A tutorial for implementing Step Selection Function in R

P. Antkowiak* H. Tripke† C. Wilhelm‡

November 28, 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

1.1 Purpose and applications of SSFs

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 are connected by a linear segment. These segments are considered as steps. The fix rate of the GPS locations that influence the step length should be choosen carefully (i.e. by conducting a pilot study) to meet the requirements of the study questions, the target species and its behaviour. Then random steps are calculated by taking measured angle and distance along steps and using the observed positions as starting points. These alternative steps represent the available habitat, which could have been chosen, 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 [3].
So far, SSF models were mainly done using Geospatial Modelling Environment (GME) that works with a GIS[1]. Moreover, many packages for analyzing animal movements are provided in R. None of these packages is designed for doing a SSF only or is at least missing fundamental steps. For example, the `adehabitatLT` has many features for analyzing your telemetry data (note that reading the help from 2011 is very much recommendend by us [2]!) but is missing any function to convert random steps. Another problem we faced when exploring the available packages in R was that some of them are lacking detailed descriptions on how to use their examples. The data for these examples is well prepared but for an user, who is freshly starting with SSF, it is hard to understand the structure of these data.
Therefore, the aim of this tutorial is to collect all these functions necessary to conduct a SSF and order and describe 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"). Since the study design is fundemental to successfully implement SSFs, we recommend to read Thurfjell et al. (2014) [3], who provide different options and suggestions for conducting SSFs, before starting your own study.

1.2 Our SSF workflow in R

Figure 2 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 be transformed a couple of times to be able to work with them. You can find the single steps on the right site of Figure 2. While there are many options to adjust your *Waypoint data* the *Raster data* describing your spatial attributes needs not much of 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] www.spatialecology.com/gme/

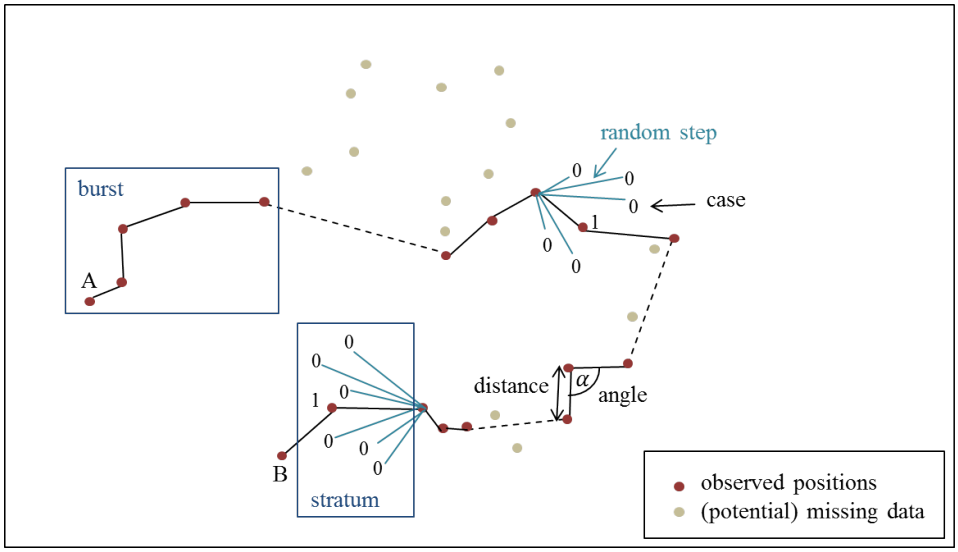

Figure 1: Animal movements in an environment. Theses data are already separated into bursts of not more than 3h 10min intervals. This prohibits to combine positions uncared of missing data points. These bursts provide the angle and distance to calculate random steps from. Thereby, strata and case have to be defined.

1.3 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. For faster processing the `install.packages` function is deactivated.

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


install.packages("adehabitatHR") # dealing with home ranges
install.packages("adehabitatHS") # habitat selection
install.packages("adehabitatLT") # trajectories
install.packages("adehabitatMA") # maps

install.packages("tkrplot")

