Introduction to implementing hidden Markov models

Théo Michelot*

December 21, 2016

We provide some advice and chunks of R code, to get started with the implementation of hidden Markov models. In particular, we present two examples: a very basic 2-state Poisson HMM (Section 2), and a slightly more complex 3-state HMM inspired by movement models (Section 3). We hope that we did not leave any inaccuracies in the text or pieces of code, but please let us know if you find a mistake in this document.

Contents

*tmichelot1@sheffield.ac.uk

1 Theoretical ideas

1.1 Definition of HMMs

A hidden Markov model (HMM) is a time series model, comprising two processes: a series of (possibly multivariate) observations $(\boldsymbol{Z}_t)_{t=1}^T$, and a sequence of unobserved states $(S_t)_{t=1}^T$. The process S_t is taken to satisfy the Markov property, and can take a finite number N of values – which we label, for convenience, $\{1, \dots, N\}$. At each instant t, the observation $\boldsymbol{Z}_t$ is drawn from one of N different distributions, according to the value S_t of the state process. We call "N-state HMM" a HMM whose state process can take N different values. The dependence structure of a HMM is illustrated in Figure 1.

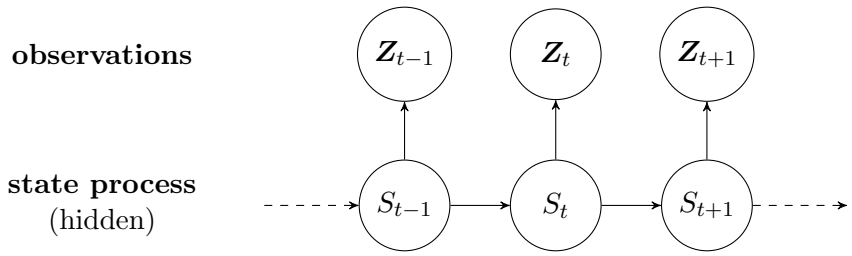


Figure 1: *Dependence structure of a HMM.*

We introduce some notation for the subsequent sections. We denote $\boldsymbol{\delta}$ the initial distribution of the process S_t, i.e.

$$\boldsymbol{\delta} = (\Pr(S_1 = 1), \Pr(S_1 = 2), \dots, \Pr(S_1 = N))$$

We denote γ_{ij} the probability of a transition from state i to state j, and $\boldsymbol{\Gamma}$ the transition probability matrix, i.e.

$$\boldsymbol{\Gamma} = \begin{pmatrix} \gamma_{11} & \cdots & \gamma_{1N} \\ \vdots & \ddots & \vdots \\ \gamma_{N1} & \cdots & \gamma_{NN} \end{pmatrix} = \begin{pmatrix} \Pr(S_t = 1 | S_{t-1} = 1) & \cdots & \Pr(S_t = N | S_{t-1} = 1) \\ \vdots & \ddots & \vdots \\ \Pr(S_t = 1 | S_{t-1} = N) & \cdots & \Pr(S_t = N | S_{t-1} = N) \end{pmatrix}$$

Finally, we denote $\boldsymbol{P}(\boldsymbol{Z}_t)$ the matrix of state-dependent distributions, defined as

$$\boldsymbol{P}(\boldsymbol{Z}_t) = \begin{pmatrix} p(\boldsymbol{Z}_t | S_t = 1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & p(\boldsymbol{Z}_t | S_t = N) \end{pmatrix}$$

1.2 Likelihood computation

In this document, we focus on drawing inference from HMMs using maximum likelihood estimation. We consider a series of T observations $\boldsymbol{z}^{(T)} = \{\boldsymbol{z}_1, ..., \boldsymbol{z}_T\}$, and we denote $\boldsymbol{S}^{(T)} = \{S_1, ..., S_T\}$. Then, in the case of a N-state HMM, the likelihood $\mathcal{L}_T$ can be written

$$\mathcal{L}_T = p(\boldsymbol{Z}^{(T)} = \boldsymbol{z}^{(T)}) = \sum_{S_1=1}^{N} \sum_{S_2=1}^{N} \cdots \sum_{S_T=1}^{N} p(\boldsymbol{Z}^{(T)} = \boldsymbol{z}^{(T)},\ \boldsymbol{S}^{(T)})$$

where each term is,

$$p(\boldsymbol{Z}^{(T)} = \boldsymbol{z}^{(T)},\ \boldsymbol{S}^{(T)}) = p(S_1) \prod_{t=2}^{T} p(S_t | S_{t-1}) \prod_{t=1}^{T} p(\boldsymbol{Z}_t = \boldsymbol{z}_t | S_t)$$

This expression of the likelihood is a sum of N^T terms, each of which is made of $2T$ factors. The complexity of this computation is $\mathcal{O}(TN^T)$, which makes it numerically intractable for large numbers of observations.

It is possible to reduce the number of operations using the forward algorithm (see Section 2.3.2 of Zucchini *et al.*, 2016). Using the notations introduced in Section 1.1, the likelihood is then calculated as

$$\mathcal{L}_T = \boldsymbol{\delta} \boldsymbol{P}(z_1) \boldsymbol{\Gamma} \boldsymbol{P}(z_2) \cdots \boldsymbol{\Gamma} \boldsymbol{P}(z_T) \mathbf{1}', \tag{1}$$

where $\mathbf{1}'$ is a column vector of ones.

This expression of the likelihood is very easy to implement, and can be recursively computed with complexity $\mathcal{O}(TN^2)$, i.e. the computational effort grows linearly with the size of the data. This expression thus allows numerical tractability of the likelihood even for large numbers of observations.

In practice, the expression of the likelihood given in Equation 1 is evaluated using the following scheme, which we implement in the subsequent sections:

$$\mathcal{L}_T = \boldsymbol{\alpha}_T \mathbf{1}',$$

where $\boldsymbol{\alpha}_T$ is computed recursively by

$$\boldsymbol{\alpha}_1 = \boldsymbol{\delta} \boldsymbol{P}(z_1), \qquad \boldsymbol{\alpha}_t = \boldsymbol{\alpha}_{t-1} \boldsymbol{\Gamma} \boldsymbol{P}(z_t) \tag{2}$$

2 Example 1: Poisson HMM

We start with the simple example of a 2-state Poisson HMM, i.e. the observations are drawn from a Poisson distribution, whose rate parameter depends on the underlying state.

2.1 Simulate data

We will start with the simulation of observations from a HMM. The idea is to implement the model described in Section 1.1.

We first define all the parameters of the model, i.e. the initial distribution and transition probability matrix of the state process, and the state-dependent Poisson rate parameters.

```
# transition probability matrix
Gamma <- matrix(c(0.8,0.2,
                  0.1,0.9),
                ncol=2,byrow=TRUE)

# initial distribution
delta <- c(0.5,0.5)

# state-dependent parameters
rate <- c(5,15)
```

The state process is initialized by sampling S_1 to be 1 or 2 (as this is a 2-state HMM), with probabilities determined by the initial distribution. Then, the first observation is generated from a Poisson distribution with rate determined by the value of S_1. In R:

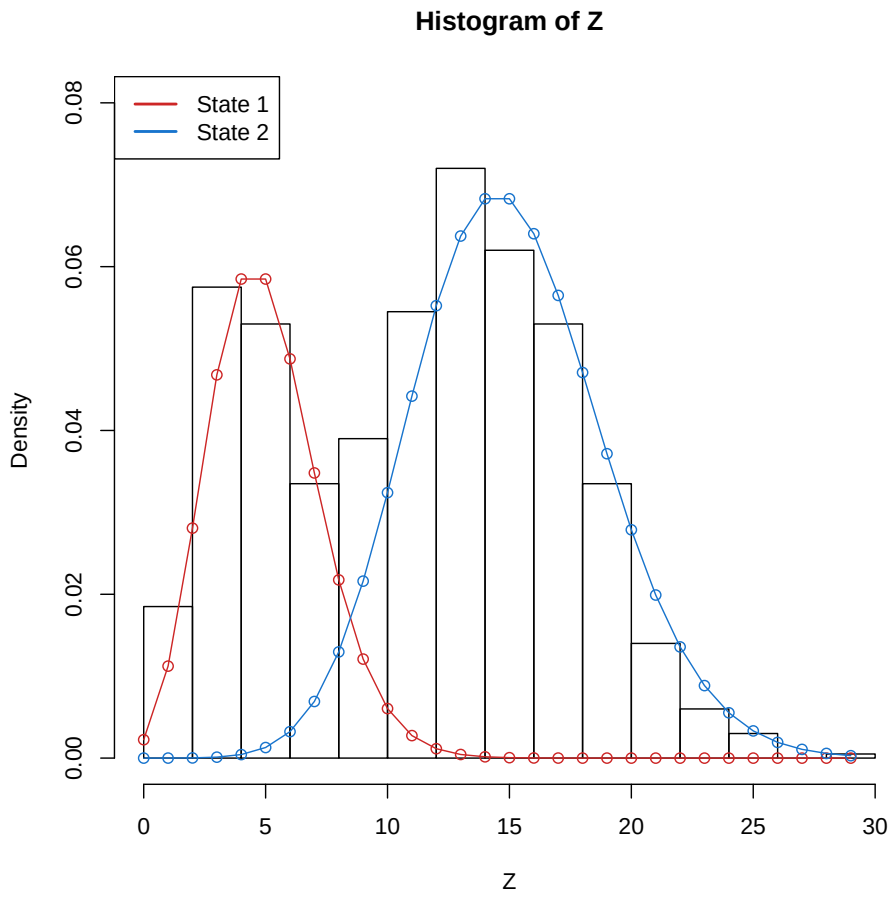
```r
nbObs <- 1000 # number of observations
S1 <- rep(NA,nbObs) # sequence of states
Z <- rep(NA,nbObs) # sequence of observations

# initialize state and observation
S1[1] <- sample(1:2,size=1,prob=delta)
Z[1] <- rpois(1,rate[S1[1]])
```

