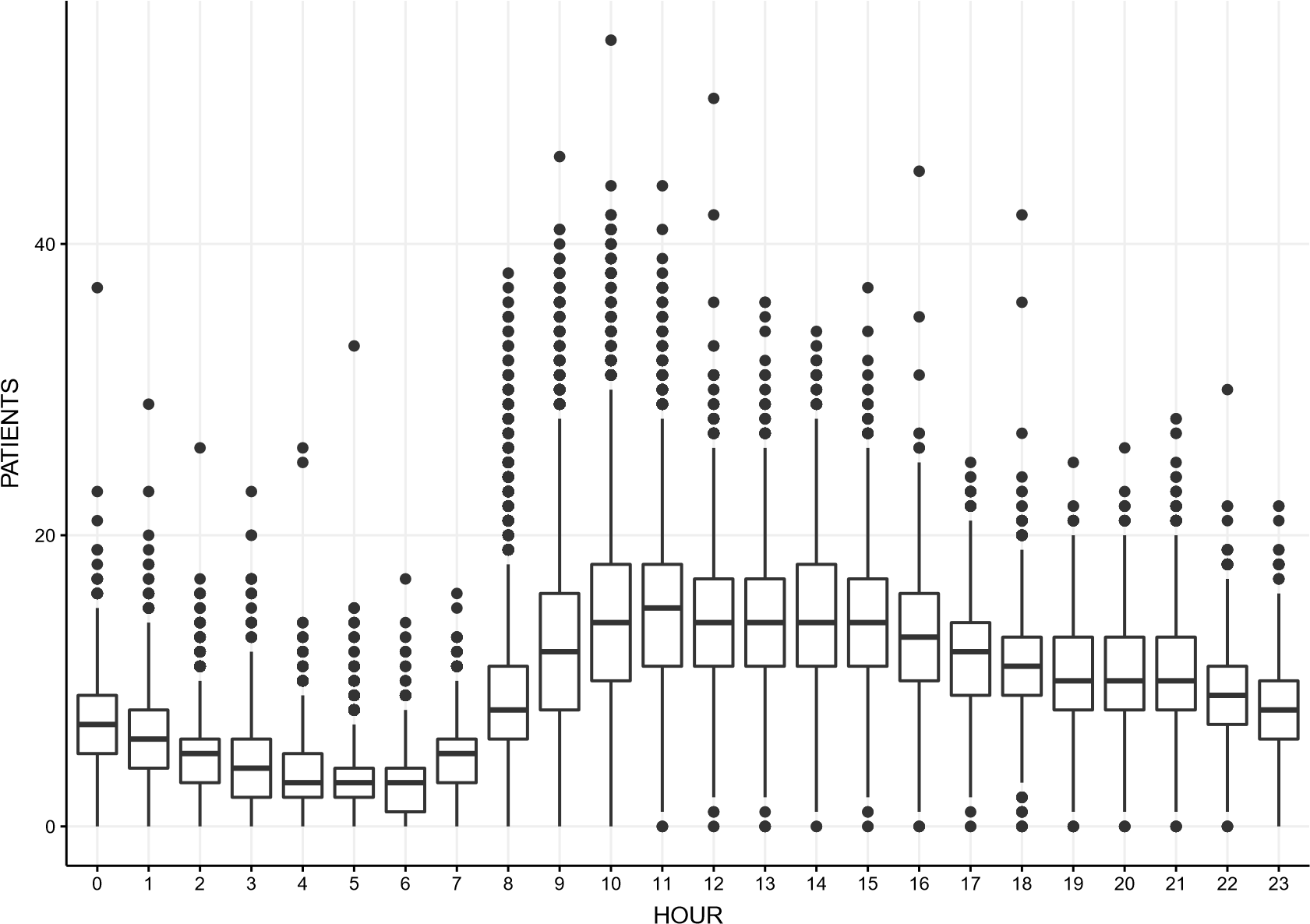
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| m | TMB1 | TMB2 | TMB3 | TMB4 |
| 1 | 1417.27 | 1423.96 | 1183.32 | 1179.11 |
| 2 | 1617.86 | 1614.06 | 2847.93 | 2308.90 |
| 3 | 1087.23 | 1082.24 | 3650.29 | 1399.04 |
| 4 | 769.36 | 761.66 | 4632.82 | 1163.51 |

Table 4 Speed percentage increase of TMB estimation on the hospital dataset

6.3 Interpretation

Here are the boxplots of hourly arrivals.