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

# for analyzing the data
install.packages("mclogit")
install.packages("lme4")
install.packages("effects")
```

Keep fingers off the `adehabitat` package! It is outdated and replaced by four different packages

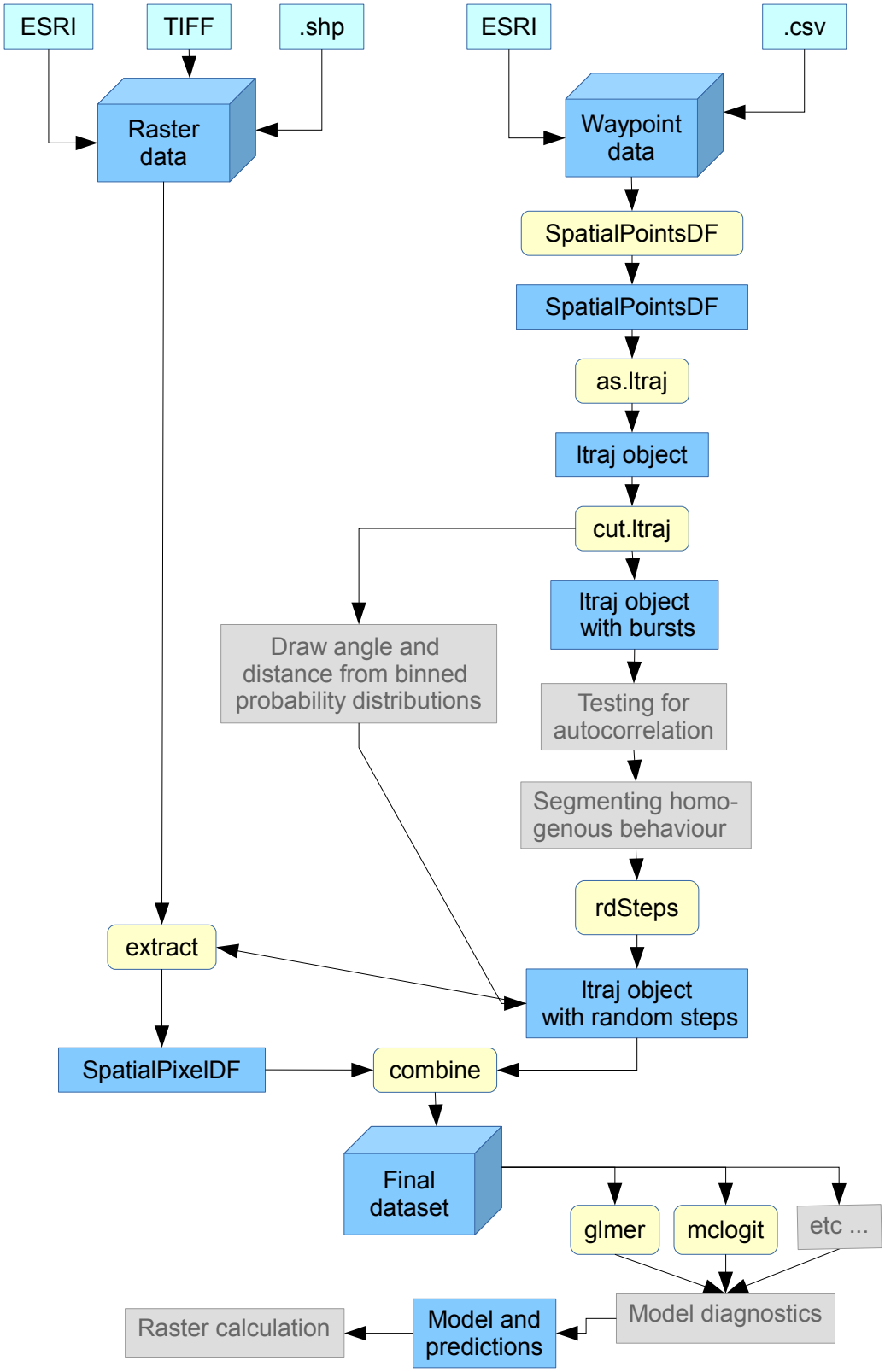

Figure 2: Stepwise conducting a Step Selection Function using existing R-packages. The yellow boxes show the name of the function applied while the blue boxes provide the type of object or data. In light grey optional steps are highlighted that are not implemented in this tutorial but should be considered by doing SSF.

designed for different analyses.
Loading packages:

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

library(sp)
library(raster)

library(mclogit)
library(lme4)
library(effects)
```

2 Processing the Waypoint Data

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 headings for each column in the first row and each observation should include at least the four following values:

- animal ID
- x-coordinate (easting)
- y-coordinate (northing)
- date and/or time

Remember 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:

- ID for each record
- GPS precision
- other recording parameters such as season or month
- temperature / elevation at the moment of record
- 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("P:/Henriette/BestPracticeR/SSF-workflow/Code")
xmpl = read.csv("UTMsREDUCED.csv", head=T)
```

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

2.2 Creating a "Spatial Points Data Frame"

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

```
library(sp)

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

In the `coords` argument of the function you should specify the "x" and "y" values 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.

To see, what information is stored in the data frame, use the `names` command:

```
names(xmpl)

##  [1] "cat"       "LINE_NO"   "GMT_DATE"  "GMT_TIME"  "LMT_DATE"  "LMT_TIME"
##  [7] "easting"   "northing"  "summer1"   "day1"      "hour"      "month"
```

2.3 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". There are two different types of trajectories. For **Type I** the time is not known or not taken into account. **Type II** is characterized by a timestamp. If the time lag between the different locations is the same, it is called "regular", if not "irregular". The function `as.ltraj` requires at least three arguments to work:

1. "x" and "y" (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, needs to be stored as a "POSIXct" value with date and time in the same cell. If it is not stored in the required format yet, 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:%S"))
```

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 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. The `id` creates subsets in your data set which you can see in the structure below.
You now may want to have a closer look at the created "ltraj" object. Display its structure by executing `str(xmpl.ltr)`. The "ltraj" object is a list containing seven different dataframes, each for one individual, which we defined with the `id`. In each dataframe all information is stored for every single observation of the individual:

- `x` and "y" (x- and y- coordinates for each point)
- `date` (timestamp for each point, given as POSIXct class)
- `dx` and "dy" (changes in x- and y- values)
- `dist` (length of trajectory)
- `dt` (time between relocations in s)
- `R2n` (squared net displacement between current relocation and first relocation)
- `abs.angle` (absolute angle of trajectory)
- `rel.angle` (angle between previous and actual trajectory)

```
str(xmpl.ltr)