For $t = 2, \ldots, T$, S_t is chosen with the probabilities specified by the transition probability matrix, i.e. $\Pr(S_t = j | S_{t-1} = i) = \gamma_{ij}$, and $\boldsymbol{Z}_t$ is drawn from a Poisson distribution with rate determined by the value of S_t.

```r
# loop over all observations
for(t in 2:nbObs) {
  S1[t] <- sample(1:2,size=1,prob=Gamma[S1[t-1],])
  Z[t] <- rpois(1,rate[S1[t]])
}
```

To visualise the simulated observations, we can plot a histogram of the data, superimposed with the mass distribution functions of the Poisson distributions used to simulate the data.

```r
# histogram of observations
hist(Z,probability=TRUE,ylim=c(0,0.08))
# lines of state-dependent distributions
# (weighted by stationary distribution)
grid <- seq(min(Z),max(Z))
station <- solve(t(diag(2)-Gamma+1),rep(1,2))
points(grid,dpois(grid,rate[1])*station[1],type="o",col="firebrick3")
points(grid,dpois(grid,rate[2])*station[2],type="o",col="dodgerblue3")
legend("topleft",col=c("firebrick3","dodgerblue3"),
       legend=c("State 1","State 2"),lwd=2)
```



2.2 Implement the likelihood function

The first step towards drawing inference from HMMs is to implement the likelihood function of the model. We take advantage of the very efficient forward algorithm described in Section 1.2, to reduce the computational cost of the maximum likelihood estimation.

We implement the likelihood as a function of the following arguments:

- the observations $(\boldsymbol{z}_t)_{t=1}^T$;
- the parameters of the model, i.e. the initial distribution and transition probability matrix of the state process, and the state-dependent rates of the Poisson distribution.

The implementation of the forward algorithm is straightforward, starting with $\boldsymbol{\delta}\boldsymbol{P}(\boldsymbol{z}_1)$ and iteratively multiplying by $\boldsymbol{\Gamma}\boldsymbol{P}(\boldsymbol{z}_t)$ for each time $t \in \{2, \ldots, T\}$. For computational convenience, we store the $\boldsymbol{P}(\boldsymbol{z}_t)$ in a more compact way: we define `allProbs`, a matrix of dimensions $T \times N$, in which each row t contains the diagonal entries of $\boldsymbol{P}(\boldsymbol{z}_t)$.

```
likPois <- function(Z,rate,Gamma,delta)
{
  # number of observations
  nbObs <- length(Z)

  # probabilities of observations conditional on state
  allProbs <- matrix(1,nrow=nbObs,ncol=2)
```

```
  for(obs in 1:nbObs) {
    allProbs[obs,1] <- dpois(Z[obs],rate[1]) # prob conditional on (state = 1)
    allProbs[obs,2] <- dpois(Z[obs],rate[2]) # prob conditional on (state = 2)
  }

  # forward algorithm
  v <- delta*allProbs[1,]
  for (t in 2:nbObs)
    v <- v%*%Gamma*allProbs[t,]

  return(sum(v))
}
```

We can now use that function on the data simulated in Section 2.1:

```
likPois(Z,rate,Gamma,delta)

## [1] 0
```

The likelihood function returns zero, and will return zero for any values of the parameters. This is due to numerical underflow: the likelihood is computed as the product of many small numbers, and so it is a very small number. It is so small that R rounds it to zero. Here is a trivial illustration of numerical underflow:

```
0.1^10

## [1] 1e-10

0.1^100

## [1] 1e-100

0.1^1000

## [1] 0
```

The solution to this problem is to calculate the log-likelihood function instead of the likelihood. Because the logarithm function is strictly monotonic, the maximum of the likelihood and that of the log-likelihood are reached for the same values of the parameters.

The likelihood is computed as a matrix product, and it is not possible to obtain the log-likelihood by simply taking the sum of the logarithm of its factors. Thus, we use the trick suggested by Zucchini *et al.* (2016) (see Chapter 3.2), that they call "scaling", and write the log-likelihood function as

```
logLikPois <- function(Z,rate,Gamma,delta)
{
  nbObs <- length(Z)

  # probabilities of observations conditional on state
  allProbs <- matrix(1,nrow=nbObs,ncol=2)
```

```
  for(obs in 1:nbObs) {
    allProbs[obs,1] <- dpois(Z[obs],rate[1]) # prob conditional on (state = 1)
    allProbs[obs,2] <- dpois(Z[obs],rate[2]) # prob conditional on (state = 2)
  }

  # forward algorithm (with scaling)
  v <- delta*allProbs[1,]
  llk <- 0
  for (t in 2:nbObs) {
    v <- v%*%Gamma*allProbs[t,]

    # scaling
    llk <- llk+log(sum(v))
    v <- v/sum(v)
  }

  return(llk)
}
```

Note that only a couple of lines of code have been added, and that these lines remain exactly the same for any HMM formulation. Now, if we compute the log-likelihood of the simulated data, we can verify that it returns a finite number (i.e. the likelihood is not zero).

```
logLikPois(Z,rate,Gamma,delta)

## [1] -2905.149
```

2.3 Numerical optimization of the likelihood

Now that we have implemented the log-likelihood function, we want to estimate the parameters of the model by maximum likelihood estimation. To do so, we use the optimizer `nlminb` in R, which roams the parameter space until finding the maximum of the function, and returns the value of the parameters which maximizes the function.

We need to make a few changes in the implementation of the likelihood function, to be able to apply numerical optimization with `nlminb`.

- `nlminb` is a minimizer, and not a maximizer. Thus, we need to find the parameters which minimize the negative log-likelihood, which is equivalent to identifying the parameters which maximize the (positive) log-likelihood.
- `nlminb` optimizes a function over a vector of parameters, so we need to store all the parameters of the model in one vector.

The negative log-likelihood of the Poisson HMM can be implemented as follows.

```
nLogLikPois <- function(Z,par)
{
  nbObs <- length(Z)

  # unpack vector of parameters
```

```
  rate <- par[1:2]
  Gamma <- matrix(NA,nrow=2,ncol=2)
  Gamma[,1] <- par[3:4]
  Gamma[,2] <- 1-Gamma[,1] # use row constraints
  delta <- c(par[5],1-par[5])

  # probabilities of observations conditional on state
  # (vectorized instead of iterative, for speed)
  allProbs <- matrix(1,nrow=nbObs,ncol=2)
  allProbs[,1] <- dpois(Z,rate[1])
  allProbs[,2] <- dpois(Z,rate[2])

  v <- delta*allProbs[1,]
  llk <- 0
  for (t in 2:nbObs) {
    v <- v%*%Gamma*allProbs[t,]
    llk <- llk+log(sum(v))
    v <- v/sum(v)
  }

  return(-llk)
}
```

We do not need to store all transition probabilities in the vector of parameters, because of the row constraints, so we choose to only store the first column. Similarly, we only store the first element of the initial distribution. We can check that the function returns the negative log-likelihood:

```
par <- c(rate,Gamma[,1],delta[1])
nLogLikPois(Z,par)

## [1] 2905.149
```

We can now minimize the negative log-likelihood function, using `nlminb`. To do so, we need to choose initial values for the parameters of the model (the point from which the optimization routine will start its search). The choice of the initial parameter values is crucial, and the optimizer might not be able to reach the global maximum of the function if the starting point is poorly chosen.

In addition, `nlminb` is a constrained optimizer, so we must specify the bounds of the parameter space, with its arguments `lower` and `upper`. The Poisson rates are in $[0, \infty)$, while the transition probabilities and the initial probabilities are in $[0, 1]$.

```
# initial parameters
rate0 <- c(3,10)
Gamma0 <- matrix(c(0.9,0.1,0.1,0.9),ncol=2)
delta0 <- c(0.5,0.5)
par0 <- c(rate0,Gamma0[,1],delta0[1])

# fit model
model <- nlminb(start=par0,objective=nLogLikPois,Z=Z,
```

```
                lower=c(0,0,0,0,0),upper=c(Inf,Inf,1,1,1))

# estimated minimum negative log-likelihood
model$objective

## [1] 2903.241

# parameter estimates
model$par

## [1]  4.9318071 14.7702122  0.7839544  0.1012449  0.0000000
```

And, if we reformat the parameter estimates, we can compare them to the simulation parameters:

```
rateMLE <- model$par[1:2]
GammaMLE <- matrix(c(model$par[3:4],1-model$par[3:4]),ncol=2)
deltaMLE <- c(model$par[5],1-model$par[5])

rateMLE

## [1]  4.931807 14.770212

GammaMLE

##           [,1]      [,2]
## [1,] 0.7839544 0.2160456
## [2,] 0.1012449 0.8987551

deltaMLE

## [1] 0 1
```

The estimates for the rates and the transition probabilities are consistent with the parameters used in the simulation. Moreover, as explained in Section 4.2.4 of Zucchini *et al.* (2016), the maximum likelihood estimate of $\boldsymbol{\delta}$ is expected to be a unit vector – i.e., in a 2-state model, either $(0, 1)$ or $(1, 0)$ – as we see here.

3 Example 2: movement HMM

In this second example, we implement a 3-state HMM, inspired by the models used to analyse animal movement data, as described by Langrock et al. (2012). Movement data usually consists in the animal's locations, collected at regular time intervals. Because the movement is observed in two dimensions (longitude and latitude values), we consider bivariate observations for this model. We use the animal's step lengths (distance between successive locations) and turning angles (angle between successive segments) as metrics of interest. The dependence structure of the model is shown in Figure 2.