**Hourly arrivals**



As could have been guessed, more patients arrive in the day rather than during the night, in the afternoon rather than in the evening. Also, most patients show up in the early part of the day and around noon. From this, we can see different distributions in the hourly arrivals of patients, justifying 3 or 4 hidden states.

Here we can see the distribution of the states of a 4 state Poisson HMM fitted on the hourly arrivals.

1234 1234 1234 1234 1234 123 1234 1234 1234 1234 1234 1234 1234 1234 1234 1234 1234 1234 1234 1234 1234 1234 hour

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

0

1

2

3

4

0

1000

2000

3000

amount

23

1

2

3

state

1

2

3

4

This graph shows the amount of states per hour across the entire dataset. The states (1, 2, 3, 4) have means (3.816, 8.64, 13.186, 22.503) respectively. This model seems to agree with our guess. Here is a table of the negative log-likelihoods corresponding to different values of *m*.

|  |  |  |  |
| --- | --- | --- | --- |
| m | nll | AIC | BIC |
| 1 | 324288.014452837 | 648578.028905674 | 648587.409989746 |
| 2 | 259295.21543868 | 518598.430877361 | 518635.955213649 |
| 3 | 247402.862167235 | 494823.72433447 | 494908.154091118 |
| 4 | 242586.762212778 | 485205.524425557 | 485355.62177071 |

Table 5 Negative log-likelihood values corresponding to different amounts of Poisson HMM states The 4 state model has the best likelihood, the best AIC, and the best BIC. It seems it has the best fit.

# Speed evaluation

7.1 Speed comparison

(i) Speed of TMB

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

The following Figure 3 below shows the time acceleration when using TMB on the lamb dataset from Leroux and Puterman (1992) through boxplots.

**Lamb data, size = 240m = 2**

0.003

0.010

0.030

DM

TMB1

TMB2

TMB3

TMB4

Exact/inexact gradient and hessian

Time (seconds)

Parameter estimation time

Figure 3 Time w/o using TMB with lamb data The average speed increases using the lamb dataset are summarized here:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| m | TMB1 | TMB2 | TMB3 | TMB4 |
| 1 | 2032.43 | 1906.41 | 2088.58 | 1746.91 |
| 2 | 2296.28 | 2285.67 | 4632.70 | 1628.34 |

Table 6 Speed percentage increase of TMB estimation on the lamb dataset

The following Figure 4 below shows the time acceleration when using TMB on a simulated dataset through boxplots.

**Simulated data, size = 2000**

1

2

3

DM

TMB1

TMB2

TMB3

TMB4

DM

TMB1

TMB2

TMB3

TMB4

DM

TMB1

TMB2

TMB3

TMB4

0.01

0.10

1.00

Exact/inexact gradient and hessian

Time (seconds)

Parameter estimation time

Figure 4 Time w/o using TMB with simulated data

The average speed increases using a simulated dataset are summarized here:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| m | TMB1 | TMB2 | TMB3 | TMB4 |
| 1 | 2020.88 | 1987.11 | 1889.85 | 1760.72 |
| 2 | 1764.85 | 1766.93 | 3518.15 | 3082.72 |
| 3 | 1128.18 | 1124.20 | 4996.50 | 3115.07 |

Table 7 Speed percentage increase of TMB estimation on the simulated dataset

1. Log-likelihoods

TMB also allows for a better fit of a HMM. As we can see in the following Table 8, the likelihood gets better when TMB can use the exact gradient.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| m | procedure | nll | AIC | BIC |
| 2 | DM | 177.52 | 363.037673881114 | 376.960229574482 |
| 2 | TMB1 | 183.95 | 375.901830120778 | 389.824385814146 |
| 2 | TMB2 | 183.95 | 375.901830120778 | 389.824385814146 |
| 2 | TMB3 | 177.52 | 363.037673881114 | 376.960229574481 |
| 2 | TMB4 | 177.52 | 363.037673881114 | 376.960229574481 |