## List of 7
##  $ :'data.frame': 1111 obs. of  10 variables:
##   ..$ x        : num [1:1111] 692362 692304 691745 691738 691743 ...
##   ..$ y        : num [1:1111] 5476589 5476529 5476031 5476042 5476035 ...
##   ..$ date     : POSIXct[1:1111], format: "2010-01-26 15:00:56" "2010-01-26 21:02:53" ...
##   ..$ dx       : num [1:1111] -57.79 -559.05 -7.28 4.68 2.45 ...
##   ..$ dy       : num [1:1111] -60.13 -497.63 10.99 -7.18 2.54 ...
##   ..$ dist     : num [1:1111] 83.4 748.45 13.18 8.57 3.53 ...
##   ..$ dt       : num [1:1111] 21717 21532 10767 10776 10775 ...
##   ..$ R2n      : num [1:1111] 0 6955 691594 688499 690582 ...
##   ..$ abs.angle: num [1:1111] -2.336 -2.414 2.156 -0.993 0.802 ...
##   ..$ rel.angle: num [1:1111] NA -0.0778 -1.7126 3.1338 1.795 ...
##   ..- attr(*, "id")= chr "10286"
##   ..- attr(*, "burst")= chr "10286"
##   ..- attr(*, "infolocs")='data.frame': 1111 obs. of  1 variable:
##   .. ..$ pkey: Factor w/ 7845 levels "10286.2010-01-26 15:00:56",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ :'data.frame': 933 obs. of  10 variables:
##   ..$ x        : num [1:933] 693628 693722 693638 693782 693789 ...
##   ..$ y        : num [1:933] 5484250 5484205 5484182 5484248 5484243 ...
##   ..$ date     : POSIXct[1:933], format: "2010-02-21 12:03:14" "2010-02-21 15:01:51" ...
##   ..$ dx       : num [1:933] 93.91 -84.48 144.27 6.78 -357.94 ...
##   ..$ dy       : num [1:933] -45.08 -22.48 66.34 -4.99 345.67 ...
##   ..$ dist     : num [1:933] 104.17 87.42 158.79 8.42 497.6 ...
##   ..$ dt       : num [1:933] 10717 21660 21482 10794 10839 ...
##   ..$ R2n      : num [1:933] 0 10852 4653 23627 25793 ...
##   ..$ abs.angle: num [1:933] -0.448 -2.882 0.431 -0.635 2.374 ...
##   ..$ rel.angle: num [1:933] NA -2.43 -2.97 -1.07 3.01 ...
##   ..- attr(*, "id")= chr "10287"
##   ..- attr(*, "burst")= chr "10287"
##   ..- attr(*, "infolocs")='data.frame': 933 obs. of  1 variable:
##   .. ..$ pkey: Factor w/ 7845 levels "10286.2010-01-26 15:00:56",..: 1112 1113 1114 1115 1116 1117 1118 1119 1120 1121 ...
##  $ :'data.frame': 812 obs. of  10 variables:
##   ..$ x        : num [1:812] 698168 698171 698172 698180 698002 ...
##   ..$ y        : num [1:812] 5488180 5488171 5488169 5488189 5489527 ...
##   ..$ date     : POSIXct[1:812], format: "2010-02-01 12:01:18" "2010-02-01 15:00:54" ...
##   ..$ dx       : num [1:812] 3.214 0.945 7.374 -177.826 4.936 ...
##   ..$ dy       : num [1:812] -8.676 -2.081 20.414 1337.288 -0.265 ...
##   ..$ dist     : num [1:812] 9.25 2.29 21.71 1349.06 4.94 ...
##   ..$ dt       : num [1:812] 10776 10802 10828 21653 10758 ...
##   ..$ R2n      : num [1:812] 0.00 8.56e+01 1.33e+02 2.26e+02 1.84e+06 ...
##   ..$ abs.angle: num [1:812] -1.216 -1.1444 1.2242 1.703 -0.0536 ...
##   ..$ rel.angle: num [1:812] NA 0.0716 2.3686 0.4788 -1.7566 ...
##   ..- attr(*, "id")= chr "10288"
##   ..- attr(*, "burst")= chr "10288"
##   ..- attr(*, "infolocs")='data.frame': 812 obs. of  1 variable:
##   .. ..$ pkey: Factor w/ 7845 levels "10286.2010-01-26 15:00:56",..: 2045 2046 2047 2048 2049 2050 2051 2052 2053 2054 ...
##  $ :'data.frame': 775 obs. of  10 variables:
##   ..$ x        : num [1:775] 725277 725254 724811 726203 730378 ...
##   ..$ y        : num [1:775] 5462266 5462033 5463762 5464355 5464921 ...
##   ..$ date     : POSIXct[1:775], format: "2010-03-09 18:01:53" "2010-03-09 21:00:47" ...
##   ..$ dx       : num [1:775] -23.4 -442.6 1392 4174.6 5401.3 ...
##   ..$ dy       : num [1:775] -233 1730 592 567 148 ...
##   ..$ dist     : num [1:775] 234 1785 1513 4213 5403 ...
##   ..$ dt       : num [1:775] 10734 54036 118772 21618 10805 ...
##   ..$ R2n      : num [1:775] 0 54825 2457311 5220574 33066643 ...
##   ..$ abs.angle: num [1:775] -1.6708 1.8213 0.4022 0.1349 0.0275 ...
##   ..$ rel.angle: num [1:775] NA -2.791 -1.419 -0.267 -0.107 ...
```