We choose the gamma and von Mises distributions to model the step lengths and turning angles, respectively. That is, at time t, the step length is modelled by a gamma distribution and the turning angle by a von Mises distribution, whose parameters depend on the state

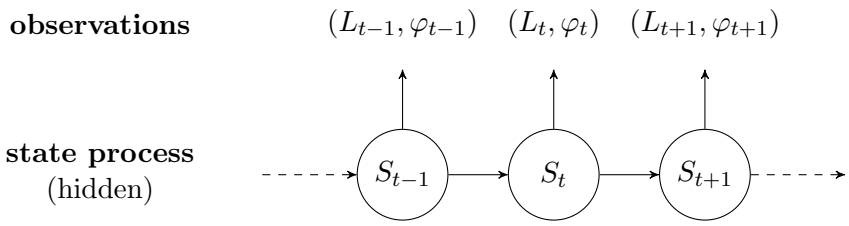


Figure 2: *Dependence structure of the movement HMM, with L_t the step length and φ_t the turning angle at time t.*

S_t. The gamma distribution depends on two parameters, which can either be the mean and standard deviation, or the shape and rate. The former are easier to think about, but R expects the latter, so we use the following formulas,

$$\text{shape} = \frac{\text{mean}^2}{\text{SD}^2}, \qquad \text{rate} = \frac{\text{mean}}{\text{SD}^2}.$$

The von Mises distribution takes two parameters: the mean angle – in $(-\pi, \pi]$ – and the concentration – strictly positive – which is analogous to the inverse of the variance.

3.1 Simulate data

We choose the values of the simulation parameters, i.e. the parameters of the state-dependent distributions, the transition probabilities, and the initial distribution. We choose the parameters to mimic a realistic pattern of animal movement: longer step lengths are associated with more directed movement (angles centred on 0, state 3), and shorter steps with more turnings (angles centred on π, state 1).

```
# transition probability matrix
Gamma <- matrix(c(0.9,0.05,0.05,
                  0.08,0.84,0.08,
                  0.05,0.15,0.8),
                ncol=3,byrow=TRUE)

# initial distribution
delta <- c(1,1,1)/3

# state-dependent parameters
stepMean <- c(0.5,5,30) # mean of gamma dis
stepSD <- c(0.5,4,15)
angleMean <- c(pi,0,0)
angleCon <- c(1,1,10)

# the gamma distribution in R uses shape/rate instead of mean/SD
stepShape <- stepMean^2/stepSD^2
stepRate <- stepMean/stepSD^2
```

We initialise the processes, as before. The state at time 1 is drawn from $\{1, 2, 3\}$, with probabilities determined by the initial distribution. Then, the first step and first angle are drawn from a gamma and von Mises distributions, respectively, with parameters determined by the state S_1.

```
library(CircStats) # for von Mises distribution
nbObs <- 1000 # number of observations
S2 <- rep(NA,nbObs) # sequence of states
step <- rep(NA,nbObs) # sequence of steps
angle <- rep(NA,nbObs) # sequence of angles

# initialize state and observations
S2[1] <- sample(1:3,size=1,prob=delta)
step[1] <- rgamma(1,stepShape[S2[1]],stepRate[S2[1]])
angle[1] <- rvm(1,angleMean[S2[1]],angleCon[S2[1]])
```

We loop over observations $2, \ldots, T$:

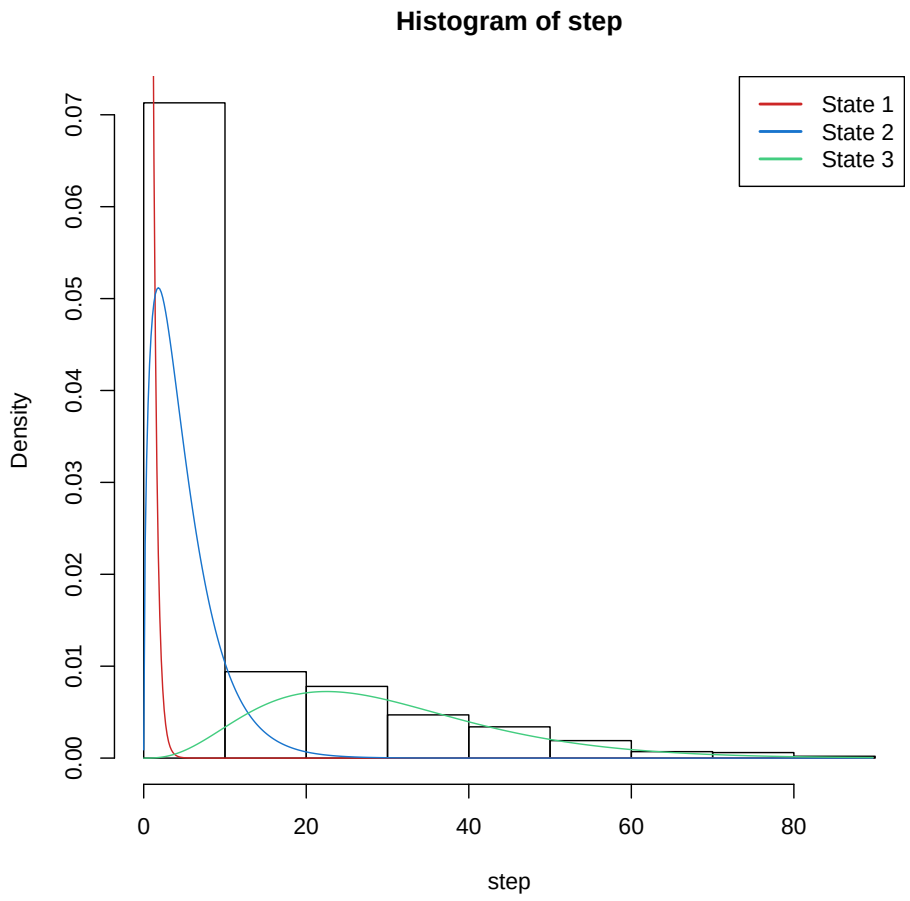
```
for(t in 2:nbObs) {
  S2[t] <- sample(1:3,size=1,prob=Gamma[S2[t-1],])
  step[t] <- rgamma(1,stepShape[S2[t]],stepRate[S2[t]])
  angle[t] <- rvm(1,angleMean[S2[t]],angleCon[S2[t]])
}
```

The function `rvm` generates values of angles between 0 and 2π, but we find it easier to work with angles in $(-\pi, \pi]$, so we move the values in $(\pi, 2\pi]$ to $(-\pi, 0]$:

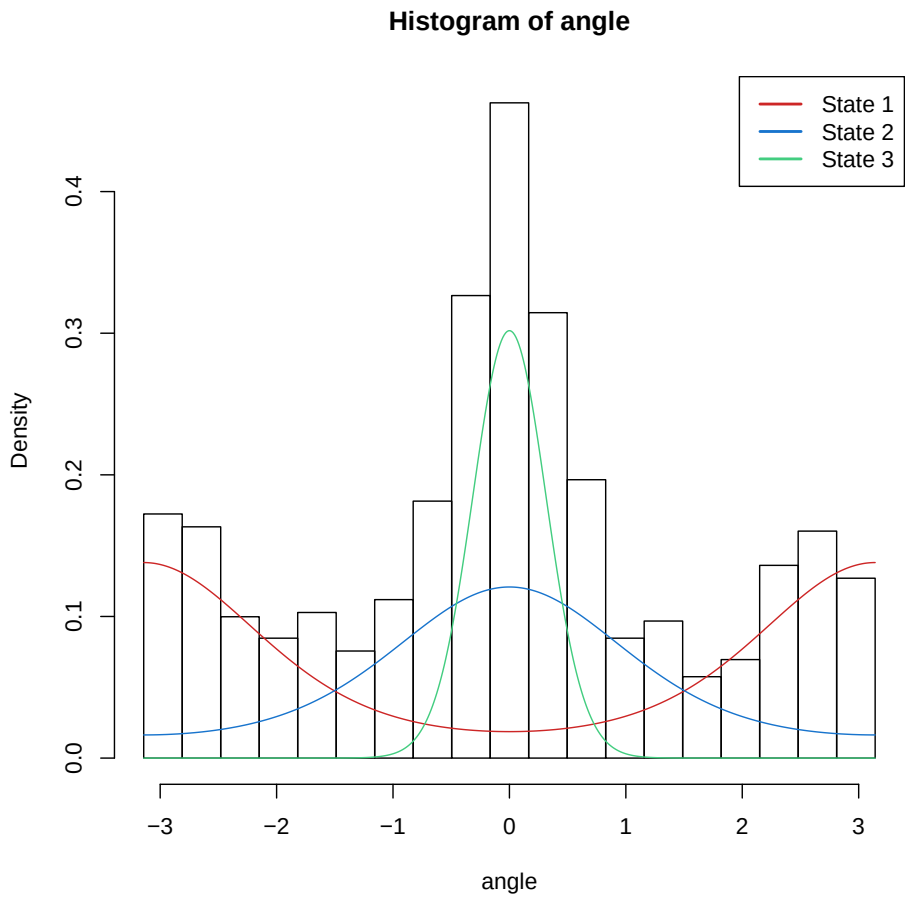
```
angle[which(angle>pi)] <- angle[which(angle>pi)]-2*pi
```

We can plot histograms of the observations, as well as the gamma and von Mises density functions used to simulate values.

```
# histogram of steps
hist(step,probability=TRUE)
# lines of state-dependent distributions
# (weighted by stationary distribution)
grid <- seq(min(step),max(step),length=500)
station <- solve(t(diag(3)-Gamma+1),rep(1,3))
points(grid,dgamma(grid,stepShape[1],stepRate[1])*station[1],
       type="l",col="firebrick3")
points(grid,dgamma(grid,stepShape[2],stepRate[2])*station[2],
       type="l",col="dodgerblue3")
points(grid,dgamma(grid,stepShape[3],stepRate[3])*station[3],
       type="l",col="seagreen3")
legend("topright",col=c("firebrick3","dodgerblue3","seagreen3"),
       legend=c("State 1","State 2","State 3"),lwd=2)
```