Table 8 Negative log-likelihood values

Moreover, the computation time of the likelihood is better with TMB although all of the times are relatively low, as we can see in the following tables.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| m | TMB1 | TMB2 | TMB3 | TMB4 |
| 1 | 2070.17 | 2056.98 | 2086.33 | 2015.90 |
| 2 | 1990.89 | 1978.91 | 2069.45 | 2049.50 |

Table 9 Speed percentage increase of TMB likelihood calculation on the lamb dataset

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| m | TMB1 | TMB2 | TMB3 | TMB4 |
| 1 | 3558.77 | 3588.53 | 3559.43 | 3532.63 |
| 2 | 1779.85 | 1783.50 | 1784.17 | 1790.04 |
| 3 | 1108.62 | 1117.58 | 1100.66 | 1107.30 |

Table 10 Speed percentage increase of TMB likelihood calculation on the simulated dataset

1. All optimization methods

Finally, we compared different optimization methods. The ones we retained were BFGS, NelderMead, L-BFGS-B, nlm, nlminb, and hjn, because the others didn’t converge. We also added the algorithm for least-squares curve fitting from the package marqLevAlg. Exact gradients and hessians are provided by TMB and fed to each algorithm. The speed comparisons are in the following

Figure 5. **Lamb data, size = 240, m = 2**

0.2

0.3

0.5

BFGS

L−BFGS−B

nlm

nlminb

hjn

marqLevAlg

Method

Time (milliseconds)

Parameter estimation time

2

3

5

BFGS

L−BFGS−B

nlm

nlminb

hjn

marqLevAlg

Time (milliseconds)

Parameter estimation time

**Simulated data, size = 2000, m = 3**

Exact/inexact gradient and hessian

Figure 5 Log estimation time vs optimization method 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 show the lower and upper bounds of the confidence intervals. The reason is that sometimes, only one bound of the interval is known and doesn’t let one have a typical 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 | 0.07 | 0.33 | 0.18 | 0.34 |
| 2 | *λ*2 | 3.11 | 1.27 | 4.95 | 0.26 | 5.19 | 1.11 | 5.12 |
| 2 | *γ*1*,*1 | 0.99 | 0.93 | 1.00 | 0.67 | 1.00 | 0.97 | 1.01 |
| 2 | *γ*2*,*1 | 0.31 | 0.04 | 0.68 | 0.05 | 1.00 | -0.05 | 0.67 |
| 2 | *γ*1*,*2 | 0.01 | 0.00 | 0.07 | 0.00 | 0.33 | -0.01 | 0.03 |
| 2 | *γ*2*,*2 | 0.69 | 0.32 | 0.96 | 0.00 | 0.95 | 0.33 | 1.05 |
| 2 | *δ*1 | 0.96 |  |  | 0.47 | 1.00 | 0.90 | 1.03 |
| 2 | *δ*2 | 0.04 |  |  | 0.00 | 0.53 | -0.03 | 0.10 |