```
##   ..- attr(*, "id")= chr "10289"
##   ..- attr(*, "burst")= chr "10289"
##   ..- attr(*, "infolocs")='data.frame': 775 obs. of  1 variable:
##   .. ..$ pkey: Factor w/ 7845 levels "10286.2010-01-26 15:00:56",..: 2857 2858 2859 2860 2861 2862 2863 2864 2865 2866 ...
##  $ :'data.frame': 1638 obs. of  10 variables:
##   ..$ x        : num [1:1638] 727631 727789 727793 726560 726284 ...
##   ..$ y        : num [1:1638] 5447646 5447722 5447708 5445560 5445322 ...
##   ..$ date     : POSIXct[1:1638], format: "2010-03-01 15:01:20" "2010-03-01 18:00:58" ...
##   ..$ dx       : num [1:1638] 157.94 4.15 -1232.31 -276.79 112.8 ...
##   ..$ dy       : num [1:1638] 75.9 -13.9 -2148.2 -237.5 -134.8 ...
##   ..$ dist     : num [1:1638] 175.2 14.5 2476.6 364.7 175.7 ...
##   ..$ dt       : num [1:1638] 10778 10796 21694 10753 43289 ...
##   ..$ R2n      : num [1:1638] 0 30701 30119 5497537 7213751 ...
##   ..$ abs.angle: num [1:1638] 0.448 -1.28 -2.092 -2.433 -0.874 ...
##   ..$ rel.angle: num [1:1638] NA -1.728 -0.812 -0.341 1.559 ...
##   ..- attr(*, "id")= chr "10290"
##   ..- attr(*, "burst")= chr "10290"
##   ..- attr(*, "infolocs")='data.frame': 1638 obs. of  1 variable:
##   .. ..$ pkey: Factor w/ 7845 levels "10286.2010-01-26 15:00:56",..: 3632 3633 3634 3635 3636 3637 3638 3639 3640 3641 ...
##  $ :'data.frame': 1026 obs. of  10 variables:
##   ..$ x        : num [1:1026] 732538 732540 733902 733793 733834 ...
##   ..$ y        : num [1:1026] 5454593 5454585 5455217 5456784 5457345 ...
##   ..$ date     : POSIXct[1:1026], format: "2010-03-10 18:01:49" "2010-03-10 21:01:48" ...
##   ..$ dx       : num [1:1026] 1.39 1362.2 -109.5 41.85 -4.68 ...
##   ..$ dy       : num [1:1026] -8.63 632.89 1566.31 561.62 -10.11 ...
##   ..$ dist     : num [1:1026] 8.74 1502.05 1570.13 563.18 11.14 ...
##   ..$ dt       : num [1:1026] 10799 21576 10746 10914 10697 ...
##   ..$ R2n      : num [1:1026] 0.00 7.63e+01 2.25e+06 6.37e+06 9.25e+06 ...
##   ..$ abs.angle: num [1:1026] -1.412 0.435 1.641 1.496 -2.004 ...
##   ..$ rel.angle: num [1:1026] NA 1.846 1.206 -0.144 2.783 ...
##   ..- attr(*, "id")= chr "10291"
##   ..- attr(*, "burst")= chr "10291"
##   ..- attr(*, "infolocs")='data.frame': 1026 obs. of  1 variable:
##   .. ..$ pkey: Factor w/ 7845 levels "10286.2010-01-26 15:00:56",..: 5270 5271 5272 5273 5274 5275 5276 5277 5278 5279 ...
##  $ :'data.frame': 1550 obs. of  10 variables:
##   ..$ x        : num [1:1550] 703848 704820 703803 704148 703890 ...
##   ..$ y        : num [1:1550] 5470822 5466573 5467903 5468040 5468453 ...
##   ..$ date     : POSIXct[1:1550], format: "2010-03-19 15:03:05" "2010-03-20 18:03:08" ...
##   ..$ dx       : num [1:1550] 971.8 -1017.4 345.5 -258.2 -27.5 ...
##   ..$ dy       : num [1:1550] -4248 1330 137 412 204 ...
##   ..$ dist     : num [1:1550] 4358 1674 372 487 206 ...
##   ..$ dt       : num [1:1550] 97203 43191 32334 10859 10675 ...
##   ..$ R2n      : num [1:1550] 0 18993292 8521412 7826708 5614599 ...
##   ..$ abs.angle: num [1:1550] -1.346 2.224 0.378 2.13 1.705 ...
##   ..$ rel.angle: num [1:1550] NA -2.713 -1.846 1.752 -0.426 ...
##   ..- attr(*, "id")= chr "10293"
##   ..- attr(*, "burst")= chr "10293"
##   ..- attr(*, "infolocs")='data.frame': 1550 obs. of  1 variable:
##   .. ..$ pkey: Factor w/ 7845 levels "10286.2010-01-26 15:00:56",..: 6296 6297 6298 6299 6300 6301 6302 6303 6304 6305 ...
##  - attr(*, "class")= chr [1:2] "ltraj" "list"
##  - attr(*, "typeII")= logi TRUE
##  - attr(*, "regular")= logi FALSE
```

To work with one of the tables only you can use `[ ]`. This is still an "ltraj" object. For example:

```
xmpl.ltr[2]

##
## *********** List of class ltraj ***********
##
## Type of the traject: Type II (time recorded)
## Irregular traject. Variable time lag between two locs
##
## Characteristics of the bursts:
##       id burst nb.reloc NAs           date.begin             date.end
## 1 10287 10287      933   0 2010-02-21 12:03:14 2010-09-03 09:01:26
##
##
##  infolocs provided. The following variables are available:
## [1] "pkey"
```

If you want to have a look at all the data stored for one single individual, or "use" it as a data frame you can use `[[ ]]`. If you assign it to a name, it is automatically converted and ready to use:

```
xmpl.df.10286 = xmpl.ltr[[1]]
```

To get a visual impression of your data you can plot the trajectory for all or for one particular animal:

```
plot(xmpl.ltr)
```

The result of this code chunk is shown in Figure 3.

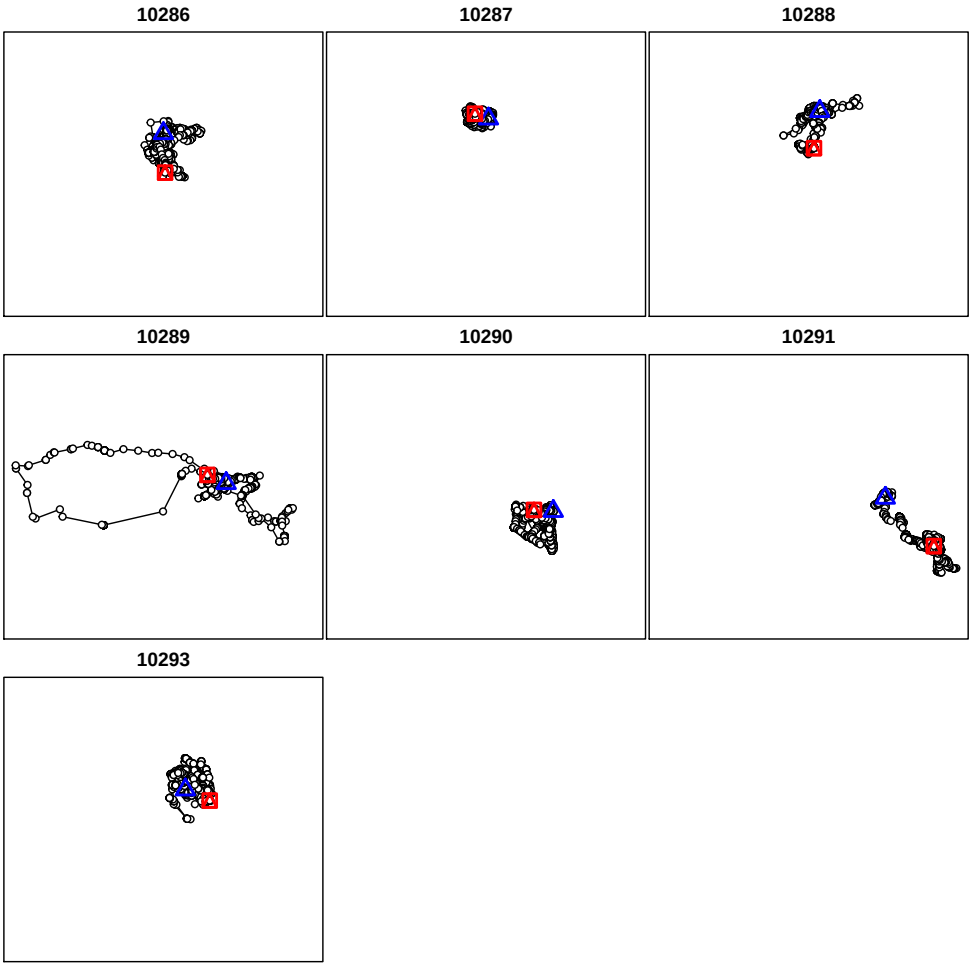

Figure 3: Visualization of the observed positions and the path from seven collared cougars.

To get a list of all cat IDs, use the `unique` command. Choose one that you are interested in!

