Tutorial - State-space models

Marie Auger-Méthé

1 State-space models: tutorial goals and set up

The goal of this tutorial is to explore how to fit state-space models (SSMs) to movement data.

There are a few packages available on CRAN that fit SSMs to movement data, but, as far as I know, there is not one package that has a comprehensive list of options. Many additional packages are available on Github or other similar online platform. In addition, you can, as I generally do for my own research, write your own models and fit them with general packages like `TMB` or `rjags`. I would say, for SSMs, you have to be flexible and creative, and choose the package and tool that works for a given circumstance.

We will explore some of the different tools you can use and each of the sections of this tutorial will be associated with a given package.

1.1 Example dataset

But first, let's load the data we will use throughout.

Here we will use one polar bear movement track. This data is Argos data, which is an alternative to GPS that allows to get locations quickly. Argos data is used often on marine animals, because classic GPS technology has difficulties gathering locations when the animals surface only for a few seconds/minutes, but Argos can do this fairly efficiently. This is a subset of the movement track of polar bear PB 1 in Auger-Méthé et al. 2017, see `https://www.int-res.com/abstracts/meps/v565/p237-249/`. It's available to use here, thanks to Dr. Andrew Derocher.

Let's read and peak at the data. You need off course to be in the good directory.

```r
# Read the file
pb <- read.csv("polarbear.csv", stringsAsFactors = FALSE)
# Look at the data
head(pb)
```

```
##      Acquisition.Time Argos.Location.Class Argos.Latitude Argos.Longitude
## 1 2009.04.20 17:01:39                    B         70.365        -131.837
## 2 2009.04.20 17:23:00                    A         70.953        -131.381
## 3 2009.04.20 18:12:15                    A         70.957        -131.365
## 4 2009.04.20 20:43:17                    A         70.909        -131.418
## 5 2009.04.21 17:11:29                    B         71.331        -131.607
## 6 2009.04.21 17:47:36                    B         71.225        -132.099
```

The first column, *Acquisition.Time* has the time of the location. The second column, *Argos.Location.Class* has the location quality class. Argos data can have large measurement errors, especially for marine animals, with some errors being in the range of 36 km. The location quality class indicates how reliable the location is. The categories are ordered as follow 3, 2, 1, 0, A, B, Z, with 3 being the highest quality, and Z being so unreliable that it is often excluded from the dataset. The last two columns, *Argos.Latitude* and *Argos.Longitude*, have the coordinates of locations of the animal.

As you can see the locations are not taken at regular time intervals. With Argos data, the time of the location depends on when satellites are above the tags, when the tags are sending signals (that's usually preprogrammed) and when the signals from the tags can be received by the satellites (e.g. for marine animals, when the animal is out of the water). One of the big differences between the different state-space models available is whether we have a model that discretize the underlying movement into steps made at regular time intervals or whether we have a continuous underlying movement model. All of these models will handle the fact that the data is irregularly spaced but, to do so, will make different assumptions on the the underlying movement of the animal.

2 `bsam`

The first package we will explore is `bsam`, which was written primarily by Ian Jonsen and is associated with Jonsen et al. 2003 (`https://esajournals.onlinelibrary.wiley.com/doi/abs/10.1890/02-0670`) and many of Jonsen et al. subsequent papers. `bsam` is available on CRAN, but for it to work, you need to have JAGS installed on your computer.

Let's load `bsam`.

```
library(bsam)

## Loading required package:  rjags

## Loading required package:  coda

## Linked to JAGS 4.3.0

## Loaded modules:  basemod,bugs

##
## Attaching package:  'bsam'

## The following object is masked from 'package:stats':
##
##    simulate
```

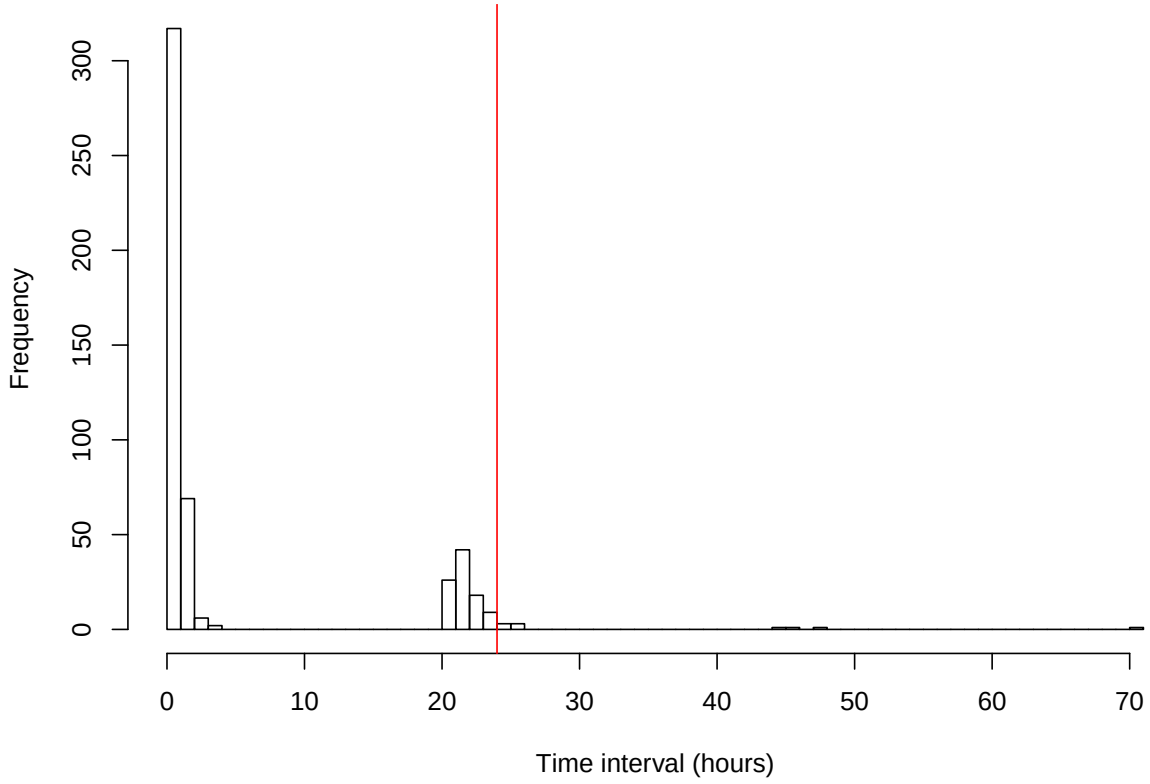
There are a few models available in `bsam`, but they all assume that the underlying movement is discretized in steps made at regular time intervals. In particular, these models assume that the animal moves in a straight line and at constant speed between two regular time steps. The observations are assumed to be on that line but with some error. So one of the important modeling decision that has to be done with `bsam` is the time interval between steps.

So let's look at how often we have locations. To do this, we will first transform the data into proper R time format. Note that here, the data is already sorted to be chronological, but if it wasn't, this is something you need to do before applying any of the models (and looking at the time difference).

```
# Transform time column into proper time format
pb$Acquisition.Time <- as.POSIXct(pb$Acquisition.Time,
    format = "%Y.%m.%d %H:%M:%S", tz = "GMT")
# Let's look at the time differences
ti <- as.numeric(diff(pb$Acquisition.Time), units = "hours")
summary(ti)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.2189  0.6292  5.2915  1.6828 70.4953
```

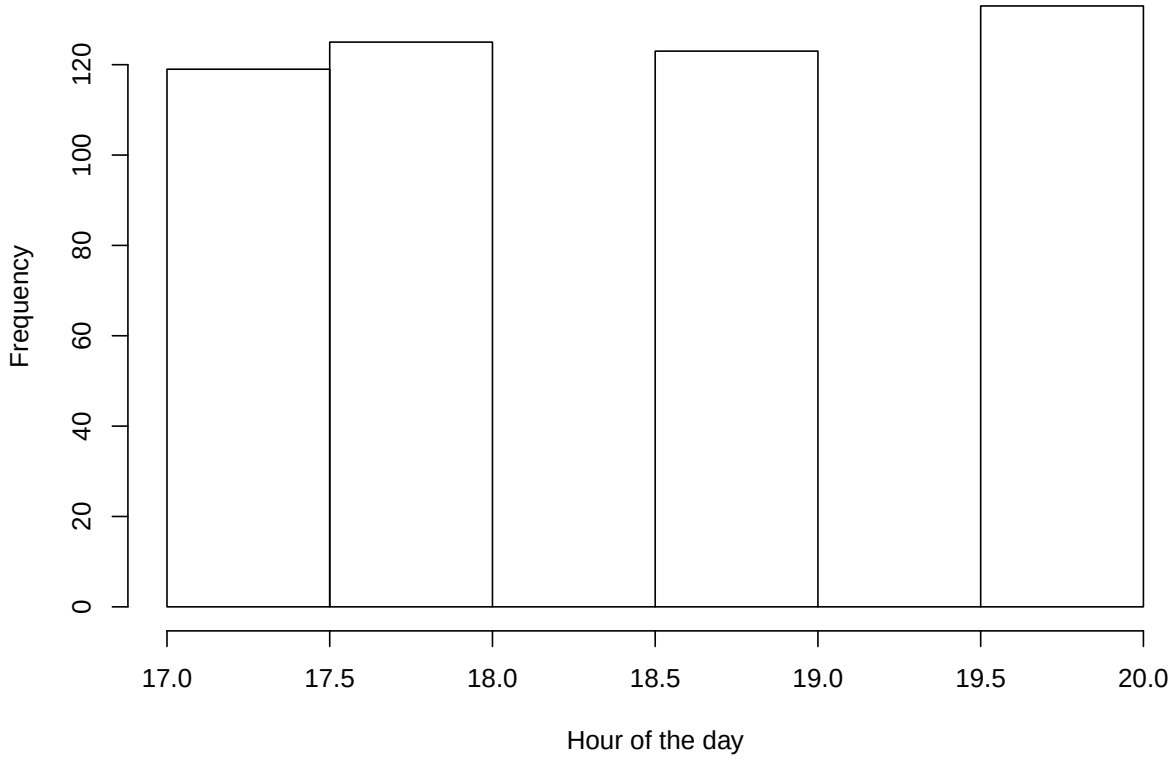
```
hist(ti, breaks = 100, xlab = "Time interval (hours)",
    main = "")
abline(v = 24, col = "red")
```




```
# At what hour of the day these locations taken?
tod <- as.numeric(format(pb$Acquisition.Time, format = "%H"))
summary(tod)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   17.00   18.00   19.00   18.54   20.00   20.00

hist(tod, xlab = "Hour of the day", main = "")
```



