A tutorial for Step Selection Function

P. Antkowiak* H. Tripke† C. Wilhelm‡

November 26, 2014

Contents

*M.Sc. programme "GIS und Umweltmodellierung" at University of Freiburg
†M.Sc. programme "Wildlife, Biodiversity and Vegetation" at University of Freiburg
‡M.Sc. programme "Wildlife, Biodiversity and Vegetation" at University of Freiburg

1 Introduction

In addition to Resources Selection Functions (RSF) another powerful tool for evaluating data on animal movements and habitat selection are Step Selection Functions (SSF). The latter are used to estimate resource selection by comparing observed habitat use with available structures. Given GPS locations of a collared individual each observation is connected by a linear segment. These segments are considered as steps. The time intervals influencing the step length should be choosen carefully (i.e. by conducting a pilot study) to meet the requirements of the study questions and the target species. The SSF than calculates random steps by taking measured angle and distance along steps and using the observed positions as starting points. These alternative steps represent the available habitat within a realistic step length of the observed positions. Finally, we can compare spatial attributes on both and test for effects that explain habitat selection by animals [2].
So far, SSF models were mainly done using Geospatial Modelling Environment (GME) that works with a GIS[1]. However, more and more packages for analyzing animal movements are provided in R. Non of these packages is designed for doing a SSF only but quite a number provide already helpful functions to perform single steps of the Selection Function. Therefore, the aim of this tutorial is to collect all functions necessary to conduct a SSF and order them in a way that intuitively makes you understand how to run a SSF with your own data. Each step will be explained using an exemplary dataset of GPS locations collected from seven Cougars (*Puma concolor*) in the year 2010 (in the following adressed as `xmpl`).

Figure 1 provides an overview of all necessary steps and potential options to conduct a SSF. This tutorial will guide you through each step and gives brief instructions on how to implement the functions and what to consider beforehand. To conduct a SSF using this tutorial we need you to store your initial data in two independant datasets:

1. A raster file of your spatial attributes (*Raster data*) and
2. GPS locations of your individuals assigned with a time stamp (*Waypoint data*).

We will start with the *Waypoint data* because these need to transformed a couple of times to be able to work with them. You can find the single steps on the right site of Figure 1. While there are many options to adjust your *Waypoint data* the *Raster data* describing your spatial attributes needs not much of a change. Once you created random steps for your observed positions you can extract the spatial attributes for each of those positions by using the function `extract`. At this point *Waypoint* and *Raster data* will be combined and your final model can be written.

1.1 Installing and loading Packages

Before you can actually start using this tutorial for conducting SSF you need to install a bunch of packages in R. Some of them require others so that you have to add all these to your library:

```
## for implementing SSF

install.packages("adehabitatHR")
install.packages("adehabitatHS")
install.packages("adehabitatLT")
install.packages("adehabitatMA")
# Keep fingers off the "adehabitat" package! It is outdated.

install.packages("tkrplot")
```