```
unique(xmpl.spdf$cat)

## [1] 10286 10287 10288 10289 10290 10291 10293
```

You can then look at the trajectory of a single individual only.

```
plot(xmpl.ltr, id=10289)
```

The result of this code chunk is shown in Figure 4.

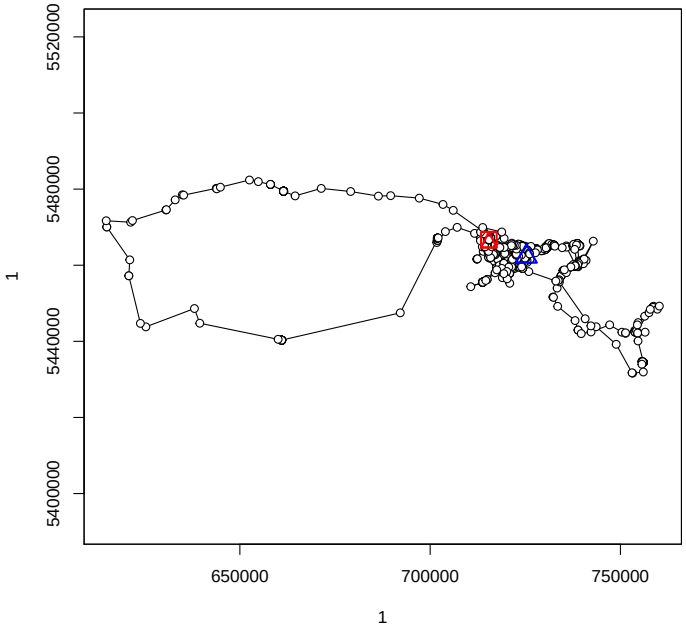

Figure 4: Visualization of the observed positions and the path from only one individual.

2.4 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. It is also possible to split your data into homogenous behaviour. Suggestions how to do that are explained in one of the following sections.
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 missing values and to get an overview of the time lags between the relocations, you can plot the changes of your trajectory over time `dt`. 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. As `dt`, the time between successive relocations, is measured in seconds, we need to convert it into 3 hours.

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

The result of this code chunk is shown in Figure 5.
The relocations which are 3 hours apart, are plottet at value 1 on the y-axis, if one relocation is missing (time lag of 6 hours) the relocation gets the value 2. As you can see, there are a lot of relocations missing. To keep the time lag constant, we now want to split the existing bursts (individuals) into "sub-bursts" where the time lag is bigger than 3 hours. 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.

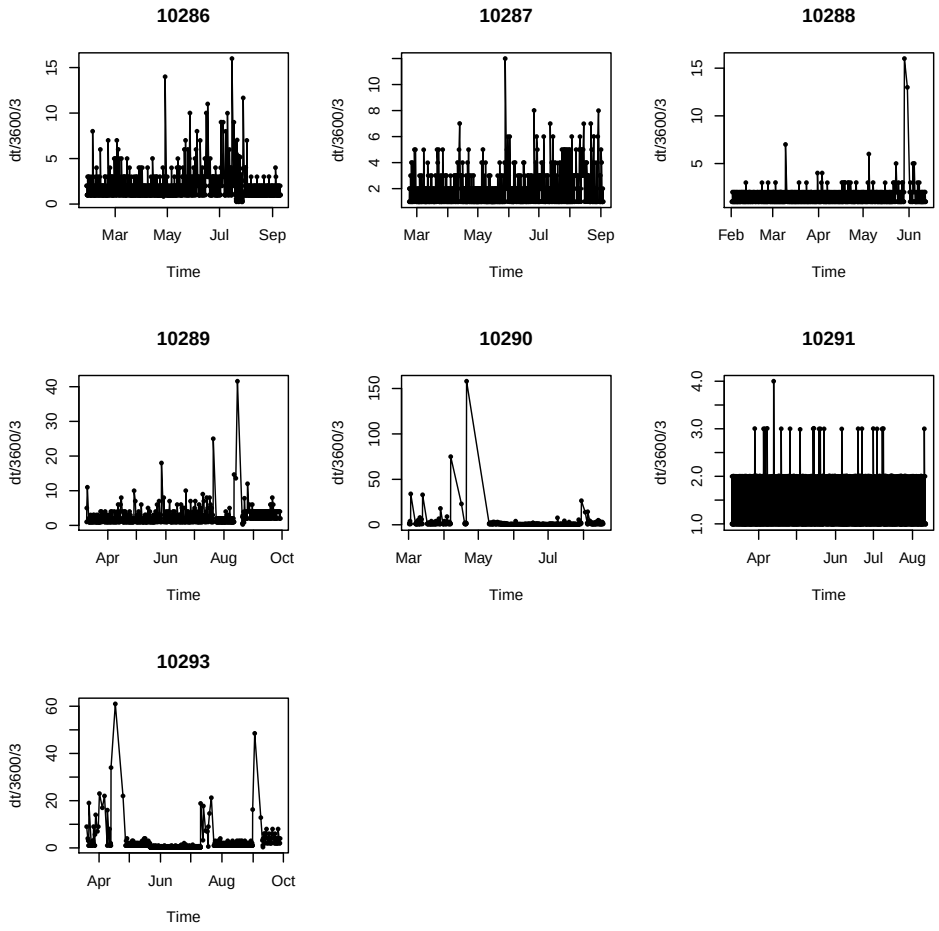

Figure 5: For each individual the observations are plotted. The y-axis shows the time interval from one observation to the next. Observations at 1 means that they are not more than 3 hours apart.