Ok, it looks like we have locations taken a few minutes to a few hours apart and locations taken a day apart. Looks like the locations are only between 17:00 and 21:00 (the collar was set to only send signals during this 4 hr block), so we have many locations during that time period and big gaps until the next day. So here, for a model with underlying discrete time steps, it makes sense to have time steps of one day. Smaller steps would results in estimating location at times where we have very little information.

Note, that one peculiarity of Argos data is that you sometime have locations exactly at the same time or very very close in time. That's why we have time differences equal to 0. This is not a problem for `bsam` but it can be for other models.

To be able to fit SSMs with `bsam`, the `data.frame` with the data needs to have very specific columns and column names. In particular, we need the columns *id*, *lc*, *lon*, *lat*. Optionally,

you can have two additional columns: *laterr* and *lonerr*. These could be use to specify the specific error for each location [1].

Let's format the files for `bsam`.

```
# Let's copy the original data set
pbP <- pb
# Let's look at the column names again
colnames(pbP)

## [1] "Acquisition.Time"     "Argos.Location.Class" "Argos.Latitude"
## [4] "Argos.Longitude"

# And rename them according to bsam's requirements
colnames(pbP) <- c("date", "lc", "lat", "lon")
# Add an ID column
pbP$id <- "PB1"
```

Note that we are adding a new column with the individual ID. We only have the track of one individual, so this column is not meaningful here. However, `bsam` allows you to fit the models to multiple individuals at the same time. There are options to decide whether the parameters are shared among individuals. You can select among these options by selecting different models available (e.g. DCRW vs hDCRW), see below and help file and Jonsen et al. 2016 (`https://www.nature.com/articles/srep20625`) for more details.

Ok, we are ready to fit a SSM with `bsam`. The main function to use is `fit_ssm`. The arguments are:

- `data`: that's our data frame, with the columns *id*, *lc*, *lon*, *lat*
- `model`: that's the model you want to fit, the default is *DCRW*, which is the simplest model. It's the one we will use first (so we won't set that argument below). See help file for the list of model you can fit.
- `tstep`: the time step of the underlying behaviour. The unit of the time steps is in days and the default is 1 day. As mentioned above, 1 day is the appropriate time step for our data set, so again we won't set that argument and use the default value.

[1] this is useful for other type of data, such as light-based geolocation. Note that when you use these columns the values in column *lc* need to be set to *G*, see help file for more details.

- `adapt`: the number of samples taken during the adaptation and burn-in phases of the chain, the default is 10,000. See the Bayesian primer below.
- `samples`: the number of posterior samples of the chain to run after the adaptation and burn-in phases. The default is 5,000. See the Bayesian primer below.
- `thin`: amount of thinning applied, to help decrease the autocorrelation in a chain's samples. See the Bayesian primer below.
- `span`: parameter that controls how the initial values of the chain are selected. See the Bayesian primer below.

`bsam` is a package with Bayesian models that uses JAGS (via `rjags`) to get posterior samples from the models using Markov chain Monte Carlo (MCMC). It's beyond the scope of this tutorial to go over Bayesian methods and theory in detail.

I'm partly assuming that you know the basic of MCMC. If you are familiar with MCMC, skip this paragraph. If you are unfamiliar, here is a very brief primer and key practical aspects you need to know to be able to fit a model with `bsam`. This is really just to give you an intuitive (-ish, maybe) view of MCMC. Before you start using `bsam` on your own data, you should familiarize yourself with Bayesian statistics, its potential issues, and ways to diagnose models. In a Bayesian framework you are interested in the posterior distribution which quantifies the probability of the model (in particular the parameter values) given the data and is a combination of the likelihood and of priors. The likelihood quantifies the probability of your data given your model, and in the term *model* we include the parameter values. The priors quantify how probable we think the different parameters values are. With MCMC you can sample your posterior distribution, even if your model is really complex and the likelihood is impossible to compute directly, that's in part why MCMC methods are so popular. To characterize your posterior distribution you sample at random possible values for the parameters and calculate the posterior based on this values. The way you sample the parameter space is through a random walk, where you select where in parameter space you will sample based on how good parameter values at the previous time step were (note here the time steps have nothing to do with the movement of the animal, these time steps refer to the iteration step of the sampling/fitting procedure). The time series of parameter values sampled is called the chain. How much of the parameter space you explored will depend on where you started in parameter space (i.e. your initial/starting parameter value. This is what the argument `span` affects) and on how many samples you took (this is what the arguments `adapt` and `samples` affect). Because at a given time step you are sampling parameter values that are close to the previous sampled parameter values, subsequent samples are correlated. The argument `thin` subsamples the chain to reduce that autocorrelation. In general, the

bigger the values of `adapt` and `samples` the better you are exploring the space. But bigger values, means much longer computing time. So it's a big trade-off. In general you have more than one chain, this is something I'll go back to. But as far as I know, the number of chains in `bsam` is set to 2 and cannot be changed easily.

For the purpose of this tutorial, I'm going to use smaller chains than the default values, because if not the models will take too long to run. Regardless, running the models below will take a few minutes, so be patient and just keep on reading.

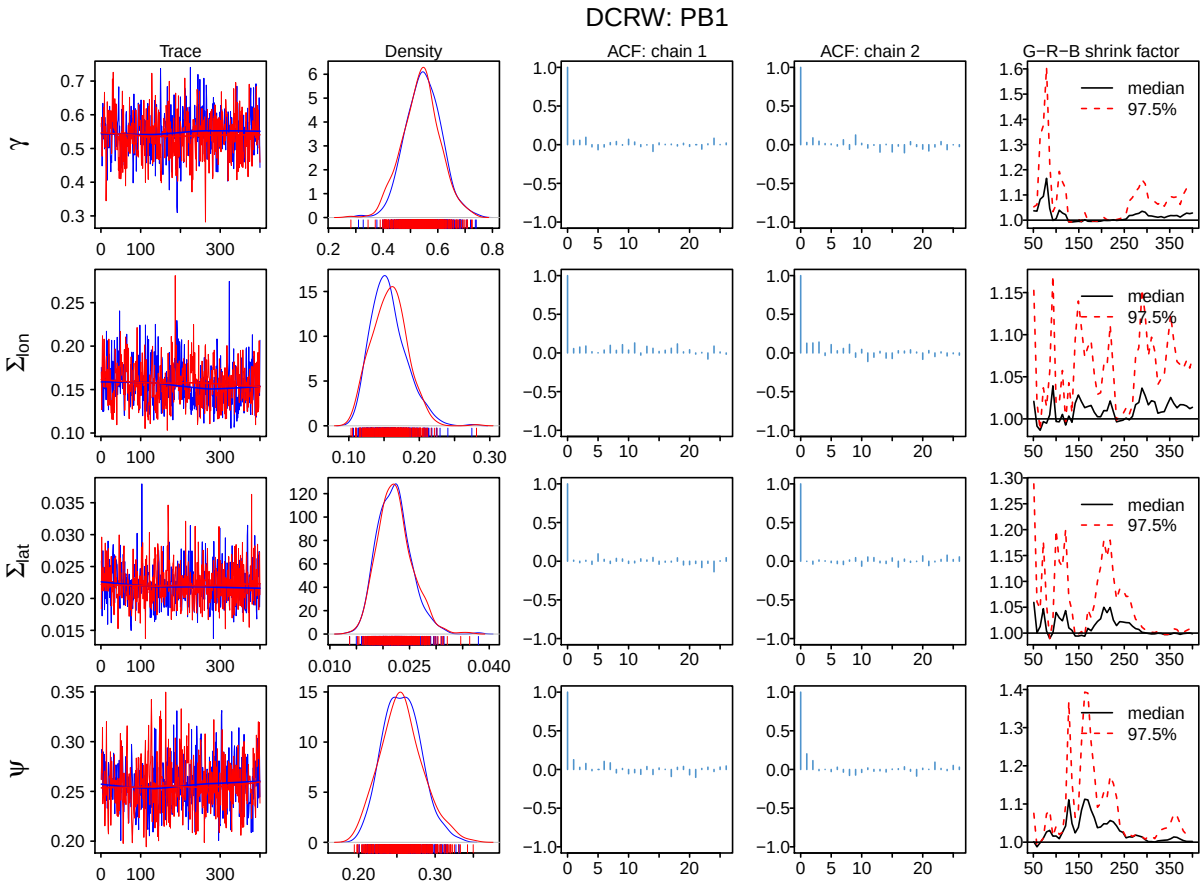
The first model we will explore, the *DCRW* assumes that the animal has a single underlying movement behaviour.

```
# Fit the DCRW - a model with a single underlying
# movement behaviour
dcrwB <- fit_ssm(pbP, tstep = 1, adapt = 5000, samples = 2000)

## Compiling data graph
##    Resolving undeclared variables
##    Allocating nodes
##    Initializing
##    Reading data back into data table
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 998
##    Unobserved stochastic nodes: 115
##    Total graph size: 7953
##
## Initializing model
##
## NOTE: Stopping adaptation
##
##
## Elapsed time:  1.33 min
```