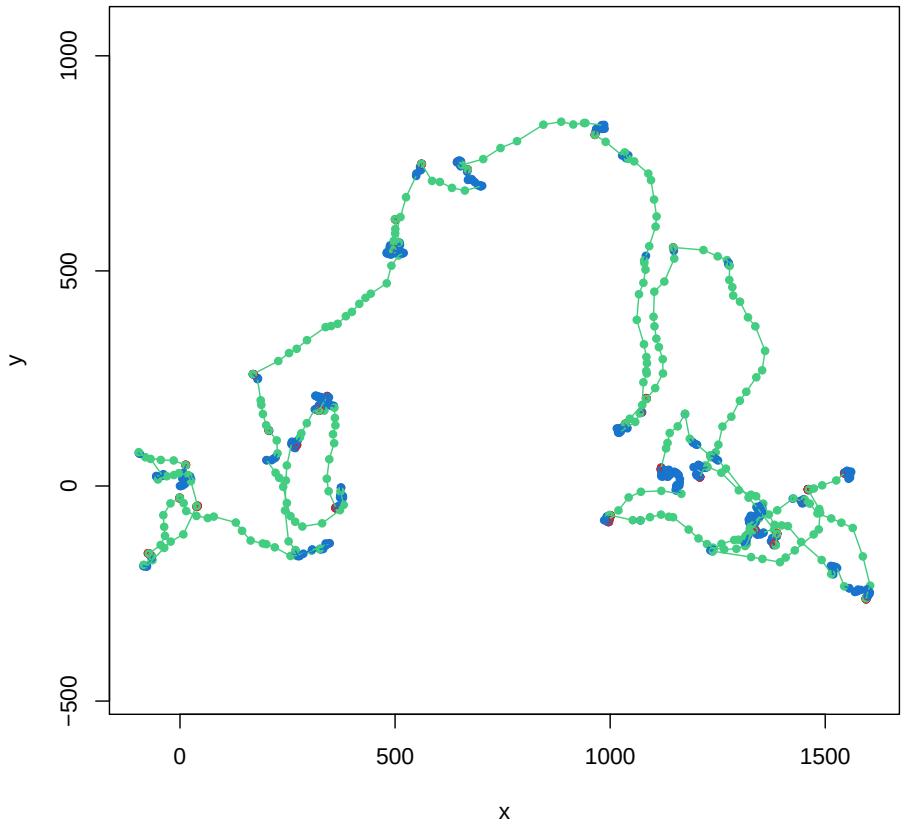



```
# histogram of angles
hist(angle,probability=TRUE,breaks=seq(-pi,pi,length=20))
# lines of state-dependent distributions
# (weighted by stationary distribution)
grid <- seq(-pi,pi,length=500)
points(grid,dvm(grid,angleMean[1],angleCon[1])*station[1],
       type="l",col="firebrick3")
points(grid,dvm(grid,angleMean[2],angleCon[2])*station[2],
       type="l",col="dodgerblue3")
points(grid,dvm(grid,angleMean[3],angleCon[3])*station[3],
       type="l",col="seagreen3")
legend("topright",col=c("firebrick3","dodgerblue3","seagreen3"),
       legend=c("State 1","State 2","State 3"),lwd=2)
```



It is possible to transform the series of steps and angles to the simulated locations of the animal, given the initial location – which we choose arbitrarily as $(0, 0)$ – e.g. to plot the simulated track.

```
# matrix of locations (one row = one x/y location)
X <- matrix(NA,nrow=nbObs,ncol=2)
X[1,] <- c(0,0) # arbitrary value for initial location

phi <- 0 # compass direction

for(t in 2:nbObs) {
  # 2-dimensional step taken at time 't'
  m <- step[t-1]*c(Re(exp(1i*phi)),Im(exp(1i*phi)))

  X[t,] <- X[t-1,] + m # update location
  phi <- phi + angle[t-1] # update compass direction
}

# plot track, colored by states
plot(X[,1],X[,2],xlab="x",ylab="y",pch=19,cex=0.7,asp=1,
        col=c("firebrick3","dodgerblue3","seagreen3")[S2])
segments(X[-nrow(X),1],X[-nrow(X),2],X[-1,1],X[-1,2],
        col=c("firebrick3","dodgerblue3","seagreen3")[S2[-nrow(X)]])
```



3.2 Implement the likelihood function

Using the same ideas as before, we implement the log-likelihood function of the 3-state movement HMM. Note that we compute the joint probability of the step and angle as the product of the marginal probabilities, because the step and angle are assumed to be independent, conditional on the state.

```
logLikMove <- function(step,angle,delta,Gamma,stepShape,stepRate,
        angleMean,angleCon)
{
  nbObs <- length(step)

  # probabilities of observations conditional on state
  allProbs <- matrix(1,nrow=nbObs,ncol=3)
  for(obs in 1:nbObs) {
    for(state in 1:3) {
      stepProb <- dgamma(step[obs],stepShape[state],stepRate[state])
      angleProb <- dvm(angle[obs],angleMean[state],angleCon[state])
      # joint probability = step probability * angle probability
      allProbs[obs,state] <- stepProb*angleProb
    }
  }
```

```
  # forward algorithm
  v <- delta*allProbs[1,]
  llk <- 0
  for(t in 2:nbObs) {
    v <- v%*%Gamma*allProbs[t,]
    llk <- llk+log(sum(v))
    v <- v/sum(v)
  }

  return(llk)
}
```

The code for the forward algorithm is exactly identical to what we used for the 2-state Poisson HMM: only the computation of the matrix `allProbs` changes here. We can evaluate the log-likelihood on the simulated movement data:

```
logLikMove(step,angle,delta,Gamma,stepShape,stepRate,angleMean,angleCon)

## [1] -3878.016
```

3.3 Numerical optimization of the likelihood

We want to find the maximum of the (log-)likelihood function, and the values of the parameters of the model for which it is reached. In a model with three states (and more), it is a bit trickier to parametrise the transition probability matrix. It is necessary to ensure that all transition probabilities are in $[0, 1]$, and that each row of the transition probability matrix sums to 1. In a 2-state HMM, we estimate one probability per row, and it is sufficient to specify that it should be constrained to $[0, 1]$ (because $p \in [0, 1] \Rightarrow 1 - p \in [0, 1]$). However, with three states, we estimate two probabilities per row, and constraining them to $[0, 1]$ is not sufficient. In the following, we first estimate unconstrained parameters on $[0, \infty)$, and then satisfy the constraints by dividing each row of the transition probability matrix by the sum of its elements. We apply the same idea to estimate the initial distribution of the model.

First, we write the negative log-likelihood function, as a function of the observations `step` and `angle`, and the vector `par` of all the parameters.

```
nLogLikMove <- function(step,angle,par)
{
  nbObs <- length(step)

  # unpack parameters
  stepShape <- par[1:3]
  stepRate <- par[4:6]
  angleMean <- par[7:9]
  angleCon <- par[10:12]

  Gamma <- diag(3) # diagonal of ones
  Gamma[!Gamma] <- par[13:18] # fill non-diagonal entries
  Gamma <- Gamma/rowSums(Gamma) # divide by row sums
```

```
  delta <- c(par[19],par[20],1)
  delta <- delta/sum(delta)

  # probabilities of observations conditional on state
  allProbs <- matrix(1,nrow=nbObs,ncol=3)

  # vectorized instead of iterating over observations, for speed
  for(state in 1:3) {
    stepProb <- dgamma(step,stepShape[state],stepRate[state])
    angleProb <- dvm(angle,angleMean[state],angleCon[state])
    # joint probability = step probability * angle probability
    allProbs[,state] <- stepProb*angleProb
  }

  # forward algorithm
  v <- delta*allProbs[1,]
  llk <- 0
  for(t in 2:nbObs) {
    v <- v%*%Gamma*allProbs[t,]
    llk <- llk+log(sum(v))
    v <- v/sum(v)
  }

  return(-llk)
}
```

We check that the function returns the negative log-likelihood, with the simulated data. We need to transform the transition probabilities and initial probabilities, to be positive valued, before passing them to the function (because of the constraints mentioned above).

```
wGamma <- Gamma/diag(Gamma)
wDelta <- delta/delta[3]
par <- c(stepShape,stepRate,angleMean,angleCon,wGamma[!diag(3)],wDelta[1:2])
nLogLikMove(step,angle,par)

## [1] 3878.016
```

We select initial parameters, and fit the model. Note that we transform the transition probabilities and initial distribution to be positive valued.

```
# initial parameters
stepMean0 <- c(1,10,25)
stepSD0 <- c(1,5,15)
stepShape0 <- stepMean0^2/stepSD0^2
stepRate0 <- stepMean0/stepSD0^2
angleMean0 <- c(3,0,0)
angleCon0 <- c(0.5,1.5,5)
Gamma0 <- matrix(c(0.8,0.1,0.1,
                   0.1,0.8,0.1,
```

```
                    0.1,0.1,0.8),ncol=3)

# transform Gamma0 and delta0 to working scale
wGamma0 <- Gamma0/diag(Gamma)
wGamma0 <- Gamma0[!diag(3)]
delta0 <- c(1,1,1)/3
wDelta0 <- delta0[-3]/delta0[3]

par0 <- c(stepShape0,stepRate0,angleMean0,angleCon0,wGamma0,wDelta0)

model <- nlminb(start=par0,objective=nLogLikMove,step=step,angle=angle,
                lower=c(rep(0,6),rep(-pi,3),rep(0,11)),
                upper=c(rep(Inf,6),rep(pi,3),rep(Inf,11)))
```