[1] www.spatialecology.com/gme/

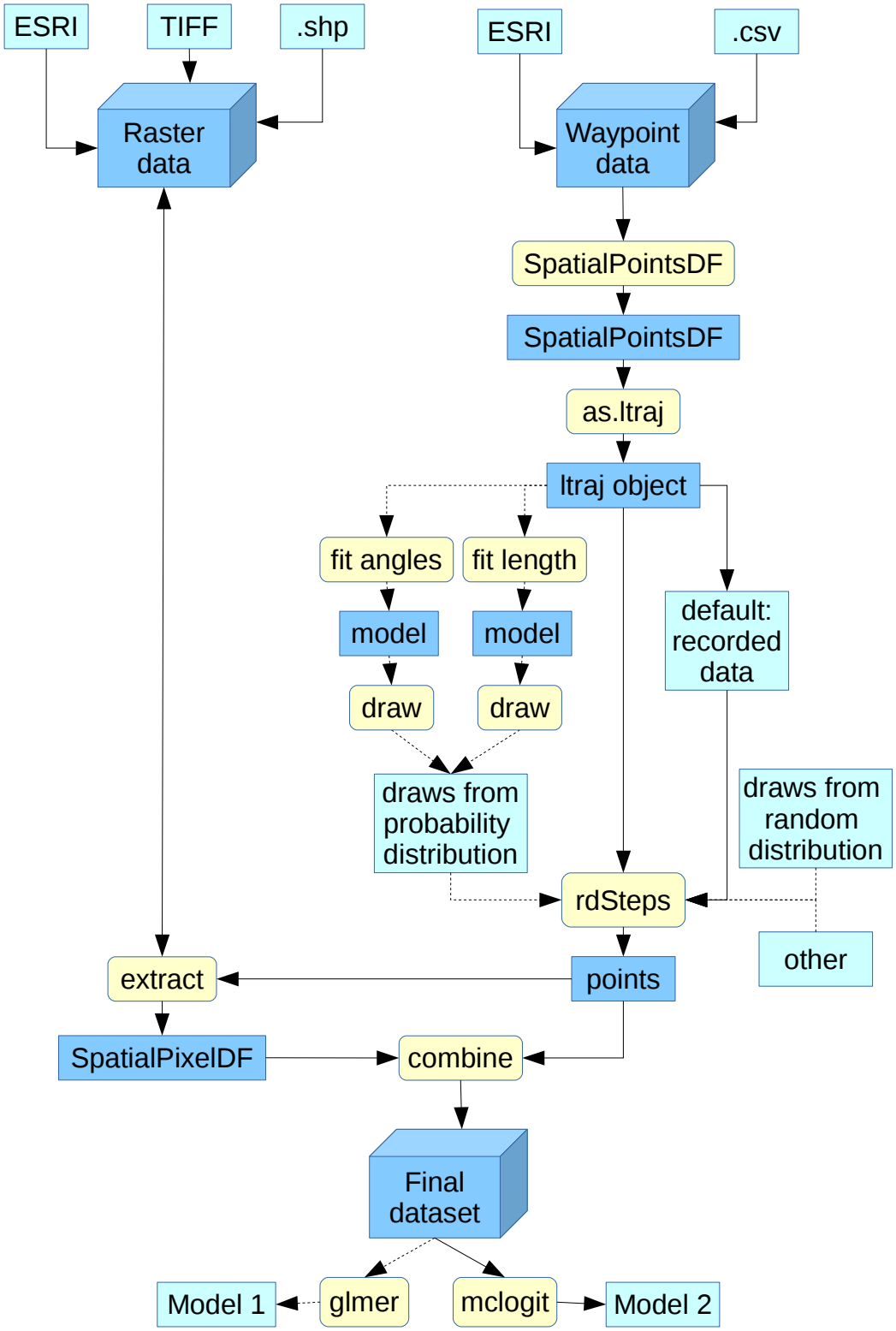

Figure 1: Conducting a Step Selection Function using existing R-packages. The yellow boxes show the name of the fucntion applied while the blue boxes provide the type of object or data. Following the arrows a step by step instruction is provided ...

```
install.packages("hab", repos = "http://ase-research.org/R/") # regular
install.packages("hab", repos = "http://ase-research.org/R/", type = "source") # for self-c

# for handling raster data
install.packages("move")
install.packages("raster")
install.packages("rgdal")
```

Loading packages:

```
require(hab)
require(adehabitatMA)
require(adehabitatHR)
require(adehabitatHS)
require(adehabitatLT)

require(sp)
require(raster)
#require(move)
#require(rgdal)
#require(tkrplot)
```

2 Processing the Waypoint Data (or telemetry?)

2.1 Loading Waypoint Data (*.csv, ESRI)

The data for the analysis should be saved in a simple *.csv file format. The table should have column headings in the first line and for each observation include at least the following values:

1. x-coordinate (easting)
2. y-coordinate (northing)
3. date and/or time
4. animal ID

Note that the coordinates need to be provided in the same coordinate system and spatial projection as the raster data.

Depending on your analysis you can include further values such as:

1. ID for each record
2. GPS precision
3. other recording parameters such as season
4. temperature / elevation at the moment of record
5. other values that might be of interest in the further analysis

Use the following commands to set your working directory and read the data:

```
setwd("/your/working/directory")
xmpl = read.csv("xmpl.csv", head=T)
```

You can execute `head(xmpl)` and `str(xmpl)` to check, whether the data were successfully read.

2.2 Broken stick model

2.3 Creating a Spatial Points Data Frame

The functions used along the rest of the toolchain can only process data that are stored as objects of class "SpatialPointsDataFrame". This object class stores the coordinates separately and can be created using the according function from package "sp".

```
require(sp)

xmpl.spdf = SpatialPointsDataFrame(coords = xmpl[,c("easting","northing")], data = xmpl)

names(xmpl)
```

In the "coords" argument of the function you should specify the coordinate vectors for your dataset. In our example, we simply assign the two coordinate columns of the example dataset, but you can read the coordinates from a separate file if you want.