To diagnose whether the model was properly fitted, we need to look at the Markov chains. To do so, we can use the function `diag_ssm`.

```
diag_ssm(dcrwB)
```

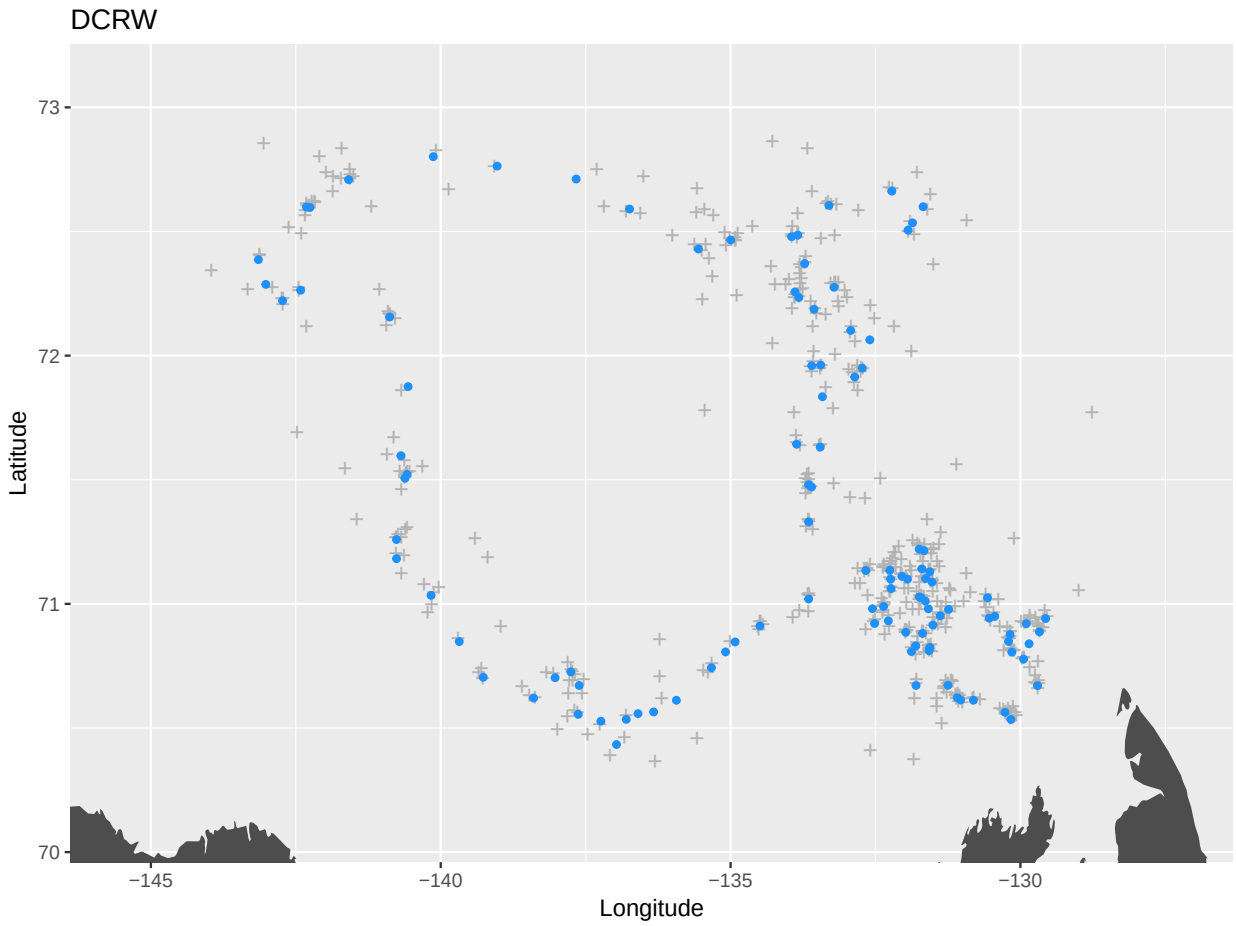


In the first column, we get the two chains (the two time-series of parameter values explored). The main thing you want to look for is whether the two chains are well mixed. Although not perfect, it looks fine. But you wouldn't want for example two separate lines or two diverging lines. In the second column, you get the posterior distributions computed with each chain. Here again things look ok. You want the two distributions (blue and red lines) to overlap and have the same general shape. In the next two columns, we have the autocorrelation functions of the two chains. You don't want autocorrelation. Here it looks like we might have some issues, especially for the parameter ψ (the last row). You could try to fix this by increasing the chain length and thinning more (e.g. set `samples = 4000` and set `thin = 10`). The last column gives you the temporal evolution of the Gelman-Rubin scale-reduction factor.

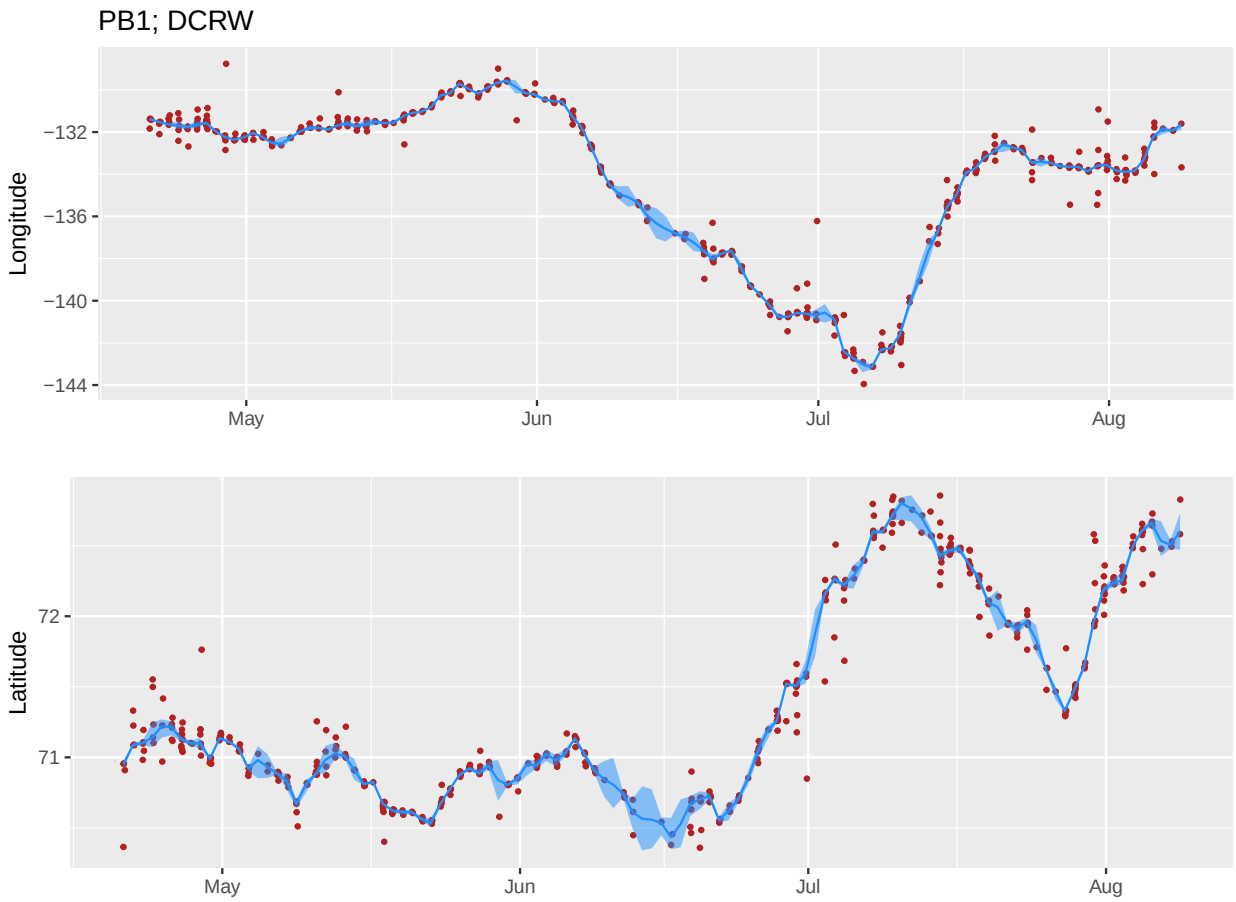
A scale-reduction factor of 1 means that the variance between the chains is the same as the within chain variance. As a rule if thumb you want the scale factor to converged towards a value lower than 1.1, indicating that each chain adequately sampled the parameter space.