Finally, we can unpack the parameter estimates, to compare them with the parameters used in the simulation.

```
stepShapeMLE <- model$par[1:3]
stepRateMLE <- model$par[4:6]
stepMeanMLE <- stepShapeMLE/stepRateMLE
stepSDMLE <- sqrt(stepShapeMLE)/stepRateMLE
angleMeanMLE <- model$par[7:9]
angleConMLE <- model$par[10:12]

GammaMLE2 <- diag(3)
GammaMLE2[!GammaMLE2] <- model$par[13:18]
GammaMLE2 <- GammaMLE2/apply(GammaMLE2,1,sum)

deltaMLE2 <- c(model$par[19],model$par[20],1)
deltaMLE2 <- deltaMLE2/sum(deltaMLE2)

stepMeanMLE
## [1]  0.5527997  4.8875161 30.3121434
stepSDMLE
## [1]  0.553539  3.729172 16.843165
angleMeanMLE
## [1]  3.10009174 -0.11538832 -0.01762986
angleConMLE
## [1] 1.0591711 0.9790836 8.5825098
GammaMLE2
##             [,1]       [,2]       [,3]
## [1,] 0.89371636 0.04041003 0.06587361
## [2,] 0.07065495 0.85895291 0.07039214
## [3,] 0.05761083 0.12409390 0.81829527
deltaMLE2
## [1] 0.000000e+00 9.999983e-01 1.653669e-06
```

4 Extensions

4.1 Covariates

4.1.1 Method

In Equation 1, the transition probability matrix was taken to be constant in time. However, it is often of interest to model the state transition probabilities as functions of time-varying covariates. For example, in movement ecology, this can be used to answer questions such as: "what influence does this environmental covariate have on the animal's activity?". This can be done by assuming the Markov chain to be time-varying, with transition probability matrix

$$\mathbf{\Gamma}^{(t)} = \begin{pmatrix} \gamma_{11}^{(t)} & \cdots & \gamma_{1N}^{(t)} \\ \vdots & \ddots & \vdots \\ \gamma_{N1}^{(t)} & \cdots & \gamma_{NN}^{(t)} \end{pmatrix}$$

The likelihood of the model then becomes

$$\mathcal{L}_T = \boldsymbol{\delta} \boldsymbol{P}(\boldsymbol{z}_1) \mathbf{\Gamma}^{(2)} \boldsymbol{P}(\boldsymbol{z}_2) \cdots \mathbf{\Gamma}^{(T)} \boldsymbol{P}(\boldsymbol{z}_T) \mathbf{1}'$$

The transition probabilities can be linked to the covariate(s) via the multinomial logit link. In the general case of N states,

$$\gamma_{ij}^{(t)} = \Pr\big(S_t = j | S_{t-1} = i\big) = \frac{\exp(\eta_{ij})}{\sum_{k=1}^{N} \exp(\eta_{ik})}, \tag{3}$$

where $i, j = 1, \ldots, N$, and

$$\eta_{ij} = \begin{cases} \beta_0^{(ij)} + \sum_{l=1}^{p} \beta_l^{(ij)} w_{lt} & \text{if } i \neq j, \\ 0 & \text{otherwise,} \end{cases}$$

Here, w_{lt} is the l-th covariate at time t, and p is the number of covariates considered. Note that in the case of a 2-state HMM, the expression of the transition probability matrix can be rewritten

$$\mathbf{\Gamma}^{(t)} = \begin{pmatrix} 1 - \text{logit}^{-1}(\eta_{12}) & \text{logit}^{-1}(\eta_{12}) \\ \text{logit}^{-1}(\eta_{21}) & 1 - \text{logit}^{-1}(\eta_{21}) \end{pmatrix}$$

Here we describe the inclusion of covariates to the general case of a N-state HMM, but the implementation can be simplified for a 2-state HMM, using the expression above.

4.1.2 Implementation

If covariates are included in the model, the parameters to estimate are the regression coefficients $\beta_l^{(ij)}$, rather than the transition probabilities directly. Then, the transition probability matrix needs to be computed at each time step t of the forward algorithm.

We store the coefficients β for the off-diagonal transition probabilities in a $(p+1) \times (N \cdot (N-1))$ matrix. For example, for a 3-state HMM with two covariates,

$$\boldsymbol{\beta} = \begin{pmatrix} \beta_0^{(12)} & \beta_0^{(13)} & \beta_0^{(21)} & \beta_0^{(23)} & \beta_0^{(31)} & \beta_0^{(32)} \\ \beta_1^{(12)} & \beta_1^{(13)} & \beta_1^{(21)} & \beta_1^{(23)} & \beta_1^{(31)} & \beta_1^{(32)} \\ \beta_2^{(12)} & \beta_2^{(13)} & \beta_2^{(21)} & \beta_2^{(23)} & \beta_2^{(31)} & \beta_2^{(32)} \end{pmatrix}$$

Here the first row corresponds to the intercept terms and the other two rows to the slope coefficients associated with the two covariates. There are as many columns as there are off-diagonal entries in the 3×3 transition probability matrix, and that matrix is filled row-wise (i.e. column 1 in $\boldsymbol{\beta}$ is linked to $\gamma_{12}^{(t)}$, column 2 is linked to $\gamma_{13}^{(t)}$, column 3 is linked to $\gamma_{21}^{(t)}$, etc.).

We store the covariate values in a design matrix with $p+1$ columns: one for the intercept, and one for each covariate.

$$\boldsymbol{W} = \begin{pmatrix} 1 & w_{11} & w_{21} & \cdots & w_{p1} \\ 1 & w_{12} & w_{22} & \cdots & w_{p2} \\ \vdots & \vdots & \vdots & \vdots & \\ 1 & w_{1T} & w_{2T} & \cdots & w_{pT} \end{pmatrix}$$

Each row of the design matrix $\boldsymbol{W}$ corresponds to one time index . At each instant, the $N \cdot (N-1)$ predictors η_{ij} are obtained by multiplying one row of $\boldsymbol{W}$ by $\boldsymbol{\beta}$:

$$\begin{pmatrix} 1 & w_{1t} & w_{2t} & \cdots & w_{pt} \end{pmatrix} \cdot \boldsymbol{\beta} = \begin{pmatrix} \eta_{12} & \eta_{13} & \cdots & \eta_{N,N-1} \end{pmatrix}$$

Then, we can compute the transition probabilities as in Equation 3. Thus, in each iteration t of the forward algorithm, within the likelihood function, a few lines of code should be added to compute $\boldsymbol{\Gamma}^{(t)}$. In the code below, we call `covs` the design matrix $\boldsymbol{W}$, and `beta` the matrix $\boldsymbol{\beta}$ of coefficients.

```
v <- delta*allProbs[1,]
llk <- 0

for(t in 2:nbObs) {
  # matrix with ones on diagonal
  Gamma <- diag(nbStates)
  # fill non-diagonal elements with exp of linear predictors ("eta"s)
  Gamma[!Gamma] <- exp(covs[t,]%*%beta)
  # transpose, because R fills matrices column-wise
  Gamma <- t(Gamma)
  # normalize rows to sum to one
  Gamma <- Gamma/rowSums(Gamma)

  v <- v%*%Gamma*allProbs[t,]
  llk <- llk+log(sum(v))
  v <- v/sum(v)
}
```

4.2 More to come...

HMMs have many more conceptual extensions, as described by Zucchini *et al.* (2016), so we might add to this section later. If you would be interested in a particular extension, let us now!

5 Inference and model assessment

5.1 State decoding with the Viterbi algorithm

The Viterbi algorithm (Zucchini *et al.*, 2016, Chapter 5) is used to compute the sequence of states most likely to have given rise to the observations. It is referred to as global decoding, and consists in a recursive scheme to maximize over all possible state sequences.

5.1.1 Poisson HMM

The following function implements the Viterbi algorithm for the 2-state Poisson HMM. It takes the data and values of the model parameters as inputs.

```
viterbiPois <- function(Z,rate,Gamma,delta)
{
    nbObs <- length(Z)

    # probabilities of observations conditional on state
    allProbs <- matrix(1,nrow=nbObs,ncol=2)
    allProbs[,1] <- dpois(Z,rate[1])
    allProbs[,2] <- dpois(Z,rate[2])

    xi  <-  matrix(0,nrow=nbObs,ncol=2)
    v <- delta*allProbs[1,]
    xi[1,] <- v/sum(v)

    for (t in 2:nbObs) {
        v <- apply(xi[t-1,]*Gamma,2,max)*allProbs[t,]
        xi[t,] <- v/sum(v)
    }

    # most probable state sequence
    stSeq <- rep(NA,nbObs)
    stSeq[nbObs] <- which.max(xi[nbObs,])

    for (t in (nbObs-1):1)
        stSeq[t] <- which.max(Gamma[,stSeq[t+1]]*xi[t,])

    return(stSeq)
}
```

We can now decode the most probable state sequence, for the model fitted to the simulated data, and compare with the true (simulated) state sequence.

```
# Z are the simulated observations, and the other arguments are the maximum
# likelihood estimates of the parameters of the model
states <- viterbiPois(Z,rateMLE,GammaMLE,deltaMLE)

# Proportion of decoded states identical to true (simulated) states
length(which(states==S1))/length(states)
```

```
## [1] 0.977
```

The states were correctly decoded for 97.7% of the observations.

5.1.2 Movement HMM

Only the computation of $\boldsymbol{P}(\boldsymbol{z}_t)$ needs to be changed, to adapt the function to the 3-state movement HMM.

```
viterbiMove <- function(step,angle,stepShape,stepRate,angleMean,angleCon,
                        Gamma,delta)
{
    nbObs <- length(step)

    # probabilities of observations conditional on state
    allProbs <- matrix(1,nrow=nbObs,ncol=3)
    for(state in 1:3) {
        stepProb <- dgamma(step,stepShape[state],stepRate[state])
        angleProb <- dvm(angle,angleMean[state],angleCon[state])
        # joint probability = step probability * angle probability
        allProbs[,state] <- stepProb*angleProb
    }

    xi  <-  matrix(0,nrow=nbObs,ncol=3)
    v <- delta*allProbs[1,]
    xi[1,] <- v/sum(v)

    for (t in 2:nbObs) {
        v <- apply(xi[t-1,]*Gamma,2,max)*allProbs[t,]
        xi[t,] <- v/sum(v)
    }

    # most probable state sequence
    stSeq <- rep(NA,nbObs)
    stSeq[nbObs] <- which.max(xi[nbObs,])

    for (t in (nbObs-1):1)
        stSeq[t] <- which.max(Gamma[,stSeq[t+1]]*xi[t,])

    return(stSeq)
}
```

5.2 State probabilities

We call state probabilities the probabilities of being in the different states at each time point (Zucchini *et al.*, 2016, Chapter 5). This approach is referred to as local decoding.