2.4 Creating an ltraj object

After storing the data in a Spatial Points Data Frame, you now need to connect the single points and turn them into a set of trajectories. This operation is carried out by the function `as.ltraj` from the "hab" package and produces objects of class "ltraj". The function `as.ltraj` requires at least three arguments to work:

1. "xy" (x- and y- coordinates for each point)
2. "date" (timestamp for each point, given as POSIXct class)
3. "id" (the animal id)

Both coordinates and animal id can easily be adopted from the Spatial Points Data Frame. The timestamp however, is not stored in the required format yet and you therefore need to convert it first:

```
date <- as.POSIXct(strptime(paste(xmpl.spdf$LMT_DATE, xmpl.spdf$LMT_TIME), "%d/%m/%Y  %H:%M
```

It is necessary to combine date and time in one POSIXct value. If your dataset already features a POSIXct timestamp, you can skip this step.

Now you can proceed and actually create the ltraj object by executing the following command:

```
xmpl.ltr <- hab:::as.ltraj(xy = xmpl.spdf@coords, date = date, id = xmpl.spdf$cat)
```

Two comments to the function as used: By typing `hab:::as.ltraj` you tell R to use the `as.ltraj` function from the "hab" package which is speed optimized against its `adehabitatLT` sibling. Unlike the "xmpl.spdf@coords" promt which works for any SPDF object, the `xmpl.spdf$cat` prompt is specific to your dataset. In the example dataset, animal ID's are stored as an integer vector called "cat". If this differs in your dataset and you should change the prompt accordingly.

You now may want to have a closer look at the created ltraj object. Display its structure with by executing `str(xmpl.ltr)`.

```
str(xmpl.ltr)
```

When scrolling through the output you will first notice that it consists of 7 elements - the number of individuals in the example dataset. This is because `as.ltraj()` automatically splits the dataset into subsets one for each individual. We will refine these subsets in the next chapter. Furthermore, the ltraj contains information on the distances and turning angles between consecutive locations. To get a visual impression of your data you can plot the trajectory for all or for one particular animal:

```
plot(xmpl.ltr)
unique(xmpl.spdf$cat) # prompts a list of all cat ID's. Choose one that you are interested
plot(xmpl.ltr, id=10289)
```

2.5 Creating Bursts

As described in the previous section, the relocations stored in the ltraj object are already devided into the different individuals. This partition is called a "burst". For analysing the data, there might be the need to create "sub-bursts" for each animal within your trajectory. For example, if the animals were only recorded during the day, the monitoring took place over two consecutive years or the time lag between the relocations differs remarkably, each accumulation of relocations can be defined as a different burst. Looking at those different parts seperately might be necessary for different reasons. The function `cutltraj` splits the given bursts of your ltraj object into smaller bursts according to a specified criterion. In contrast, the function `bindltraj` combines the bursts of an object of class "ltraj" with the same attribute "id" to one unique burst. To find out if there are more missing values, you can plot the ltraj object. For that, you need to define the time interval you are looking at. [1]

In our example, the locations of the cougars were recorded every 3 hours, starting at 3 AM. The location at midnight is always missing. We now want to split the existing bursts (individuals) into "sub-bursts" where the time lag is smaller than 3 hours. To get an impression about the time lags we plotted the different bursts (individuals). "dt", the time between successive relocations is measured in seconds.

```
plotltr(xmpl.ltr, "dt/3600/3")
```

To cut our data at our desired interval, we need a function which defines "dt". Because we want to keep relocations which are only a few minutes "wrong", we added 10 extra minutes.

```
foo = function(dt) {return(dt> (3800*3))}
```

Then we split the object of class ltraj into smaller bursts using `cutltraj` and the function above. The bursts we had before applying this function still remain.

```
xmpl.cut <- cutltraj(xmpl.ltr, "foo(dt)", nextr = TRUE)
```

There are two options of cutting the trajectory depending on nextr. If it is set as FALSE, the burst stops before the first relocation matching the criterion. if it is set as TRUE, it stops after. [1]