Ok, for the purpose of the tutorial, this is good enough. Now lets look at the model. `bsam` provides two functions to look at the predicted *true* locations of the animal: `map_ssm` and `plot_fit`. In `map_ssm`, the observations are grey plus signs and predicted locations are blue circles. In `plot_fit`, the observations are red circles, the predicted locations are the blue lines with their estimated 95% credible intervals.

```
# Map the observed and predicted locations
map_ssm(dcrwB)
```



```
# Look at each geographic coordinates separately
plot_fit(dcrwB)
```



One thing you might notice is that, unsurprisingly, in the coordinate plots, the confidence intervals are wider when you have less observations. While the plots are great and all, you might often be interested in extracting the predicted locations so you can do your own plots or further analysis. To do so, you can use the function `get_summary`.

```
dcrwBtrack <- get_summary(dcrwB)
head(dcrwBtrack)

## # A tibble: 6 x 10
```

```
##   id    date                 lon   lat lon.025 lon.5 lon.975 lat.025
##   <chr> <dttm>             <dbl> <dbl>   <dbl> <dbl>   <dbl>   <dbl>
## 1 PB1   2009-04-20 17:01:39 -131.  71.0   -131. -131.   -131.    70.9
## 2 PB1   2009-04-21 17:01:39 -132.  71.1   -132. -132.   -131.    71.1
## 3 PB1   2009-04-22 17:01:39 -132.  71.1   -132. -132.   -132.    71.1
## 4 PB1   2009-04-23 17:01:39 -132.  71.1   -132. -132.   -131.    71.1
## 5 PB1   2009-04-24 17:01:39 -132.  71.2   -132. -132.   -132.    71.1
## 6 PB1   2009-04-25 17:01:39 -132.  71.2   -132. -132.   -131.    71.2
## # ... with 2 more variables: lat.5 <dbl>, lat.975 <dbl>
```

Notice that you have multiple *lon* and *lat* columns for each time step. The first two are the mean of the samples, the *.025*, *.5*, *.975* are the 2.5th, 50th and 97.5th percentile of the samples, respectively. You can use the *.025* and *.975* to create your credible intervals.

A very popular model from the package `bsam` is the DCRWS, for which the animal is assumed to have two underlying movement behaviours. Just like the hidden Markov models we have see in the previous tutorial, the animal switches between the two behaviours according to a Markov chain. The way to fit the model is identical to the previous model. The only argument we will change is the model. For the purpose of the tutorial, we will here again use small chains.

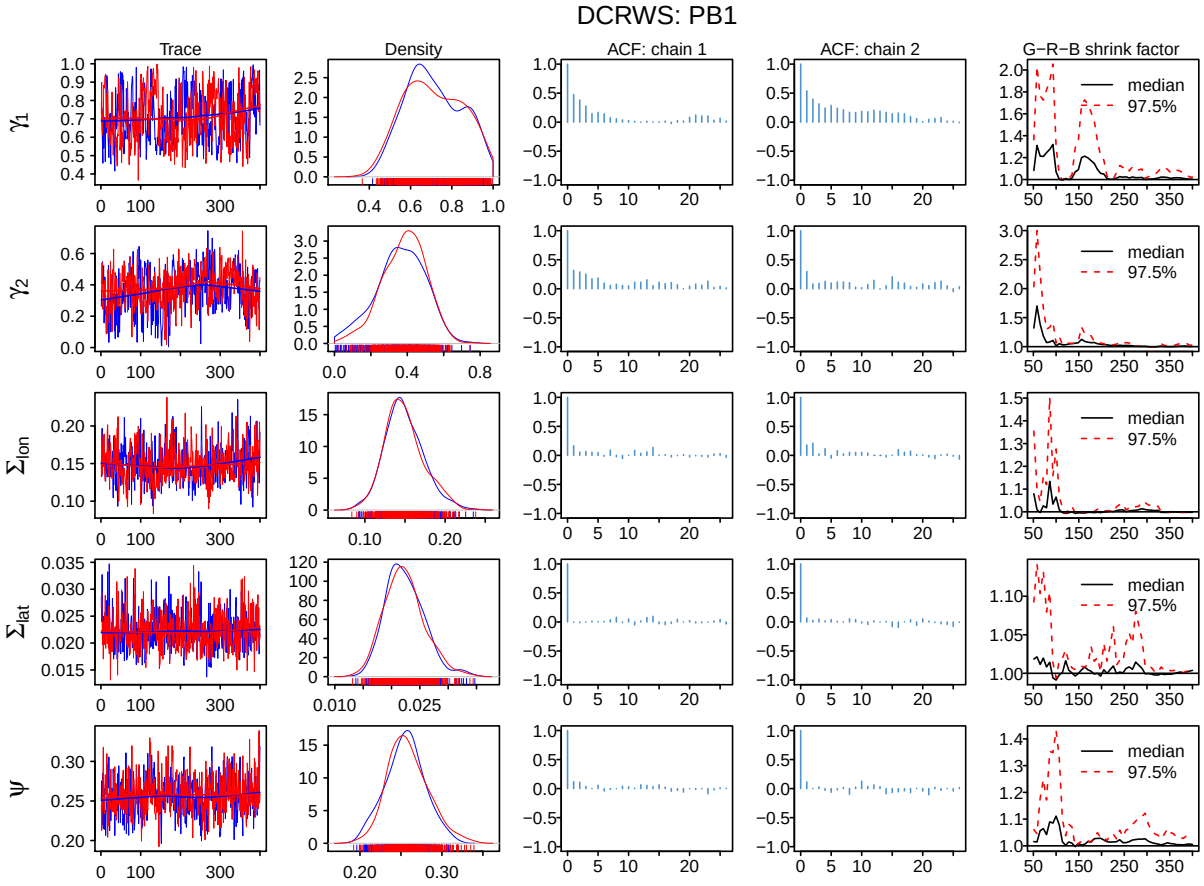
```
# Fit the DCRWS - a model with a single underlying
# movement behaviour
dcrwS <- fit_ssm(pbP, model = "DCRWS", tstep = 1, adapt = 5000,
    samples = 2000)


## Compiling data graph
##    Resolving undeclared variables
##    Allocating nodes
##    Initializing
##    Reading data back into data table
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 998
##    Unobserved stochastic nodes: 230
##    Total graph size: 8512
```

```
##
## Initializing model
##
## NOTE: Stopping adaptation
##
##
## Elapsed time:  1.87 min
```

Let's look at whether our posterior samples appear adequate.