5.2.1 Poisson HMM

The computation of state probabilities is based on the so-called forward probabilities and backward probabilities. The forward probabilities $\boldsymbol{\alpha}_t$ are the quantities recursively computed

in the forward algorithm (Equation 2). The j-th component of $\boldsymbol{\alpha}_t$ is

$$\alpha_t(j) = \Pr(\boldsymbol{Z}_1 = \boldsymbol{z}_1, \boldsymbol{Z}_2 = \boldsymbol{z}_2, \ldots, \boldsymbol{Z}_t = \boldsymbol{z}_t, S_t = j)$$

To avoid numerical issues, we compute the forward log-probabilities, using the same scaling trick as before.

```
logAlphaPois <- function(Z, rate, Gamma, delta)
{
    nbObs <- length(Z)
    lalpha <- matrix(NA,nbObs,2) # ncol = number of states

    # probabilities of observations conditional on state
    allProbs <- matrix(1,nrow=nbObs,ncol=2)
    allProbs[,1] <- dpois(Z,rate[1])
    allProbs[,2] <- dpois(Z,rate[2])

    lscale <- 0
    v <- delta*allProbs[1,]
    lalpha[1,] <- log(v)

    for(t in 2:nbObs) {
        v <- v%*%Gamma*allProbs[t,]
        lscale <- lscale + log(sum(v))
        v <- v/sum(v)
        lalpha[t,] <- log(v) + lscale
    }

    return(lalpha)
}
```

The j-th component of the backward probability $\boldsymbol{\beta}_t$ is the conditional probability

$$\beta_t(j) = \Pr(\boldsymbol{Z}_{t+1} = \boldsymbol{z}_{t+1}, \boldsymbol{Z}_{t+2} = \boldsymbol{z}_{t+2}, \ldots, \boldsymbol{Z}_T = \boldsymbol{z}_T | S_t = j)$$

```
logBetaPois <- function(Z, rate, Gamma)
{
    nbObs <- length(Z)
    nbStates <- 2 # number of states
    lbeta <- matrix(NA,nbObs,nbStates)

    # probabilities of observations conditional on state
    allProbs <- matrix(1,nrow=nbObs,ncol=2)
    allProbs[,1] <- dpois(Z,rate[1])
    allProbs[,2] <- dpois(Z,rate[2])

    lscale <- log(nbStates)
    v <- rep(1,nbStates)/nbStates
    lbeta[nbObs,] <- 0
```

```
    for(t in (nbObs-1):1) {
        v <- Gamma%*%(allProbs[t+1,]*v)
        lbeta[t,] <- log(v) + lscale
        sumv <- sum(v)
        v <- v/sumv
        lscale <- lscale + log(sumv)
    }

    return(lbeta)
}
```

Finally, to get the state probabilities, we use that

$$\alpha_t(j)\beta_t(j) = \Pr(\boldsymbol{Z}_1 = \boldsymbol{z}_1, \ldots, \boldsymbol{Z}_T = \boldsymbol{z}_T, S_t = j)$$

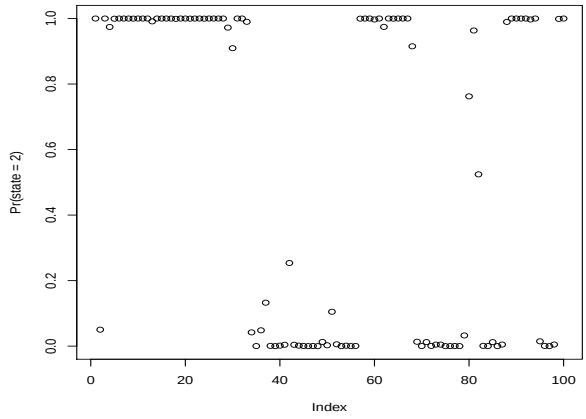
```
stateProbsPois <- function(Z, rate, Gamma, delta)
{
    nbObs <- length(Z)

    la <- logAlphaPois(Z, rate, Gamma, delta) # forward log-probabilities
    lb <- logBetaPois(Z, rate, Gamma) # backward log-probabilities

    c <- max(la[nbObs,]) # cancels out below; prevents numerical errors
    llk <- c + log(sum(exp(la[nbObs,]-c)))

    stateProbs <- matrix(NA,nbObs,2) # ncol = number of states
    for(t in 1:nbObs)
        stateProbs[t,] <- exp(la[t,]+lb[t,]-llk)

    return(stateProbs)
}
```

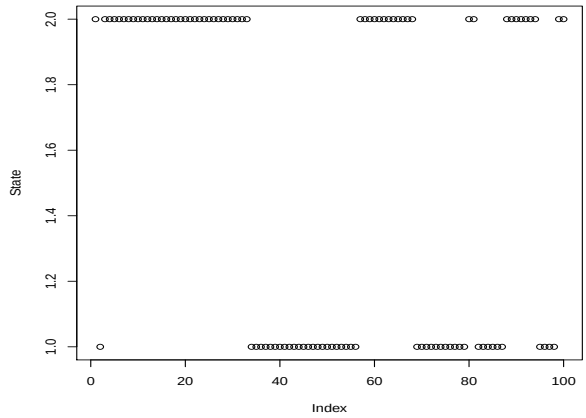
This returns a matrix with one row for each observation, and one column for each state. The element on row i and column j is the probability of being in state j at time i.

```
sp <- stateProbsPois(Z, rateMLE, GammaMLE, deltaMLE)
```

We can plot the state probabilities and the true sequence of (simulated) states, to check that they are consistent.

```
plot(sp[1:100,2], ylab="Pr(state = 2)")
plot(S1[1:100], ylab="State")
```





5.2.2 Movement HMM

The only change needed is to adapt the computation of the matrix `allProbs` in the functions for the forward and backward probabilities.

```
logAlphaMove <- function(step,angle,stepShape,stepRate,angleMean,angleCon,
                         Gamma,delta)
{
    nbObs <- length(step)
    lalpha <- matrix(NA,nbObs,3)

    # probabilities of observations conditional on state
    allProbs <- matrix(1,nrow=nbObs,ncol=3)
    for(state in 1:3) {
        stepProb <- dgamma(step,stepShape[state],stepRate[state])
        angleProb <- dvm(angle,angleMean[state],angleCon[state])
        # joint probability = step probability * angle probability
        allProbs[,state] <- stepProb*angleProb
    }

    lscale <- 0
    v <- delta*allProbs[1,]
```

```
    lalpha[1,] <- log(v)

    for(t in 2:nbObs) {
        v <- v%*%Gamma*allProbs[t,]
        lscale <- lscale + log(sum(v))
        v <- v/sum(v)
        lalpha[t,] <- log(v) + lscale
    }

    return(lalpha)
}

logBetaMove <- function(step,angle,stepShape,stepRate,angleMean,angleCon,Gamma)
{
    nbObs <- length(step)
    nbStates <- 3 # number of states
    lbeta <- matrix(NA,nbObs,nbStates)

    # probabilities of observations conditional on state
    allProbs <- matrix(1,nrow=nbObs,ncol=3)
    for(state in 1:3) {
        stepProb <- dgamma(step,stepShape[state],stepRate[state])
        angleProb <- dvm(angle,angleMean[state],angleCon[state])
        # joint probability = step probability * angle probability
        allProbs[,state] <- stepProb*angleProb
    }

    lscale <- log(nbStates)
    v <- rep(1,nbStates)/nbStates
    lbeta[nbObs,] <- 0

    for(t in (nbObs-1):1) {
        v <- Gamma%*%(allProbs[t+1,]*v)
        lbeta[t,] <- log(v) + lscale
        sumv <- sum(v)
        v <- v/sumv
        lscale <- lscale + log(sumv)
    }

    return(lbeta)
}

stateProbsMove <- function(step,angle,stepShape,stepRate,angleMean,angleCon,
                           Gamma,delta)
{
    nbObs <- length(step)

    # forward log-probabilities
    la <- logAlphaMove(step,angle,stepShape,stepRate,angleMean,angleCon,Gamma,
```

```
                        delta)
    # backward log-probabilities
    lb <- logBetaMove(step,angle,stepShape,stepRate,angleMean,angleCon,Gamma)

    c <- max(la[nbObs,]) # cancels out below; prevents numerical errors
    llk <- c + log(sum(exp(la[nbObs,]-c)))

    stateProbs <- matrix(NA,nbObs,3) # ncol = number of states
    for(i in 1:nbObs)
        stateProbs[i,] <- exp(la[i,]+lb[i,]-llk)

    return(stateProbs)
}
```

5.3 Pseudo-residuals

Formal assessment of an HMM is done using so-called normal pseudo-residuals, described in Chapter 6 of Zucchini *et al.* (2016). The pseudo-residuals are computed in two steps. First, the uniform pseudo-residuals u_t are computed as

$$u_t = \Pr(Z_t \leq z_t)$$

If the model fits the data well, the u_t are uniformly distributed on $[0, 1]$. Then, the normal pseudo-residuals are given by

$$r_t = \Phi^{-1}(u_t)$$

where Φ is the standard normal cumulative distribution function. If the fit is good, the normal pseudo-residuals are standard normally distributed.

5.3.1 Poisson HMM

The definition of pseudo-residuals given above applies to continuous distributions only. In the Poisson HMM, we use a discrete Poisson distribution, and thus need to extend that definition.