Table 11 Estimates and standard errors on the lamb dataset

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| m | Parameter | Parameter.value | Parameter.estimate | Profile.L | Profile.U | Bootstrap.L | Bootstrap.U | TMB.L | TMB.U |
| 1 | *λ*1 | 1.00 | 1.02 |  |  | 0.98 | 1.07 | 0.98 | 1.07 |
| 2 | *λ*1 | 1.00 | 0.96 | 0.90 | 1.02 | 0.90 | 1.02 | 0.90 | 1.02 |
| 2 | *λ*2 | 20.00 | 19.93 | 19.66 | 20.21 | 19.67 | 20.20 | 19.66 | 20.21 |
| 2 | *γ*1*,*1 | 0.80 | 0.79 | 0.76 | 0.81 | 0.76 | 0.81 | 0.76 | 0.81 |
| 2 | *γ*2*,*1 | 0.20 | 0.21 | 0.18 | 0.23 | 0.18 | 0.24 | 0.18 | 0.23 |
| 2 | *γ*1*,*2 | 0.20 | 0.21 | 0.19 | 0.24 | 0.19 | 0.24 | 0.19 | 0.24 |
| 2 | *γ*2*,*2 | 0.80 | 0.79 | 0.77 | 0.82 | 0.76 | 0.82 | 0.77 | 0.82 |
| 2 | *δ*1 | 0.50 | 0.49 |  |  | 0.46 | 0.54 | 0.45 | 0.54 |
| 2 | *δ*2 | 0.50 | 0.51 |  |  | 0.46 | 0.54 | 0.46 | 0.55 |
| 3 | *λ*1 | 1.00 | 0.97 | 0.89 | 1.04 | 0.90 | 1.05 | 0.89 | 1.04 |
| 3 | *λ*2 | 10.50 | 10.47 | 10.16 | 10.77 | 10.13 | 10.79 | 10.16 | 10.77 |
| 3 | *λ*3 | 20.00 | 20.26 | 19.87 | 20.65 | 19.86 | 20.65 | 19.86 | 20.65 |
| 3 | *γ*1*,*1 | 0.80 | 0.79 | 0.75 | 0.84 | 0.76 | 0.82 | 0.76 | 0.83 |
| 3 | *γ*2*,*1 | 0.10 | 0.10 | 0.08 | 0.13 | 0.08 | 0.13 | 0.08 | 0.13 |
| 3 | *γ*3*,*1 | 0.10 | 0.11 | 0.09 | 0.13 | 0.08 | 0.13 | 0.08 | 0.13 |
| 3 | *γ*1*,*2 | 0.10 | 0.10 | 0.08 | 0.12 | 0.07 | 0.12 | 0.07 | 0.12 |
| 3 | *γ*2*,*2 | 0.80 | 0.77 | 0.72 | 0.82 | 0.73 | 0.81 | 0.73 | 0.81 |
| 3 | *γ*3*,*2 | 0.10 | 0.11 | 0.09 | 0.14 | 0.08 | 0.14 | 0.08 | 0.14 |
| 3 | *γ*1*,*3 | 0.10 | 0.11 | 0.09 | 0.13 | 0.09 | 0.14 | 0.08 | 0.13 |
| 3 | *γ*2*,*3 | 0.10 | 0.13 | 0.10 | 0.15 | 0.10 | 0.16 | 0.09 | 0.16 |
| 3 | *γ*3*,*3 | 0.80 | 0.78 | 0.73 | 0.83 | 0.74 | 0.82 | 0.75 | 0.82 |
| 3 | *δ*1 | 0.33 | 0.34 |  |  | 0.29 | 0.39 | 0.29 | 0.39 |
| 3 | *δ*2 | 0.33 | 0.31 |  |  | 0.26 | 0.36 | 0.27 | 0.36 |
| 3 | *δ*3 | 0.33 | 0.35 |  |  | 0.30 | 0.40 | 0.30 | 0.40 |

Table 12 Estimates and standard errors on the simulated dataset

All the code used to produce those results is available in the GitHub repository GITHUB REPO. The files executing the code are poi hmm lamb.R, poi hmm simul.R, and poi hmm hosp.R.

# Discussion

1. Throughout this paper, when we actually used hidden states (*m >* 1), we found that TMB3 yields the greatest speed increase (when TMB’s exact gradient is provided to the optimizer but not the hessian). This also stayed true when using other optimizers such as nlm. The reason is unclear. 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.
2. 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

1. Code to setup global parameters and declare functions used
2. Packages used
3. Functions used in TMB related code
4. Functions used in non TMB related code
5. Code to run estimations and comparisons using the lamb dataset
6. Code to run estimations and comparisons using a simulated dataset
7. Code to run estimations and comparisons using the hospital dataset

A.2. C++ Code

1. Poisson HMM negative log-likelihood calculation
2. Functions used in C++ Poisson HMM code
3. Linear model negative log-likelihood calculation
4. Functions used in C++ linear model code

A.1. Additional graphs

1. simple C acceleration of Zucchini scripts p. 333, A 1.7, A 1.8 with conditional probabilities outside of the forward / backward loop
2. Use same order as in Zucchini
3. the .cpp file with transformation code
4. the .cpp file with likelihood
5. an .R file showcasing the use

# References

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. OCLC: 797437299.

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

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 Hardle, 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 MarkovDependent Mixture Models. *Biometrics*, 48(2):545–558, 1992. ISSN 0006-341X. doi: 10.2307/2532308.

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.

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: 10.2307/ 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.

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.

1. Corresponding author: e-mail: timothee.bacri@uib.no, Phone: +33-636-775-063 [↑](#footnote-ref-1)