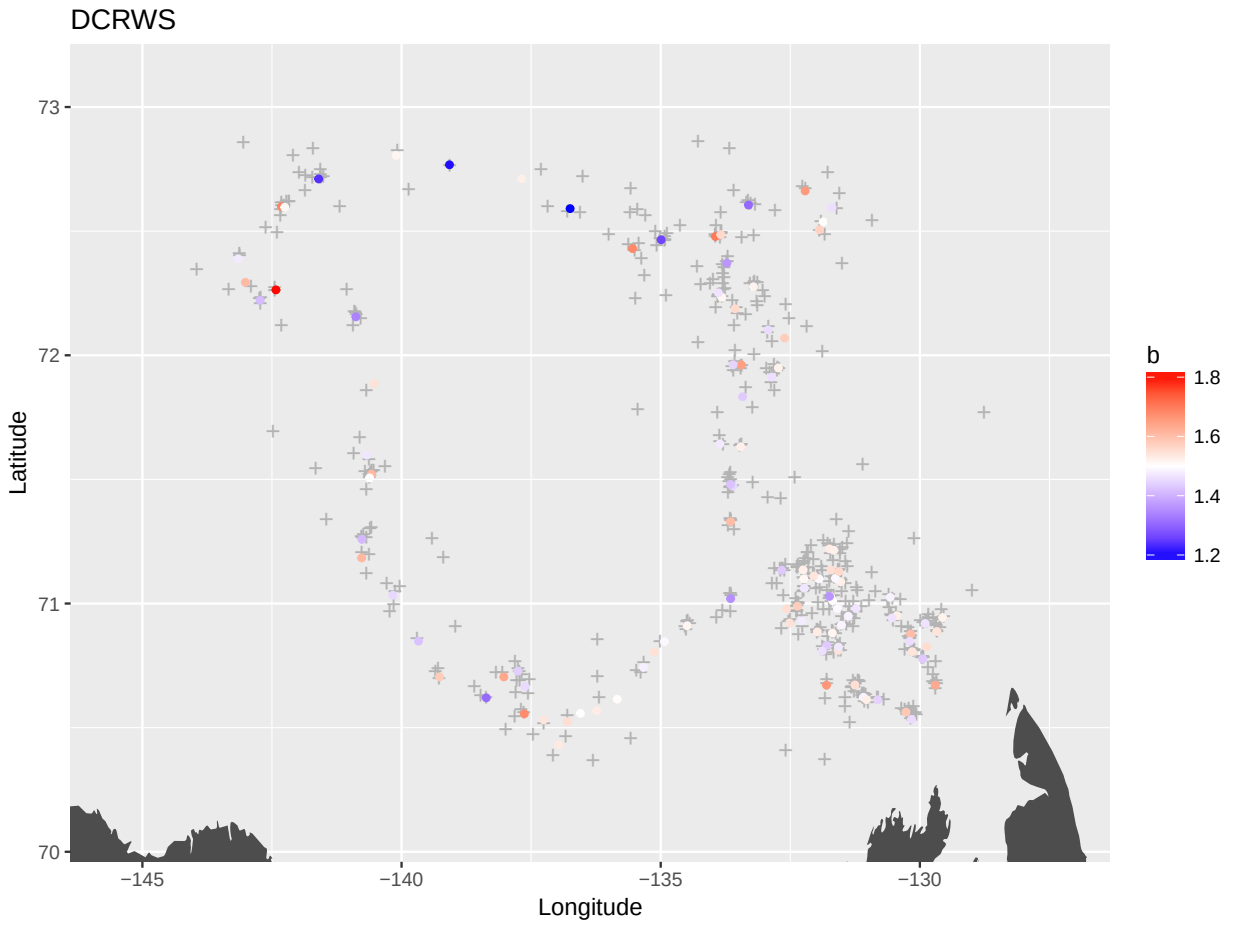
```
diag_ssm(dcrwS)
```



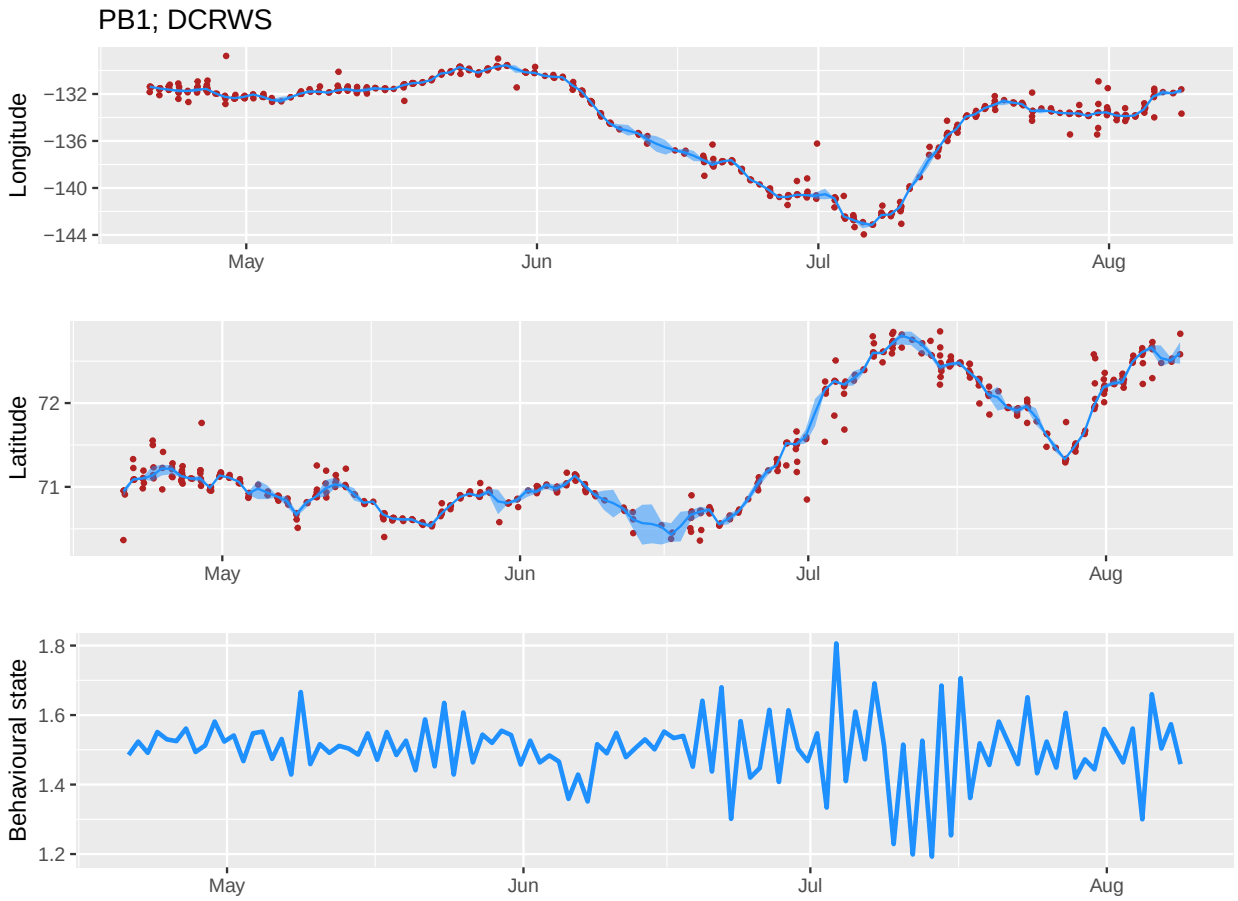
Ok, there are definitely a few bigger signs that there are problems with the samples here. The chains are not particularly well mixed for the two λs and their posterior distributions are not overlapping well. There is autocorrelation in the samples, especially for the top three parameters. So here, if we were to do this analysis for real, you would need to increase you chain, increase the thinning, and maybe even increase the adaptation period. It's not surprising that there are issues, because this is a much more complex model, with an added layer of unknowns (the behavioural states). However, just for time sake, for the tutorial we will stick with these results.

Let's look at the plots.

```
map_ssm(dcrwS)
```



```
plot_fit(dcrwS)
```



The first plot show us the map again, but here the colour indicates how close the behaviour is to the behavioural state 1 or state 2. In the second plot, we have an additional panel which shows similar information.

But what does that mean a behaviour of 1.6?

Just like for the hidden Markov models, the behavioural states decides which parameter is used to describe the observation. For the DCRWS, the parameter that changes according to the behavioural state is γ and this represent how correlated in speed and direction the current step is from the previous step. Here, the estimated parameters for each behavioural states are:

```
dcrwS$PB1$mcmc$gamma
```

```
## mcarray:
## [1] 0.7125789 0.3724827
##
## Marginalizing over: iteration(400),chain(2)
```

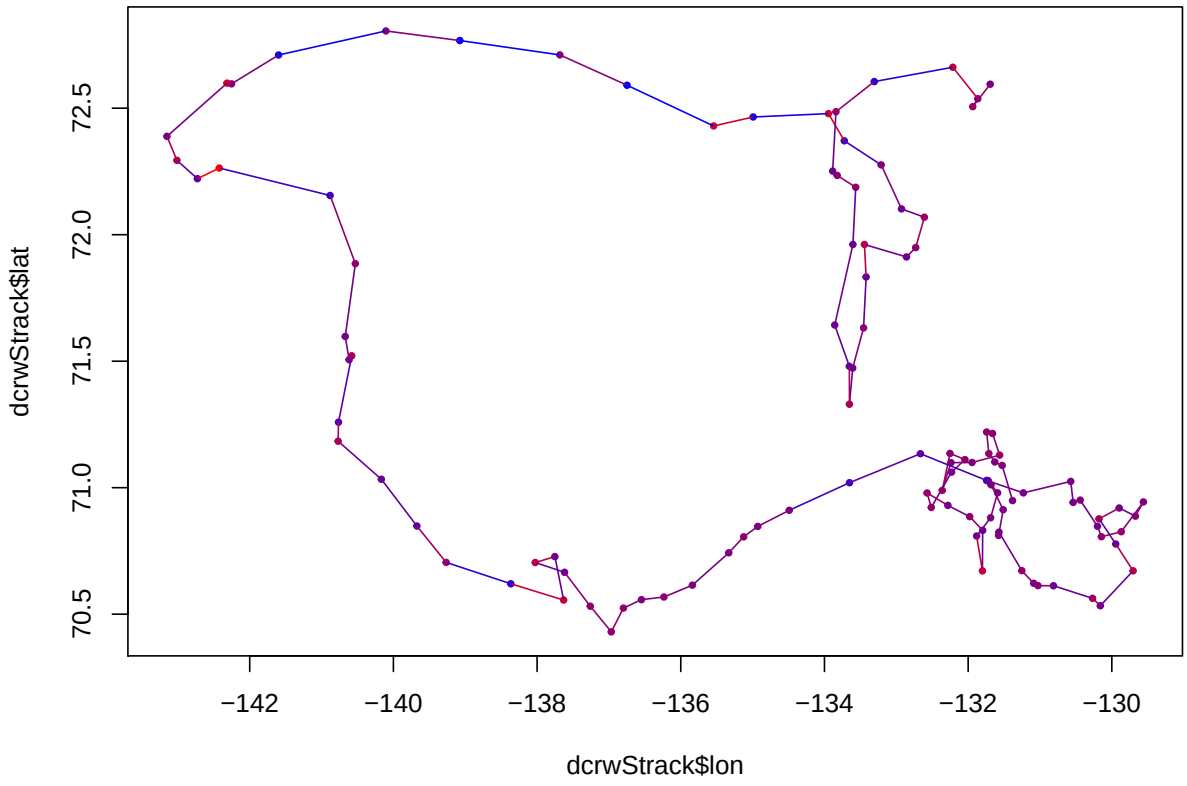
$\gamma_1 > \gamma_2$ which means that behaviour 1 has more auto correlated movement (travelling? than behaviour 2 (searching for food). We get values between 1 and 2 for the behaviour because the value returned is the mean across the MCMC samples. If we use the median (or the 50th percentile) we will get the most common value. You can get both of these by extracting the data.

```
dcrwStrack <- get_summary(dcrwS)
head(dcrwStrack)
```

```
##        id                date       lon      lat   lon.025     lon.5
## PB1.1 PB1 2009-04-20 17:01:39 -131.3817 70.94885 -131.4056 -131.3809
## PB1.2 PB1 2009-04-21 17:01:39 -131.5262 71.08789 -131.5668 -131.5265
## PB1.3 PB1 2009-04-22 17:01:39 -131.6284 71.10162 -131.6603 -131.6299
## PB1.4 PB1 2009-04-23 17:01:39 -131.7116 71.13492 -131.8696 -131.7014
## PB1.5 PB1 2009-04-24 17:01:39 -131.7428 71.21995 -131.8423 -131.7365
## PB1.6 PB1 2009-04-25 17:01:39 -131.6608 71.21426 -131.8100 -131.6593
##         lon.975  lat.025    lat.5  lat.975       b b.5
## PB1.1 -131.3589 70.88335 70.95298 70.96778 1.48500   1
## PB1.2 -131.4960 71.07147 71.08723 71.10491 1.52375   2
## PB1.3 -131.5756 71.08800 71.10139 71.11587 1.49125   1
## PB1.4 -131.6077 71.06954 71.12174 71.24470 1.55125   2
## PB1.5 -131.6773 71.13626 71.22412 71.26389 1.53000   2
## PB1.6 -131.4786 71.15200 71.21851 71.25792 1.52500   2
```

If we plot the movement as connected circles, we can see that the locations associated with lower b values are more straight and consistent than the locations associated with high b values.

```
# Get value between 0 and 1 for behaviour that span
# the range of value to make colors
colG <- (dcrwStrack$b - min(dcrwStrack$b))/(max(dcrwStrack$b) -
    min(dcrwStrack$b))
plot(dcrwStrack$lon, dcrwStrack$lat, col = rgb(colG,
    0, 1 - colG), pch = 19, cex = 0.5)
# To be able to color the line segment in different
# color associated with the behaviour, you need to
# use segment
segments(dcrwStrack$lon[-nrow(dcrwStrack)], dcrwStrack$lat[-nrow(dcrwStrack)],
    dcrwStrack$lon[-1], dcrwStrack$lat[-1], col = rgb(colG,
        0, 1 - colG))
```



This is a quick and dirty plot, just to show you how the behaviour is related to the persistence in movement.