For discrete distribution, the uniform pseudo-residuals are defined as segments:

$$[u_t^-, u_t^+] = [\Pr(Z_t \leq z_t^-), \Pr(Z_t \leq z_t)]$$

where z_t^- is the largest possible value of the distribution which is strictly less than z_t. The normal pseudo-residuals are then defined as

$$[r_t^-, r_t^+] = [\Phi(u_t^-), \Phi(u_t^+)]$$

This section will be completed soon!

5.3.2 Movement HMM

In the movement data, the observations are bivariate: we have one step length and one turning angle for each time point. Thus, we need to compute two series of pseudo-residuals, one for the steps and one for the angles.

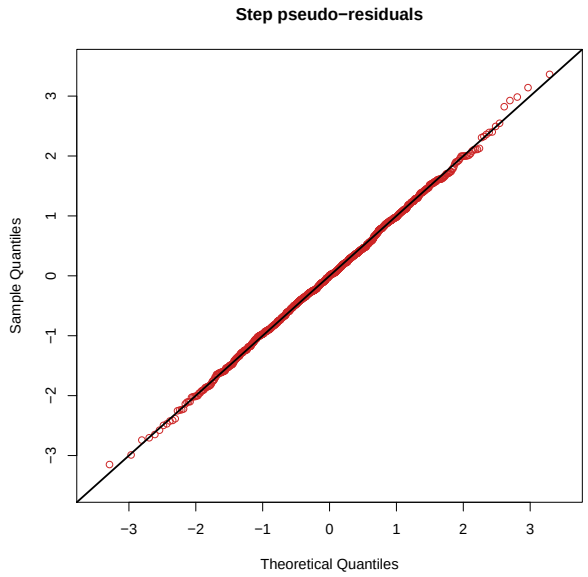
```r
pseudoResMove <- function(step,angle,stepShape,stepRate,angleMean,angleCon,
                          Gamma,delta)
{
    nbObs <- length(step)
    nbStates <- 3

    pStepMat <- matrix(NA,nbObs,nbStates)
    pAngleMat <- matrix(NA,nbObs,nbStates)
    stepRes <- rep(NA,nbObs)
    angleRes <- rep(NA,nbObs)

    la <- logAlphaMove(step,angle,stepShape,stepRate,angleMean,angleCon,
                       Gamma,delta)

    for(state in 1:nbStates) {
        for(t in 1:nbObs) {
            # integrate step density function
            pStepMat[t,state] <- pgamma(step[t],stepShape[state],stepRate[state])
            # integrate angle density function
            pAngleMat[t,state] <- integrate(dvm, lower=-pi, upper=angle[t],
                                            angleMean[state],
                                            angleCon[state])$value
        }
    }

    stepRes[1] <- qnorm(delta%*%pStepMat[1,])
    angleRes[1] <- qnorm(delta%*%pAngleMat[1,])

    for(t in 2:nbObs) {
        c <- max(la[t-1,]) # cancels out below; prevents numerical errors
        a <- exp(la[t-1,]-c)

        stepRes[t] <- qnorm(t(a)%*%(Gamma/sum(a))%*%pStepMat[t,])
        angleRes[t] <- qnorm(t(a)%*%(Gamma/sum(a))%*%pAngleMat[t,])
    }

    return(list(stepRes=stepRes,angleRes=angleRes))
}
```

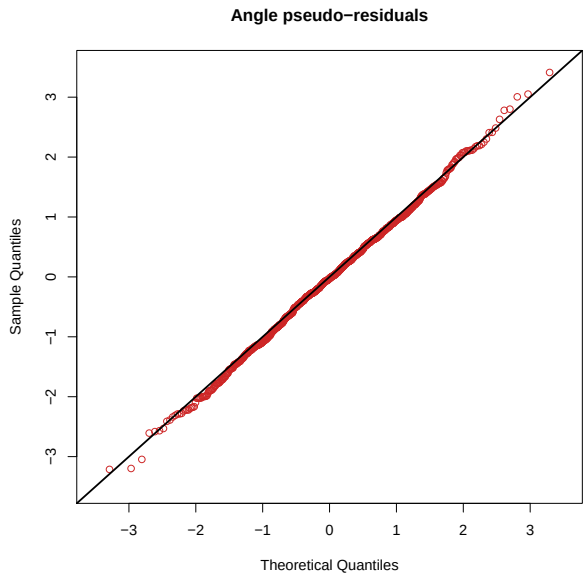
Note that, instead of using `pvm` to compute the uniform pseudo-residuals for the angles, we apply the function `integrate` to the density `dvm`. This is because `pvm` works on angle values ranging from 0 to 2π, whereas we work with values on $[-\pi, \pi]$. Using `integrate` allows us to specify the lower bound of the interval to integrate on.

Then, we plot the standard normal qq-plots of the residuals for the steps and angles, to check their normality. As could be expected for simulated data, the model seems to fit well.

```r
pr <- pseudoResMove(step,angle,stepShapeMLE,stepRateMLE,angleMeanMLE,
                    angleConMLE,GammaMLE2,deltaMLE2)

qqnorm(pr$stepRes, xlim=c(-3.5,3.5), ylim=c(-3.5,3.5), col="firebrick3",
```

```
       main="Step pseudo-residuals")
abline(0,1,lwd=2)
qqnorm(pr$angleRes, xlim=c(-3.5,3.5), ylim=c(-3.5,3.5), col="firebrick3",
       main="Angle pseudo-residuals")
abline(0,1,lwd=2)
```





6 Speed up with Rcpp

The R package Rcpp (Eddelbuettel and Francois, 2011) makes it simple to call C++ functions from R, which can significantly improve the computational performance. It is our experience that fitting an HMM is usually between five and ten times quicker when the likelihood function is implemented in C++ (compared with pure R code).

We do not intend to give a comprehensive tutorial on the use of Rcpp, but only to provide simple examples, based on the models described in the previous sections.

In each example, we provide C++ code, and the R code that should be executed. We make use of RcppArmadillo (Eddelbuettel and Sanderson, 2014), a linear algebra library, to speed up matrices computations and take advantage of the friendly syntax. We want to point out that, with the help of Rcpp and RcppArmadillo, writing C++ code does not require an advanced knowledge of the language, for someone who is familiar with R.

6.1 Poisson HMM

We write the core of the log-likelihood in C++, to speed up each evaluation of the function, and thus its optimization. The line preceding the function (`// [[Rcpp::export]]`) indicates that an interface will automatically be generated to call it from R.

```
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]

// [[Rcpp::export]]
double nllk_rcpp(arma::vec Z, arma::vec rate, arma::mat Gamma,
                 arma::rowvec delta)
{
  int nbObs = Z.size();

  // probabilities of observations conditional on state
  arma::mat allProbs(nbObs,2);
  for(int t=0 ; t<nbObs ; t++) {
    allProbs(t,0) = R::dpois(Z(t),rate(0),0);
    allProbs(t,1) = R::dpois(Z(t),rate(1),0);
  }

  arma::rowvec v = delta%allProbs.row(0);
  double llk = 0;

  for (int t=1 ; t<nbObs ; t++) {
    v = v*Gamma%allProbs.row(t);
    llk = llk + log(sum(v));
    v = v/sum(v);
  }

  return llk;
}
```

Then, we source the C++ file from an R session, and can use the function `nllk_pois_rcpp` from R. Here, we decide to unpack the parameters in the R code, for convenience, but the whole function could be written in C++, and directly provided to `nlminb` for the optimization.

```
library(Rcpp)
sourceCpp("code/poissonHMM.cpp")

nLogLikPois_rcpp <- function(Z,par)
{
```

```
  # unpack parameters
  rate <- par[1:2]
  Gamma <- matrix(NA,nrow=2,ncol=2)
  Gamma[,1] <- par[3:4]
  Gamma[,2] <- 1-Gamma[,1]
  delta <- c(par[5],1-par[5])

  # nllk_pois_rcpp is defined in the C++ source file
  llk <- nllk_pois_rcpp(Z,rate,Gamma,delta)

  return(-llk)
}
```

Let's make sure that our new function returns the same result as the pure R negative log-likelihood function:

```
par <- c(rate,Gamma[,1],delta[1])
nLogLikPois_rcpp(Z,par)

## [1] 2905.149
```

We use the exact same code as before to call `nlminb` on the function:

```
# initial parameters
rate0 <- c(3,10)
Gamma0 <- matrix(c(0.9,0.1,0.1,0.9),ncol=2)
delta0 <- c(0.5,0.5)
par0 <- c(rate0,Gamma0[,1],delta0[1])

# fit model
model <- nlminb(start=par0,objective=nLogLikPois_rcpp,Z=Z,
                lower=c(0,0,0,0,0),upper=c(Inf,Inf,1,1,1))
```

The estimates are the same as the ones found with the R code:

```
rateMLE <- model$par[1:2]
GammaMLE <- matrix(c(model$par[3:4],1-model$par[3:4]),ncol=2)
deltaMLE <- c(model$par[5],1-model$par[5])

rateMLE

## [1]  4.931807 14.770212

GammaMLE

##           [,1]      [,2]
## [1,] 0.7839544 0.2160456
## [2,] 0.1012449 0.8987551

deltaMLE

## [1] 0 1
```