2.6 Creating Random Steps

Given your final bursts we now randomly draw angle and distance from your observed data to get random steps. The angle is taken from the observed position before your starting point of the random step while the distance is taken from the starting point to the next observed position. This means that each burst should at least contain of three observed positions. Before applying the function `rdSteps` you might want to check for correletaion between turning angle and distance. In case your individuals tend to move long distances by turning only in small angles (e.g. a species migrating) but stop for several days for feeding you want to pick the distance and the angle as pairs. If no correlation is found you can pick both variables independently. To check for this we use the `plot` function but first have to convert our ltraj object back to a data frame by using `ld`. AUTOCORRELATION ???

```
## Error in eval(expr, envir, enclos): konnte Funktion "ld" nicht finden
```

```
with(xmpl.cut.df, plot(dist, rel.angle))

## Error in with(xmpl.cut.df, plot(dist, rel.angle)): Objekt 'xmpl.cut.df'
nicht gefunden
```

The plot shows a correleation of step length and turning angle and therefore the random steps should be taken as pairs (`simult = T`). Per default the angle and distance for each random step is drawn from the observed values you provide with `x`. If your random steps shall be taken from a different dataset you can do so by writting it in `rand.dist = YourDataSet`. Hereby, you can also specify a distribution for estimating angle and distance. In our case we stick to the same dataset and apply `rdSteps`.

```
xmpl.steps <- rdSteps(x = xmpl.cut, nrs = 10, simult = T, rand.dis = NULL,
                      distMax = Inf, reproducible = TRUE, only.others = FALSE)
          # use simult = FALSE if your data is not correlated
```

The function `rdSteps` uses some default settings which offer you options to modify your random steps. You can for example, easily change the number of steps taken from the observed data by defining `nrs` (default is 10) or if you only need steps shorter than a certain value specify `distMax` to that value (per default all steps are taken). By setting reproducible = TRUE a seed is used to get reproducible random steps. If you want to exclude your current individual to draw angle and distance from than set `only.others = TRUE`. All in all, `rdSteps` is very straight forward and computes a lot of useful things for you:

```
head(xmpl.steps)

## Error in head(xmpl.steps): Objekt 'xmpl.steps' nicht gefunden
```

The table still includes your cat id, burst id, the rel.angle and dist of your observed positions. Furthermore, the "case" is provided as categories of 0 and 1 for available and used. The "strata" defines all 10 random steps and the one observed location, so you can later tell your function what to compare. Depending on your analysis you might want to compare only your observed positions with the endpoints of your random steps or you want to investigate in the selection of spatial attributes along the path. To do so you find the rel.angle and distance for each random point. Only new coordinates for your random points are missing. Instead two columns provide the differences of your x- and y- coordinates for each random step ("dx" and "dy"). To get new coordinates for your random steps we simply add these differences to your initial coordinates and create two new columns.

```
xmpl.steps$new_x <- xmpl.steps$x + xmpl.steps$dx
xmpl.steps$new_y <- xmpl.steps$y + xmpl.steps$dy
```

Thereby, the ???rst observed position will be overwritten as the ???rst random step. That is necessary because there is no random step to compare the ???rst observed position with (angle could not be calculated!). Also the last observed position will be lost for similar reasons (distance could not be calculated!).
After running this chapter you get your final SpatialPointDataFrame to use with the selection function.

```
head(xmpl.steps)

## Error in head(xmpl.steps): Objekt 'xmpl.steps' nicht gefunden
```

3 Processing Spatial Covariates

This section explains the handling of spatial parameters that will be tested for selection by the target species. You should store these data in raster files (ESRI *.adf or georeferenced *.tif). These should have the same coordinate system as your telemetry data and should (for time reasons) already be clipped to your study area and. For instructions how to do this in R, please read the GIS instructions from the other group ;)

3.1 Load Raster Data (ESRI, *.tif, (*.shp))

With a simple function stored in the package **raster** you are able to upload any raster file into R. Examplarily we use raster data on the following parameters for the study area:

1. ruggedness of the terrain
2. land cover
3. canopy cover
4. distance to the nearest highway
5. distance to the nearest road

For reading the raster data, three packages are required:

```
require(raster)
require(rgdal)
require(sp)
```

The source files for the raster data can be stored in the working directory or loaded by specifying the exact path. The `raster()` function is a universal and very powerful tool for loading all kinds of raster data. For reading shapefiles, use the `readOGR()` function. Below is an example of how to read a set of raster layers.

```
# First, make sure that your working directory is still the one specified earlier:
getwd()

# Now read the layers:
ruggedness <- raster("ruggedness.adf")
```

```
landcover <- raster("landcover.adf")
canopycover <- raster("canopycover.adf")
disthighway <- raster("disthighway.adf")
distroad <- raster("distroad.adf")
# In the ESRI directory system, raster layers are usually stored in files called "w001001.a
```