Ok, that's where we will stop with `bsam`. There are two other models available in `bsam`: *hDCRW* and `hDCRWS`. These are essentially the same models, but when they are fitted to multiple individuals they assume that the some parameters are shared across all individuals and other parameters are individual specific (one parameter value is estimated for each individual) while the models we explored above assume that all parameters are individual specific (equivalent of fitting the model to each individual separately).

As a note, usually fitting these state-space models is the first step in an analysis and the researchers use the predicted locations or states to look at for example the relationship with environmental covariates. For example, you could apply an step selection function to the prediction locations. While this is the most common approach and there are almost no out-of-the box alternatives, it may be inappropriate in many instances to do so because it ignores the fact that the predicted locations are not data and they come with large uncertainties (look for example at the CI associated with some of the coordinate values).

3 argosTrack

An alternative to `bsam` is the package `argosTrack`. This package is associated with the paper Albertsen et al. 2015. `https://esajournals.onlinelibrary.wiley.com/doi/full/10.1890/14-2101.1`. It is not available on CRAN but is available Github at `https://github.com/calbertsen/argosTrack`. It can be installed with the package `devtools`. This can be a bit finicky, but as long as you have the packages `devtools` and `TMB` installed and the required tools[2] you should be able to install it as follow.

```
library(devtools)
install_github("calbertsen/argosTrack")  # Or install_github('calbertsen/argosTrack',re
```

Now let's load the package.

```
library(argosTrack)
```

There are multiple model you can fit with `argosTrack`, including the simple one-behaviour DCRW with fitted above (although with some slight differences). Let's start with this to explore the differences in the two packages.

[2] windows user see Rtools, `https://cran.r-project.org/bin/windows/Rtools/`, linux and mac users, you should be fine

Preparing the to fit the model the package `argosTrack` requires many steps. First, we need to create an `Observation` object, which collates the lat, lon, dates, and Argos quality class info. Second, we need to create a `Measurement` object, which allows you to set arguments like the distribution used for the measurement error (argument *model*).

```
# Create an Observation object
obs <- Observation(lon = pbP$lon, lat = pbP$lat, dates = pbP$date,
    locationclass = pbP$lc)
# Create a Measurment object, here specifying that we
# want to use a t-distribution for the measurment
# erro
meas <- Measurement(model = "t")
# Let's look at these
obs
```

```
## Observations:
## -------------
##
## Number of observations: 500
## First date: 2009-04-20
## Last date: 2009-08-08
## Location classes:
## qual
## GPS   3   2   1   0   A   B   Z
##   0   4   6  10   7 169 304   0
```

```
meas
```

```
##
##
## Measurement model
## -----------------
##
## Measurement distribution: t
## Use nautical observations: FALSE
## Variance parameters:
##           GPS        3        2         1         0        A        B
## Latitude    1 3.674437 108.4620 2214.5400 2113.2874 7211.267 69064.77
```

```
## Longitude   1 5.187231  60.7277  372.5607  442.5706 1620.030 61685.24
##                 Z
## Latitude  8886111
## Longitude 8886111
## Degrees of freedom parameters:
## GPS   3   2   1   0   A   B   Z
##   4   4   4   4   4   4   4   4
```

The Observation object prints the number of locations we have in each Argos quality category and some other general info on the data. The Measurement object prints information about the measurement error model, we see both the set of initial parameter values for the variance and the degrees of freedom. Note that this is a difference with `bsam`, here these parameters are estimated while they are fixed to specific values in `bsam`. Note however, that some of the parameters are assumed to be equal to decrease the number of parameters estimated[3]. Note that all variance parameters for the Argos categories are estimated, including the variance for the GPS and Z category, for which we have no information since we have no GPS or Z observations.

We also need to create an object associated with the underlying movement model you are interested to fit, here the DCRW. Again, the DCRW assumes that the animals make discrete step and for `argosTrack` to be able to know when these time steps occur, we need to create a sequence with these regular time steps. Here again, we need the dates to be in a proper time format (which we have already done above) and we will choose 1 day as our time interval.

```
# Check that it's a proper time format
class(pbP$date)

## [1] "POSIXct" "POSIXt"

# Create time step sequence
dseq <- seq(min(pbP$date), max(pbP$date), "day")
# Create our DCRW (model) object, which is a single
# behaviour first difference correlated random walk
movDCRW <- DCRW(dseq)
# Let's look at it
movDCRW
```

[3]For the degree of freedoms we estimate one value for GPS, 3, 2, 1, and 0, one value for A, and one value for B and Z.

```
##
##
## Movement model:
## -------------
##
## Model: Discrete Time Correlated Random Walk (DTCRW)
## Movement parameters:
## logit[0,1](gamma)               phi  logit[-1,1](rho)
##                 0                 0                 0
## Movement variance parameters:
## log(sigma_lat) log(sigma_lon)
##              0              0
## Number of latent variables: 222
## Using nautical states: FALSE
## Using time unit: hours
```

Just like the Observation object the DCRW object has information on the model. In particular the starting parameter values.

Ok, we now have all of the elements to fit the model to the data, but we need to combine all the elements in an object type called `Animal`. I'm not sure why the structure of the object is so complex, but as far as I know, you have to construct these to be able to fit it with `argosTrack`.

```
# Create Animal object
anim <- Animal(name = "pb", observation = obs, movement = movDCRW,
    measurement = meas)
```

Ok, we can fit the model to data. The main difference with `bsam` is that the model here is fitted with `TMB` rather than with `rjags`/JAGS. I'll go into more details of `TMB` in the next section but quickly, something to note is that `TMB` is used here to maximize the likelihood rather than to sample the posterior like `rjags` does in `bsam`. It does so numerically and uses tricks to handle the complexity of the model. In particular it uses the Laplace approximation to estimate the underlying states (here not the behavioural states, but the states in statistical terms, which for this model means the true unobserved locations of the animal). It makes the model fitting extremely efficient. `TMB` is really flexible, but has some limitation, see section below.

Ok, back to the specifics of `argosTrack`. Once you have your Animal object you can use the function `fitTrack` to fit the model. I set the argument *silent* to TRUE because if not the message about inner and outer minimization would fill in the page.

```
dcrwAT <- fitTrack(anim, silent = TRUE)

## Warning in sqrt(diag(cov)):  NaNs produced

## Warning in sqrt(diag(object$cov.fixed)):  NaNs produced

## Warning in sqrt(diag(object$cov.fixed)):  NaNs produced
```

We got warnings. Should we be worried? And more generally was this model well fitted? It's a hard question. But let's look at the returned object. In particular, the *message* and the *gr*.

```
dcrwAT$message

## [1] "false convergence (8)"

dcrwAT$gr

##  [1]  4.796348e+08 -1.912914e+10 -4.436435e+09 -1.381978e+10 -3.299968e+09
##  [6] -4.712804e+09  7.396958e+08 -5.051887e-02  1.336492e+00 -6.112208e+05
## [11] -3.264620e+04  2.108842e+00 -1.977895e+00 -1.173833e+10 -2.230730e+08
## [16]  7.026141e+09  9.628014e+08 -2.845353e+04 -1.683901e+09  4.003808e+07
```

Here we have a false convergence message, which in itself is not always a problem. But the gradients are far from 0. If we were at the MLE, we would expect gradients of 0. So I think we have some problems. This could be because the model is not adequate for the data. Also the fact that we are estimated parameters for which we have no information (see above) is going to cause problems. So I would not end my analysis here!

But for the purpose of the tutorial, let's continue on.

`argosTrack` has the peculiarity of changing the object that is being fitted directly[4]. So for example, compare these values to those printed for these exact same objects above.