6.2 Movement HMM

Once again, we write the main part of the log-likelihood function in C++. We also define the function `dvm_rcpp`, the density function of the von Mises distribution, which we need but is not included in Rcpp (we use its definition, based on the modified Bessel function of order 0), and `dgamma_rcpp`, a vectorized version of `R::dgamma`.

```
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]

#include <time.h>

// gamma density function
arma::colvec dgamma_rcpp(arma::vec x, double shape, double rate)
{
  arma::colvec res(x.size());

  for(int i=0;i<x.size();i++)
    res(i) = R::dgamma(x(i),shape,1/rate,0);

  return res;
}

// von Mises density function
arma::colvec dvm_rcpp(arma::vec x, double mu, double kappa)
{
  arma::colvec res(x.size());
  double b = R::bessel_i(kappa,0,2);

  for(int i=0;i<x.size();i++)
    res(i) = 1/(2*M_PI*b)*pow((exp(cos(x(i)-mu)-1)),kappa);

  return res;
}

// [[Rcpp::export]]
double nllk_mov_rcpp(arma::vec step, arma::vec angle,arma::vec stepShape,
                     arma::vec stepRate, arma::vec angleMean,arma::vec angleCon,
                     arma::mat Gamma, arma::rowvec delta)
{
  int nbObs = step.size();

  // probabilities of observations conditional on state
  arma::colvec stepProb;
  arma::colvec angleProb;
  arma::mat allProbs(nbObs,3);
  for(int state=0 ; state<3 ; state++) {
    stepProb = dgamma_rcpp(step,stepShape(state),stepRate(state));
    angleProb = dvm_rcpp(angle,angleMean(state),angleCon(state));
    // joint probability = step probability * angle probability
```

```
    allProbs.col(state) = stepProb%angleProb;
  }

  // forward algorithm
  arma::rowvec v = delta%allProbs.row(0);
  double llk = 0;
  for(int t=1 ; t<nbObs ; t++) {
    v = v*Gamma%allProbs.row(t);
    llk = llk+log(sum(v));
    v = v/sum(v);
  }

  return llk;
}
```

In the R function, we only unpack the parameters, and pass them to the C++ function which performs the computation.

```
sourceCpp("code/movementHMM.cpp")

nLogLikMove_rcpp <- function(step,angle,par)
{
  nbObs <- length(step)

  # unpack parameters
  stepShape <- par[1:3]
  stepRate <- par[4:6]
  angleMean <- par[7:9]
  angleCon <- par[10:12]

  Gamma <- diag(3)
  Gamma[!Gamma] <- par[13:18]
  Gamma <- Gamma/apply(Gamma,1,sum)

  delta <- c(par[19],par[20],1)
  delta <- delta/sum(delta)

  # nllk_mov_rcpp is defined in the C++ source file
  llk <- nllk_mov_rcpp(step,angle,stepShape,stepRate,angleMean,
                       angleCon,Gamma,delta)

  return(-llk)
}
```

We check that the function returns the same result as earlier, on the simulated data.

```
wGamma <- Gamma/diag(Gamma)
wDelta <- delta/delta[3]
par <- c(stepShape,stepRate,angleMean,angleCon,wGamma[!diag(3)],wDelta[1:2])
nLogLikMove_rcpp(step,angle,par)
```

```
## [1] 3878.016
```

We fit the model again, using `nlminb`:

```
# initial parameters
stepMean0 <- c(1,10,25)
stepSD0 <- c(1,5,15)
stepShape0 <- stepMean0^2/stepSD0^2
stepRate0 <- stepMean0/stepSD0^2
angleMean0 <- c(3,0,0)
angleCon0 <- c(0.5,1.5,5)
Gamma0 <- matrix(c(0.8,0.1,0.1,
                   0.1,0.8,0.1,
                   0.1,0.1,0.8),ncol=3)

# transform Gamma0 and delta0 to working scale
wGamma0 <- Gamma0/diag(Gamma)
wGamma0 <- Gamma0[!diag(3)]
delta0 <- c(1,1,1)/3
wDelta0 <- delta0[-3]/delta0[3]

par0 <- c(stepShape0,stepRate0,angleMean0,angleCon0,wGamma0,wDelta0)

model_rcpp <- nlminb(start=par0,objective=nLogLikMove_rcpp,step=step,angle=angle,
                lower=c(rep(0,6),rep(-pi,3),rep(0,11)),
                upper=c(rep(Inf,6),rep(pi,3),rep(Inf,11)))
```

We can verify that the estimates are identical to those returned by the pure R code (actually, the last digits can be slightly different, due to numerical approximations).

```
stepShapeMLE <- model_rcpp$par[1:3]
stepRateMLE <- model_rcpp$par[4:6]
stepMeanMLE <- stepShapeMLE/stepRateMLE
stepSDMLE <- sqrt(stepShapeMLE)/stepRateMLE
angleMeanMLE <- model_rcpp$par[7:9]
angleConMLE <- model_rcpp$par[10:12]

GammaMLE2 <- diag(3)
GammaMLE2[!GammaMLE2] <- model_rcpp$par[13:18]
GammaMLE2 <- GammaMLE2/apply(GammaMLE2,1,sum)

deltaMLE2 <- c(model_rcpp$par[19],model_rcpp$par[20],1)
deltaMLE2 <- deltaMLE2/sum(deltaMLE2)

stepMeanMLE

## [1]  0.5527997  4.8875163 30.3121438

stepSDMLE

## [1]  0.553539  3.729172 16.843165
```

```
angleMeanMLE

## [1]  3.10009177 -0.11538838 -0.01762983

angleConMLE

## [1] 1.0591710 0.9790834 8.5825098

GammaMLE2

##             [,1]       [,2]       [,3]
## [1,] 0.89371640 0.04041002 0.06587358
## [2,] 0.07065495 0.85895293 0.07039212
## [3,] 0.05761083 0.12409389 0.81829528

deltaMLE2

## [1] 0.000000e+00 9.999983e-01 1.719918e-06
```

7 Unconstrained optimization

As mentioned earlier, `nlminb` is a constrained optimizer: it can deal with parameters constrained to bounded interval (which we specify with the arguments `lower` and `upper`). There are various other optimizers in R, and some of them are unconstrained, i.e. they can only deal with unbounded parameters – in $(-\infty, +\infty)$. We provide an example of code to perform the minimization of the log-likelihood function, using the unconstrained optimization function `nlm`.

One solution is to use two sets of parameters: the natural parameters (bounded to their natural scale), and the working parameters (unbounded), as described in Section 3.3.1 of Zucchini *et al.* (2016). We define the functions `n2w` and `w2n` to perform the transformation from natural to working parameters, and reciprocally.

```
library(boot) # for logit

# Transform parameters from natural scale (possibly bounded)
# to working scale (unbounded)
n2w <- function(rate,Gamma,delta)
{
  wrate <- log(rate) # from [0,Inf) to (-Inf,Inf)
  wGamma <- logit(Gamma[,1]) # from [0,1] to (-Inf,Inf)
  wdelta <- logit(delta[1]) # from [0,1] to (-Inf,Inf)

  return(c(wrate,wGamma,wdelta))
}

# Transform parameters from working scale (unbounded)
# to natural scale (possibly bounded)
w2n <- function(wpar)
{
```

```
  rate <- exp(wpar[1:2]) # from (-Inf,Inf) to [0,Inf)
  Gamma <- matrix(NA,2,2)
  Gamma[,1] <- inv.logit(wpar[3:4]) # from (-Inf,Inf) to [0,1]
  Gamma[,2] <- 1-Gamma[,1]
  delta <- c(inv.logit(wpar[5]),1-inv.logit(wpar[5])) # from (-Inf,Inf) to [0,1]

  return(list(rate=rate,Gamma=Gamma,delta=delta))
}
```

Then, we modify slightly the negative log-likelihood function, to work on unbounded parameters. The argument `wpar` is the one over which `nlm` optimizes the function, so it contains the working parameters. To compute the likelihood, we transform the parameters to their natural scale with `w2n`.

```
nLogLikPois2 <- function(Z,wpar)
{
  nbObs <- length(Z)

  # transform parameters to natural scale
  par <- w2n(wpar)

  # probabilities of observations conditional on state
  allProbs <- matrix(1,nrow=nbObs,ncol=2)
  allProbs[,1] <- dpois(Z,par$rate[1])
  allProbs[,2] <- dpois(Z,par$rate[2])

  v <- par$delta*allProbs[1,]
  llk <- 0
  for (t in 2:nbObs) {
    v <- v%*%par$Gamma*allProbs[t,]
    llk <- llk+log(sum(v))
    v <- v/sum(v)
  }

  return(-llk)
}
```

Finally, we can call `nlm` of this function, to fit the model. Like before, it is necessary to specify starting values for the parameters of the model. We define them on their natural scale, and transform them to their working scale with `n2w` before providing them to the optimizer. After the estimates have been found, we transform them back to their natural scale with `w2n` before displaying them.

```
# initial parameters
rate0 <- c(3,10)
Gamma0 <- matrix(c(0.9,0.1,0.1,0.9),ncol=2)
delta0 <- c(0.5,0.5)

# transform to working scale
wpar0 <- n2w(rate0,Gamma0,delta0)
```

```r
# fit model
model <- nlm(p=wpar0,f=nLogLikPois2,Z=Z)
par <- w2n(model$estimate)

par

## $rate
## [1]  4.931801 14.770192
##
## $Gamma
##           [,1]      [,2]
## [1,] 0.7839543 0.2160457
## [2,] 0.1012448 0.8987552
##
## $delta
## [1] 6.868472e-08 9.999999e-01
```

References

Eddelbuettel, D., and Francois, R. (2011), "Rcpp: Seamless R and C++ integration". *Journal of Statistical Software*, 40 (8), 1–18.

Eddelbuettel, D., and Sanderson, C. (2014), "RcppArmadillo: Accelerating R with high-performance C++ linear algebra". *Computational Statistics and Data Analysis*, 71, 1054–1063.

Langrock R., King R., Matthiopoulos J., Thomas L., Fortin D., and Morales J.M. (2012), "Flexible and practical modeling of animal telemetry data: hidden Markov models and extensions". *Ecology*, 93 (11), 2336–2342.

Zucchini, W., MacDonald, I.L., and Langrock, R. (2016), *Hidden Markov models for Time Series: An Introduction using R, Second Edition*. Chapman & Hall/CRC Press, Boca Raton, FL.