You can plot the data for a first overview. As this can take a while with large datasets, outcomment the following chunk if you please!

```
plot(ruggedness)
plot(landcover)
plot(canopycover)
plot(disthighway)
plot(distroad)
```

3.2 Checking for Collinearity

3.3 Raster Extraction

Now that you have generated the random steps and loaded the raster data, you can take the next step and actually connect the trajectories with the spatial covariates. There are different functions that can do this. When choosing one, you need to consider that raster files are large and juggling with them occupies lots of memory and computing power. For this reason we suggest using the `extract()` function that allows for querying single pixel values without loading the whole source file into working memory. The code for compiling the final dataset involves three steps: Converting the `xmpl.steps` data frame into a Spatial Points Data frame, extracting the raster values and combining them to the final dataset. Converting `xmpl.steps` into a SpatialPointsDataFrame:

```
xmpl.steps.spdf <- SpatialPointsDataFrame(coords = xmpl.steps[,c("new_x","new_y")], data = x
```

Extracting the values from each raster layer:

```
ruggedness.extr <- extract(ruggedness, xmpl.steps.spdf, method='simple', sp=F, df=T)
canopycover.extr <- extract(canopycover, xmpl.steps.spdf, method='simple', sp=F, df=T)
disthighway.extr <- extract(disthighway, xmpl.steps.spdf, method='simple', sp=F, df=T)
distroad.extr <- extract(distroad, xmpl.steps.spdf, method='simple', sp=F, df=T)
landcover.extr <- extract(landcover, xmpl.steps.spdf, method='simple', sp=F, df=T)
```

The extraction is done separately for each layer. The option `method = 'simple'` extracts value from nearest cell whereas `method = 'bilinear'` interpolates from the four nearest cells. You can adjust this option according to the resolution of your dataset and ecological considerations. `df=T` returns the result as a data frame and `sp=F` ensures that the output is not added to the original dataset right away.

Automatically adding the extracted values to the original dataset sounds like a handy option. For two reasons we do not use it here: Firstly, we want to set the column names manually for not ending up with several columns called "w001001". Secondly, our data include a categorial covariate (landcover) that we want to reclassify and flag as a factor.

This is the code for compiling the final dataset:

```
xmpl.steps.spdf$ruggedness <- ruggedness.extr[,2]
xmpl.steps.spdf$canopycover <- canopycover.extr[,2]
xmpl.steps.spdf$disthighway <- disthighway.extr[,2]
xmpl.steps.spdf$distroad <- distroad.extr[,2]

# The landcover covariate comes coded in integers between 0 an 10 and is by default (mis)in
unique(landcover.extr[,2])
# Re-classifying landcover:
xmpl.steps.spdf$landcover <- as.factor(
    ifelse(landcover.extr[,2] == 0,"NA",
    ifelse(landcover.extr[,2] < 5, "forest",
    ifelse(landcover.extr[,2] < 8, "open","NA"))))
```

Now your final dataset should be ready for analysis. Examine it:

```
head(xmpl.steps.spdf)
```

4 Final SSF Model

Using the function `cs.` we get comparable values for each predictor. The according equation is:

$$f(x) = (x - mean)/sd(x)$$

5 Acknowledgements

Don't forget to thank TeX and R and other opensource communities if you use their products! The correct way to cite R is shown when typing "`citation()`", and "`citation("mgcv")`" for packages.

Special thanks to ❤ ♥ Simone ♥♥♥ ❤❥✿! You were our best team member! ❀

Save Models!

6 Appendix

Session Info:

```
## R version 3.1.2 (2014-10-31)
## Platform: x86_64-redhat-linux-gnu (64-bit)
##
## locale:
##  [1] LC_CTYPE=de_DE.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=de_DE.UTF-8        LC_COLLATE=de_DE.UTF-8
##  [5] LC_MONETARY=de_DE.UTF-8    LC_MESSAGES=de_DE.UTF-8
##  [7] LC_PAPER=de_DE.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=de_DE.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
```

```
## [1] knitr_1.7
##
## loaded via a namespace (and not attached):
## [1] evaluate_0.5.5 formatR_1.0    highr_0.4      stringr_0.6.2  tools_3.1.2
```

References

[1] C. Calenge. Package adehabitatlt. 2014.

[2] Henrik Thurfjell, Simone Ciuti, and Mark S Boyce. Applications of step-selection functions in ecology and conservation. *Movement Ecology*, 2(1):4, 2014.