[4] which is partly related to `TMB` ways of handling objects

```
meas
```

```
##
##
## Measurement model
## -----------------
##
## Measurement distribution: t
## Use nautical observations: FALSE
## Variance parameters:
##                    GPS            3            2            1            0
## Latitude  1.418944e-06 5.213820e-06 8.306132e-05 0.0001929079 0.0007459537
## Longitude 2.056483e-05 1.066745e-04 6.201162e-04 0.0007703085 0.0112520167
##                      A           B         Z
## Latitude  7.718973e-05 0.003001284  12.60889
## Longitude 5.776471e-04 0.031712966 182.74134
## Degrees of freedom parameters:
##      GPS        3        2        1        0        A        B        Z
## 4.129943 4.129943 4.129943 4.129943 4.129943 3.715251 3.089568 3.089568
```

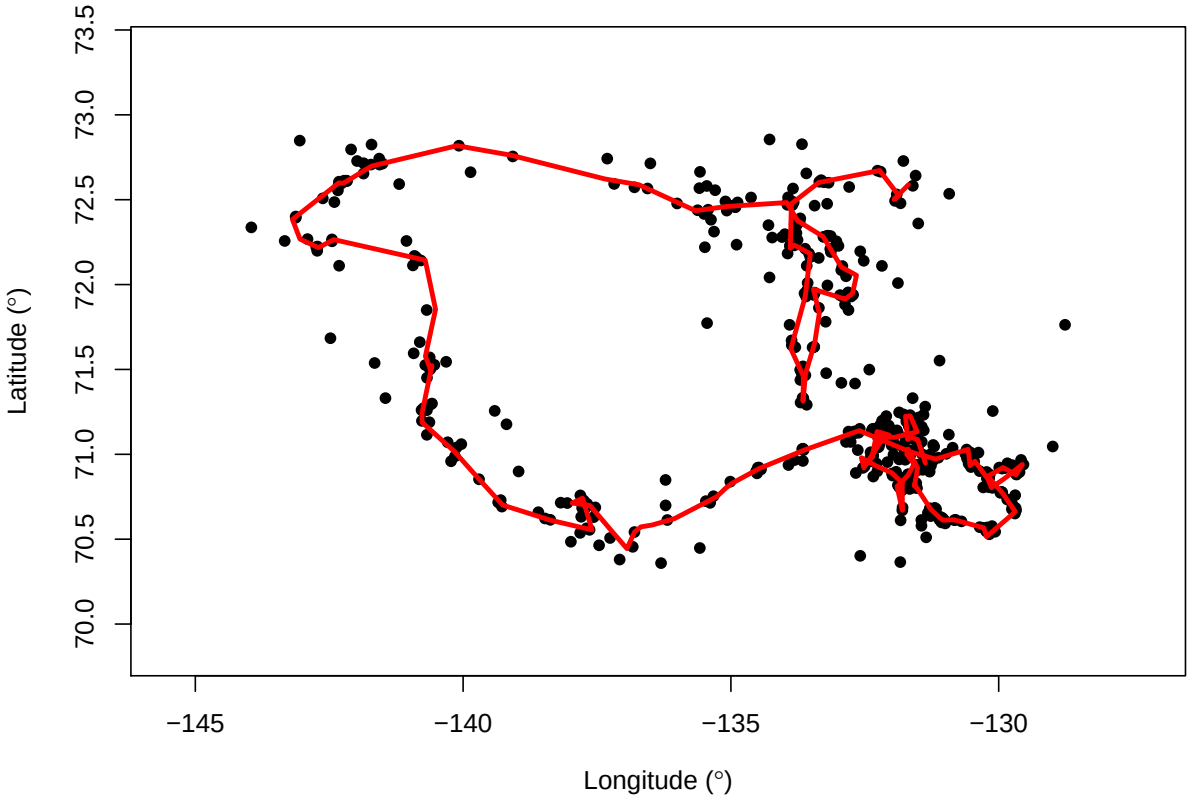
```
movDCRW
```

```
##
##
## Movement model:
## -------------
##
## Model: Discrete Time Correlated Random Walk (DTCRW)
## Movement parameters:
## logit[0,1](gamma)               phi  logit[-1,1](rho)
##       -0.08967415        0.05966983        0.09840666
## Movement variance parameters:
## log(sigma_lat) log(sigma_lon)
##      -2.223353      -0.920597
## Number of latent variables: 222
## Using nautical states: FALSE
## Using time unit: hours
```

These new objects display the estimated parameter values.

You can quickly plot the observations and predicted track using the function `plotMap` on your animal object.

```
plotMap(anim)
```



You can also extract the predicted track information to be able to do your own plots, using the function `getTrack` on your Animal object.

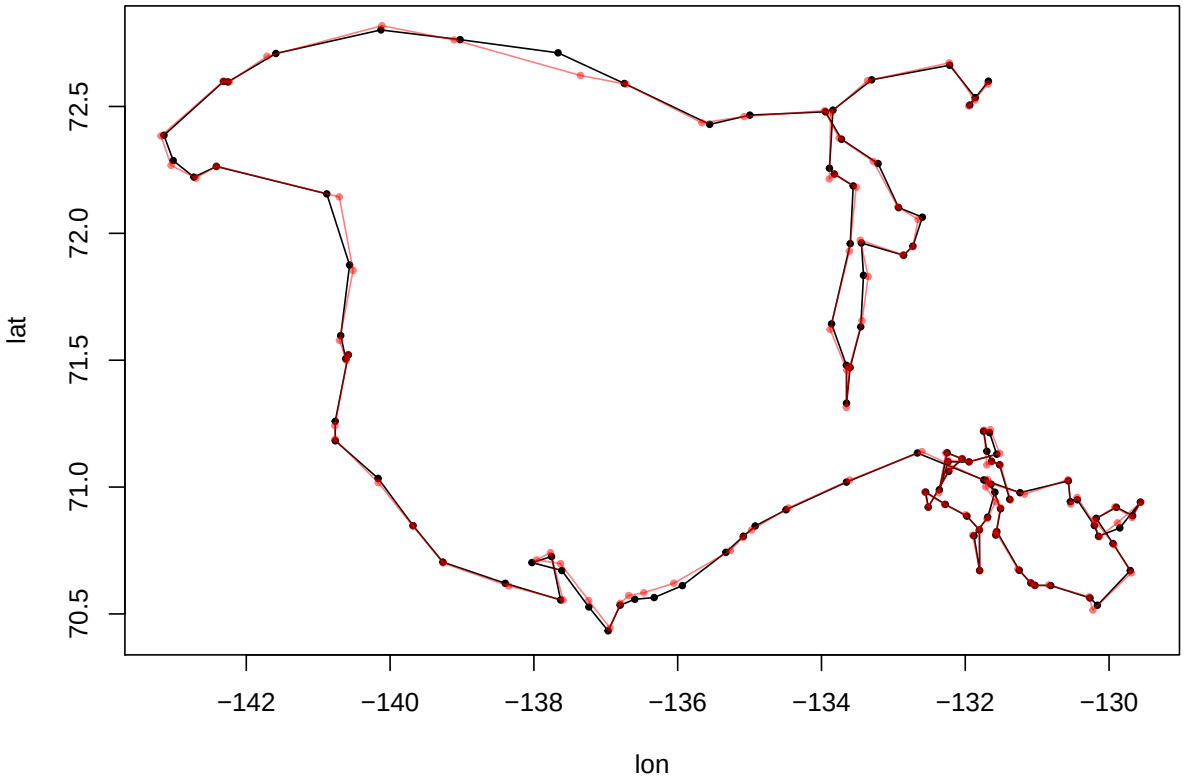
```
dcrwATtrack <- getTrack(anim)
head(dcrwATtrack)
```

```
##                  dates id obs.lat  obs.lon obs.lc  est.lat   est.lon
## 1 2009-04-20 17:01:39 pb  70.365 -131.837      B 70.94880 -131.3699
## 2 2009-04-20 17:23:00 pb  70.953 -131.381      A       NA        NA
## 3 2009-04-20 18:12:15 pb  70.957 -131.365      A       NA        NA
## 4 2009-04-20 20:43:17 pb  70.909 -131.418      A       NA        NA
## 5 2009-04-21 17:01:39 pb      NA       NA   <NA> 71.08649 -131.5123
## 6 2009-04-21 17:11:29 pb  71.331 -131.607      B       NA        NA
##      sd.lat    sd.lon
## 1       NaN       NaN
## 2        NA        NA
## 3        NA        NA
## 4        NA        NA
## 5 0.5168385 0.6178686
## 6        NA        NA
```

You can see here that you get the observed and the estimated (or predicted) locations. Note that the time of the observed and the estimated tracks are not necessarily the same. This is a function of how the model is constructed.

Just for fun, let's see how much the estimates differ from those from `bsam`.

```
# Extract estimated locations at the regular interval
# and drop rows with observations
dcrwATtrackEst <- dcrwATtrack[match(dcrwBtrack$date,
    dcrwATtrack$dates), ]
# Plot them on top of each other
plot(dcrwBtrack[, c("lon", "lat")], pch = 19, cex = 0.5,
    ty = "o")
points(dcrwATtrackEst[, c("est.lon", "est.lat")], pch = 19,
    cex = 0.5, col = rgb(1, 0, 0, 0.5), ty = "o")
```



So similar but not the same.

The main advantage of this package is that you can fit other models, including at continuous time correlated random walk (CTCRW). This is the same model as in Johnson et al. 2008 `https://esajournals.onlinelibrary.wiley.com/doi/10.1890/07-1032.1` and is a model that is also implemented in the package `crawl`, which unfortunately I won't have time to go over.

4 `TMB`

Stop me here!

While I presented the `argosTrack` package, because it's one of the package that provides many modelling options specific to animal movement. I find the package hard to navigate and it is limited in terms of options you can use. When I fit models to data, I write my own model using `TMB` directly. This might feel a little overwhelming, because it involves C++ code and involves writing your own likelihood, but it really opens the door to the type of models you can fit to movement data (and many other type of data).

As a bit of a self promotion, I have written a paper that shows how useful `TMB` is to fit animal movement data, see `https://www.int-res.com/abstracts/meps/v565/p237-249/`.