```
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)

## Warning in cutltraj(xmpl.ltr, "foo(dt)", nextr = TRUE): At least 3
relocations are needed for a burst
##  1249 relocations have been deleted
```

Note: We still keep data with different time lags (but not more than 3:10 h). As this is an irregular type II trajectory, it might cause problems with different functions (e.g. checking for autocorrelation). A constant time lag is necessary to have comparable trajectories for further analyses. There are functions to convert it into a regular one, look at the `adehabitatLT` tutorial by C. Calenge [1], the help section to get more information or define an explicit time lag in the function above. A nice way to take care of our problem with minor delays of the relocations, which might be caused by the time the GPS needs to relocate, is to round. You can first look for missing time values and replace them via setting a reference date. After that you can

round time values which slightly differ from the desired one using the function `sett0`. This however should NOT be used to convert a completely irregular trajectory into a regular one but compensate for minor descrepancies. Detailed instructions on how to do that can also be found in the totorial by Calenge. [1]

In general 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]

2.5 "ltraj" diagnostics

After defining your " bursts", there are already some options to check your created "ltraj" object. Those are described in the tutorial "Analysis of Animal Movements in R: the `adehabitatLT` Package" by clement Calenge [2]. As our dataset was revised already and missing values have been removed, we decided not to interpolate, we seperated all sequences according to the time lag. We therefore only give suggestions what could have been done additionaly but might result in a better output depending on your research question.

2.5.1 Viewing missing values

Depending on the wheather, the location of the animal or technical reasons, there might be missing relocations in your dataset. A first option is therefore to check for your "ltraj" object for missing values. A simple function helps you here:

```
adehabitatLT:::plotNAltraj(xmpl.ltr[1])
```

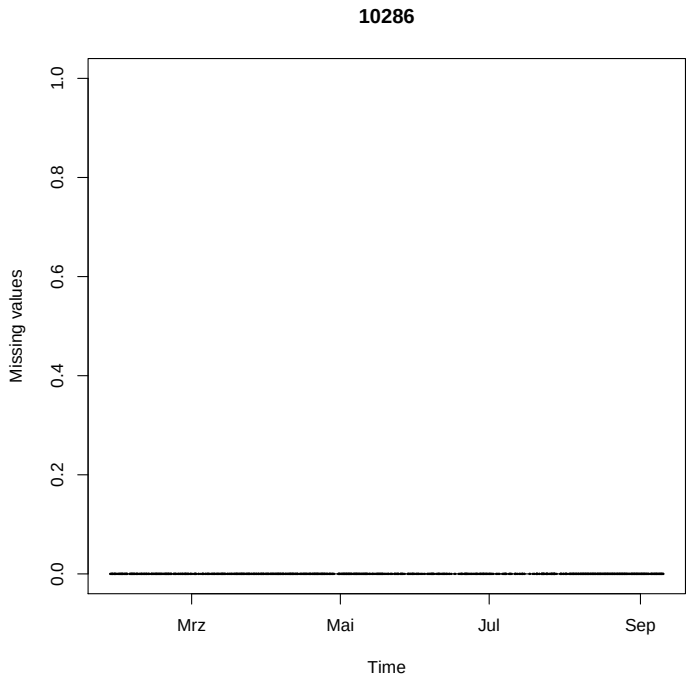

Figure 6: Showing all NAs in your dataset. In our case missing values have been removed already.

This plot shows where your dataset has missing values but as our data does not include any NAs, there is only one straight line.

2.5.2 Replace and round time

A first step for interpolating your dataset would be replacing the NAs in your `date` column by creating new ones with the `setNA` function. For that, you need to define a reference date. By doing that, you create a lot of "relocations" with NAs in the coordinates column.

The next step would be to round time values which slightly differ from the desired one using the function `sett0`. This however should NOT be used to convert a completely irregular trajectory into a regular one but compensate for minor descrepancies.

2.5.3 Testing for Autocorrelation

To "understand" the movement patterns of an animal, is it essential to check the parameters stored in the trajectory (dist, dx, dy, angle) for autocorrelation. A positive autocorrelation means, that values closer to each other tend to be more similar.

To test for autocorrelation, the ltraj object needs to be of type I or consist of constant time lags. As our time lags are not constant, we transform our trajectory into type I by using a simple `typeII2typeI` function.

```
xmpl.ltr.t1 <- typeII2typeI(xmpl.cut)
```

The new format does not include any information concerning time and date:

```
head(xmpl.ltr.t1)

##
## *********** List of class ltraj ***********
##
## Type of the traject: Type I (time not recorded)
##
## Characteristics of the bursts:
##      id     burst Nb.reloc NAs
## 1 10286 10286.003        6   0
## 2 10286 10286.004        7   0
## 3 10286 10286.008        3   0
## 4 10286 10286.009        3   0
## 5 10286 10286.010        7   0
## 6 10286 10286.012        3   0
##
##
##  infolocs provided. The following variables are available:
## [1] "pkey"
```

To test the three linear parameters (`dist, dx, dy`), the independence test of Wald and Wolfowitz (1994) can be used. It tests the sequential autocorrelation in a vector. It can be implemented as `wawotest.ltraj` for each burst in a `ltraj` object. This function removes all NAs before running the test [2].