Let's load `TMB`.

```
library(TMB)
```

Now let's write the negative log likelihood function that we will minimize (equivalent of maximizing the likelihood). We write this function in special `TMB` C++ language. It needs to be save in a text file. I usually give it the extension .cpp. You can do this direction in R Studio just as you would write and save an R script, just save it as a .cpp file.

Ok, so our text file will be name dcrwTMB.cpp and will contain:

```
/*----------------------- SECTION A --------------------------*/
#include <TMB.hpp>

// Package needed for multivariate normal distribution
using namespace density;

/*----------------------- SECTION B --------------------------*/
// Defining main function
template<class Type>
Type objective_function<Type>::operator() ()
{

/*----------------------- SECTION C --------------------------*/
  // Specifying the input data
  DATA_MATRIX(y);

  // Specifying the parameters
  PARAMETER(logSdlat); // Log of st. dev. process stochasticity - latitude
```

```
  PARAMETER(logSdlon); // Log of st. dev. process stochasticity - longitude
  PARAMETER(logSdOlat); // Log of st. dev. measurement error - latitude
  PARAMETER(logSdOlon); // Log of st. dev. measurement error - longitude
  PARAMETER(logitGamma); // Autocorrelation - logit because 0 < gamma < 1

  // Specifying the random effect/states
  PARAMETER_MATRIX(x); // true locations

/*----------------------- SECTION D --------------------------*/

  // Transform standard deviations
  // exp(log) is a trick to make sure that estimated sd > 0
  Type sdLat = exp(logSdlat);
  Type sdLon = exp(logSdlon);
  Type sdOlat = exp(logSdOlat);
  Type sdOlon = exp(logSdOlon);
  Type gamma = 1.0/(1.0+exp(-logitGamma)); // logit-1 b/c we want 0 < gamma < 1

/*----------------------- SECTION E --------------------------*/
  // Set the variable that will keep track of negative log-likelihood (nll)
  Type nll = 0.0;


/*----------------------- SECTION F --------------------------*/
  // Covariance matrix of process equation
  matrix<Type> covs(2,2);
  covs << sdLon*sdLon, 0.0,
          0.0, sdLat*sdLat;

  // Covariance matrix of measurement equation
  matrix<Type> covo(2,2);
  covo << sdOlon*sdOlon, 0.0,
          0.0, sdOlat*sdOlat;

  /* Function that calculates the neg log density (normal) of process and measurment
   * equation. Note that MVNORM_t assumes that the mean is 0,0 and returns negative
   * log density.
   */
```

```
  MVNORM_t<Type> nll_dens(covs);
  MVNORM_t<Type> nll_deno(covo);

/*----------------------- SECTION G --------------------------*/
  // Create a temporary variable used to input in multivariate normal.
  vector<Type> tmp(2);

  // Process equation for first step.
  tmp = x.col(1)-x.col(0);
  nll += nll_dens(tmp);

  // Process equation for remaining steps.
  for(int i = 2; i < x.cols(); ++i){
    tmp = x.col(i) - (x.col(i-1) + gamma*(x.col(i-1)-x.col(i-2)));
    nll += nll_dens(tmp);
  }

/*----------------------- SECTION H --------------------------*/
  // Observation equation
  for(int i = 0; i < y.cols(); ++i){
    tmp = y.col(i)-x.col(i);
    nll += nll_deno(tmp);
  }

/*----------------------- SECTION I --------------------------*/
  // Parameters to report (including their standard errors)
  ADREPORT(sdLat);
  ADREPORT(sdLon);
  ADREPORT(sdOlat);
  ADREPORT(sdOlon);
  ADREPORT(gamma);


/*----------------------- SECTION J --------------------------*/
  return nll;

}
```

Now that we have the negative log likelihood C++ file saved , we go back to R.

The first thing we want to do is compile the C++ code and load it so we can use it to calculate the negative log likelihood of our model.

```
compile("dcrwTMB.cpp")


## Note: Using Makevars in /Users/Marie/.R/Makevars
## [1] 0


dyn.load(dynlib("dcrwTMB"))
```

Now let's prepare the model for fitting. First we need to prep the data. This is going to be a list with all the items found in the .cpp files DATA_MATRIX or other similar DATA objects. Here we only have y which is the observations (lat and lon).

```
data <- list(y = t(pbP[, c("lon", "lat")]))
```

Then we need a list with the parameters. Again the names need to match those in the .cpp file. These are the starting values for the parameters.

```
parameters <- list(logSdlat = 0, logSdlon = 0, logSdOlat = 0,
    logSdOlon = 0, logitGamma = 0, x = matrix(0, ncol = dim(data$y)[2],
        nrow = dim(data$y)[1]))
```

Before we can fit the model, we need to create the TMB object with the function `MakeADFun`. This object will combine the data, the parameters, and the model and will create a function that calculate the negative log likelihood and the gradients. To identify our random effects, here the locations, we set the argument *random* to equal the name of the parameters that are random effects. The argument `DLL` identify the compile C++ function to be linked

```
obj <- MakeADFun(data, parameters, random = "x", DLL = "dcrwTMB",
    silent = TRUE)
```

If you want to fix parameters to known values, something we won't do here but that is great to know, you want to use the argument map, see help file of `MakeADFun`.

Now we can fit the model. We do this using `nlminb`, which is a basic optimizer from R, but we input the values returned by TMB.

```
opt <- nlminb(obj$par, obj$fn, obj$gr)
```

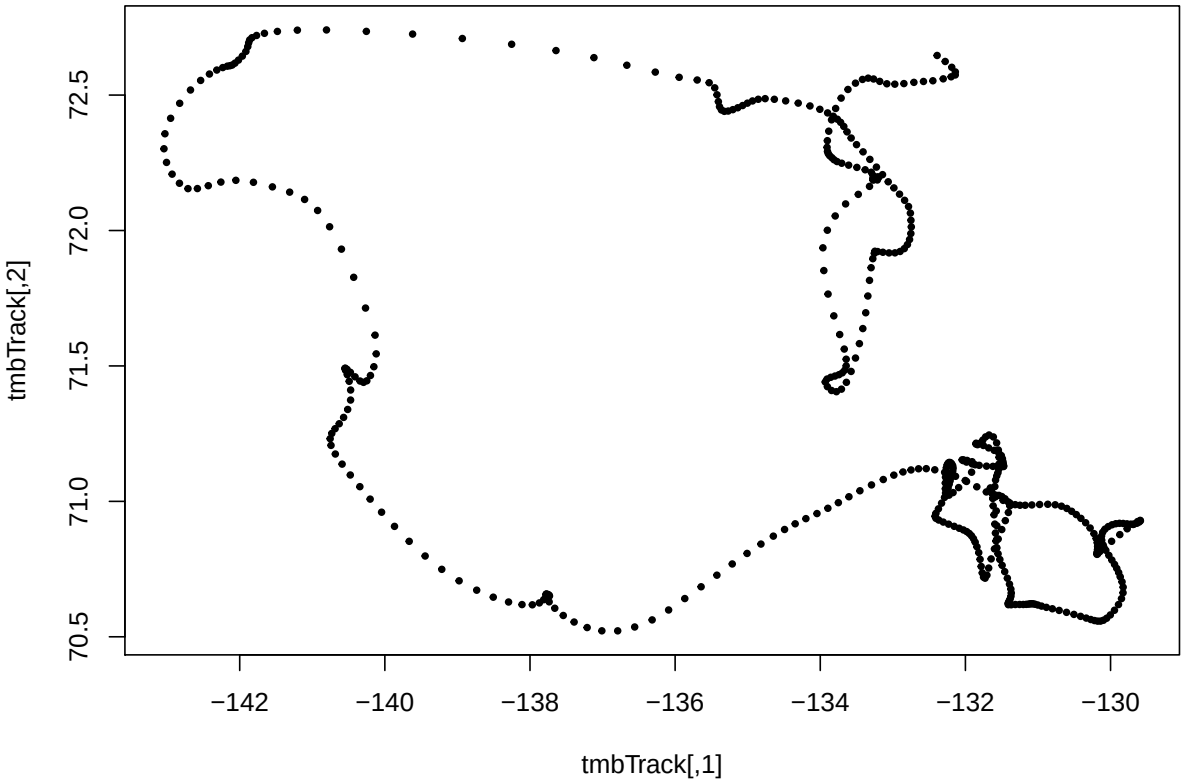
To look at the parameter estimates, you can use the function `sdreport`.

```
sdr <- summary(sdreport(obj))
# THis will have the parameters and states So we can
# just print the parameters
sdr[c("sdLat", "sdLon", "sdOlat", "sdOlon", "gamma"),
    ]


##          Estimate  Std. Error
## sdLat  0.01993633 0.004129587
## sdLon  0.08584922 0.017122466
## sdOlat 0.11638719 0.004449576
## sdOlon 0.48024964 0.017267003
## gamma  0.78099854 0.059621285
```

We can get the predicted locations as follow:

```
tmbTrack <- t(obj$env$parList()$x)
plot(tmbTrack, pch = 19, cex = 0.5)
```



This was a very simple model, but `TMB` is extremely flexible, so you could use other distribution for both the measurement and process equation. You can include other data streams, other random effects, ...

Note that the main limitation with `TMB` is that it's hard to have things like discrete states to be predicted. It's not impossible, but it requires some finagling. See Whoriskey et al. 2017 and Auger-Méthé et al. 2017 `https://www.int-res.com/abstracts/meps/v565/p237-249/` for ways to model changes in underlying behaviours.

5 Other state-space model packages for animal movement data

`crawl`, see CRAN

kfrack, see https://github.com/positioning/kalmanfilter/tree/master/kftrack