```
wawotest(xmpl.ltr.t1)
```

The functions returns parameters to test for correlation for every burst (827 in our case!, this is why we did not let it run). The p-Values indicate the correlation but as our time lag is not constant, the interpretation does not make much sense.

To identify at which scales autocorrelation occurs, an autocorrelation function (ACF) can be used. Here, the ltraj object needs to be regular. Similar tests can be done for angular parameters. For further information check out the tutorial by Clement Calenge [2].

2.5.4 Distinguish different behaviors

It is possible to divide a trajectory into segments characterized by a homogenous behaviour. You can either use a "Broken Stick Model", the method of "Gueguen" or the method of "Lavielle" [2].

2.6 Creating Random Steps

angles and distances from your observed data to get random steps. The angle is taken from the previous trajectory to the actual one while the distance is taken from the starting point to the next observed position. This means that each burst should at least contain three observed positions to create random steps for one relocation. The first and the last relocations only contribute to the calculation but don't have their "own" random steps as there is no previous trajectory an angle could be calculated from. Neither there is an endpoint to draw a distance. This means that the number of poins with random steps is lower as your actual recorded relocations. but first have to convert our "ltraj" object back to a data frame by using `ld`.
Now that your telemetry data are cleaned and split into bursts, you can proceed to the next big step: Drawing the random steps. The aim of this chapter is to generate realistic alternative steps that the animal might have taken so you can later compare them with the actually visited location. The "hab" package includes the very useful `rdSteps` function that can read data frames with spatial trajectories, generate random steps from a given distribution and add those steps to the data frame. We will demonstrate the usage of this function with all required arguments and give hints to some extended options. The `rdSteps` function reads the following arguments:

- **x** = your trajectory data
  The data can be a ltraj object or a data frame. The function `ld` is a quick way to create a data frame from an ltraj object:

```
xmpl.cut.df <- ld(xmpl.cut)
```

- **nrs** = the number of random steps to draw for each observed step
  Choosing the number of random steps to generate is certainly not the most important decision when running an SSF, but it is not completely arbitrary either. The number of alternative steps needs to be sufficiently high for producing meaningful results, especially if your species of interest depends on rather rare habitats scattered over the landscape. On the other hand, large numbers of random points can inflate your dataset and lead to long computing times without really changing the results [3]. In previous studies, the number of random steps ranged from 2 to 200. As our xmpl dataset is fairly large, we chose 10 - a medium number of random steps.

- **rand.dis** = the random distributions for step lengths and turning angles
  Per default (`rand.dis = NULL`) the angle and distance for each random step are drawn from the observed values you provide in `x`. We use this handy option because it saves us the trouble to create distributions separately. For some reasons you might still choose to use different distributions: If your dataset is small, using the observed values might lead to circularities. You might also already have good distributions from other studies and want use them. To import distributions from other datasets, set `rand.dist = YourDataSet`.

- **only.others** = include or exclude the current individual when drawing length and angle

- **simult** = draw step lengths and turning angles simultaneously or not
  Before setting this parameter, you might want to check for correletaion between turning angle and distance. Such might occur if your individuals tend to turn only in small angles when moving long distances (e.g. a species migrating) but on the other hand turn a lot and move slowly when feeding. If distance and angle correlate, you want to draw them as pairs. If no correlation is found you can pick both variables independently. To check for correlation, simply `plot` distance versus angle (code below, result shown in Figure 7).

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

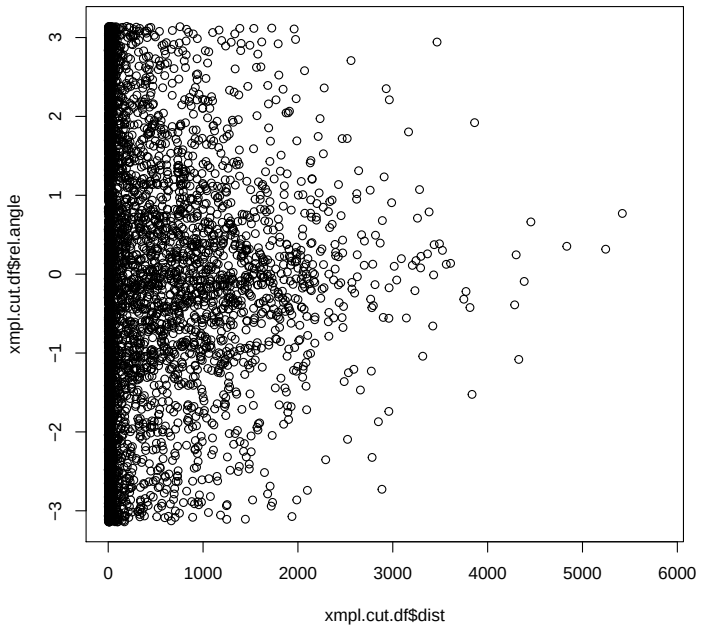

Figure 7: Testing for correlation of the observed turning angle and step length. The y-axis shows the difference in the turning angle.

The distribution of points indicates a correlation: For small turning angles (around 1) steps are longer than for big turning angles. To take into account for this, we will draw angle and step length simultaneously (`simult = T`).

- **distMax** = set an upper limit for step length and turning angle
  We stick to infinity as the default value here to include all steps.

- **reproducible** = make results reproducible or not.
  By choosing reproducible = TRUE we set a seed to get reproducible random steps.

For the example dataset run the `rdSteps` function as follows:

```
xmpl.steps <- rdSteps(x = xmpl.cut, nrs = 10, simult = T, rand.dis = NULL,
                      distMax = Inf, reproducible = TRUE, only.others = FALSE)
```

All in all, `rdSteps` is a very powerful function that computes a lot of useful things for you. To view them, execute:

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

```
##                  x       y                date          dx         dy
## 10286.2   691738.1 5476042 2010-01-27 06:01:12    4.681700   -7.18200
## 10286.2.1 691738.1 5476042 2010-01-27 06:01:12  105.850281 -134.39670
## 10286.2.2 691738.1 5476042 2010-01-27 06:01:12   -2.562687  -86.12695
## 10286.2.3 691738.1 5476042 2010-01-27 06:01:12 -124.866698  675.96404
## 10286.2.4 691738.1 5476042 2010-01-27 06:01:12   85.786965   69.73609
## 10286.2.5 691738.1 5476042 2010-01-27 06:01:12 -280.215683 -448.38129
##                dist    dt      R2n  abs.angle  rel.angle    id     burst
## 10286.2     8.573181 10776 173.7518 -0.9931106  3.1337881 10286 10286.003
## 10286.2.1 171.075294 10776 173.7518 -7.1868501 -3.0599515 10286 10286.003
## 10286.2.2  86.165064 10776 173.7518 -1.6005423  2.5263563 10286 10286.003
## 10286.2.3 687.400232 10776 173.7518 -4.5297242 -0.4028256 10286 10286.003
## 10286.2.4 110.555529 10776 173.7518 -5.6006288 -1.4737302 10286 10286.003
## 10286.2.5 528.740587 10776 173.7518 -2.1293594  1.9975393 10286 10286.003
##                                 pkey case strata
## 10286.2   10286.2010-01-27 06:01:12    1      1
## 10286.2.1 10286.2010-01-27 06:01:12    0      1
## 10286.2.2 10286.2010-01-27 06:01:12    0      1
## 10286.2.3 10286.2010-01-27 06:01:12    0      1
## 10286.2.4 10286.2010-01-27 06:01:12    0      1
## 10286.2.5 10286.2010-01-27 06:01:12    0      1
```

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" marks all 10 random steps and the visited location so you can later tell your function what to compare. Now 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
```

After running this chapter you get your final "SpatialPointDataFrame" to use with the selection function.

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

```
##                  x       y                date          dx         dy
## 10286.2   691738.1 5476042 2010-01-27 06:01:12    4.681700   -7.18200
## 10286.2.1 691738.1 5476042 2010-01-27 06:01:12  105.850281 -134.39670
## 10286.2.2 691738.1 5476042 2010-01-27 06:01:12   -2.562687  -86.12695
## 10286.2.3 691738.1 5476042 2010-01-27 06:01:12 -124.866698  675.96404
## 10286.2.4 691738.1 5476042 2010-01-27 06:01:12   85.786965   69.73609
## 10286.2.5 691738.1 5476042 2010-01-27 06:01:12 -280.215683 -448.38129
##                dist    dt      R2n  abs.angle  rel.angle    id     burst
## 10286.2     8.573181 10776 173.7518 -0.9931106  3.1337881 10286 10286.003
## 10286.2.1 171.075294 10776 173.7518 -7.1868501 -3.0599515 10286 10286.003
## 10286.2.2  86.165064 10776 173.7518 -1.6005423  2.5263563 10286 10286.003
## 10286.2.3 687.400232 10776 173.7518 -4.5297242 -0.4028256 10286 10286.003
## 10286.2.4 110.555529 10776 173.7518 -5.6006288 -1.4737302 10286 10286.003
## 10286.2.5 528.740587 10776 173.7518 -2.1293594  1.9975393 10286 10286.003
##                                 pkey case strata    new_x   new_y
## 10286.2   10286.2010-01-27 06:01:12    1      1 691742.7 5476035
## 10286.2.1 10286.2010-01-27 06:01:12    0      1 691843.9 5475908
```

```
## 10286.2.2 10286.2010-01-27 06:01:12    0      1 691735.5 5475956
## 10286.2.3 10286.2010-01-27 06:01:12    0      1 691613.2 5476718
## 10286.2.4 10286.2010-01-27 06:01:12    0      1 691823.8 5476112
## 10286.2.5 10286.2010-01-27 06:01:12    0      1 691457.8 5475594
```

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. 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:

- ruggedness of the terrain
- land cover
- canopy cover
- distance to the nearest highway
- distance to the nearest road

For reading the raster data, three packages are required:

```
library(raster)
library(rgdal)
library(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 or set your path:
# getwd()

ruggedness <- raster("P:/SSF PROJECT/NEW GIS LAYERS/tri1")
landcover <- raster("P:/SSF PROJECT/NEW GIS LAYERS/lc_30")
canopycover <- raster("P:/SSF PROJECT/NEW GIS LAYERS/cc_abmt")
disthighway <- raster("P:/SSF PROJECT/NEW GIS LAYERS/disthwy")
distroad <- raster("P:/SSF PROJECT/NEW GIS LAYERS/distsmrd")

# It is enough to load the whole folder were your *.adf files are
# stored in. The function raster() finds the needed files itself.
```

You can plot the data for a first overview. As this can take a while with large datasets, we outcommented the following chunk.

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

3.2 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 observed and potential points 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
- 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 = xmpl.steps)
```

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)interpreted as an integer string.

unique(landcover.extr[,2])

##  [1]  3  2  1  6  8  7  0  9  4 10  5

# 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)

##                  x       y                date          dx         dy
## 10286.2   691738.1 5476042 2010-01-27 06:01:12    4.681700   -7.18200
## 10286.2.1 691738.1 5476042 2010-01-27 06:01:12  105.850281 -134.39670
## 10286.2.2 691738.1 5476042 2010-01-27 06:01:12   -2.562687  -86.12695
## 10286.2.3 691738.1 5476042 2010-01-27 06:01:12 -124.866698  675.96404
## 10286.2.4 691738.1 5476042 2010-01-27 06:01:12   85.786965   69.73609
## 10286.2.5 691738.1 5476042 2010-01-27 06:01:12 -280.215683 -448.38129
##                dist    dt      R2n  abs.angle  rel.angle    id     burst
## 10286.2     8.573181 10776 173.7518 -0.9931106  3.1337881 10286 10286.003
## 10286.2.1 171.075294 10776 173.7518 -7.1868501 -3.0599515 10286 10286.003
## 10286.2.2  86.165064 10776 173.7518 -1.6005423  2.5263563 10286 10286.003
## 10286.2.3 687.400232 10776 173.7518 -4.5297242 -0.4028256 10286 10286.003
## 10286.2.4 110.555529 10776 173.7518 -5.6006288 -1.4737302 10286 10286.003
## 10286.2.5 528.740587 10776 173.7518 -2.1293594  1.9975393 10286 10286.003
##                              pkey case strata    new_x   new_y ruggedness
## 10286.2   10286.2010-01-27 06:01:12    1      1 691742.7 5476035  11.704700
## 10286.2.1 10286.2010-01-27 06:01:12    0      1 691843.9 5475908  15.198684
## 10286.2.2 10286.2010-01-27 06:01:12    0      1 691735.5 5475956  18.027756
## 10286.2.3 10286.2010-01-27 06:01:12    0      1 691613.2 5476718  16.522711
## 10286.2.4 10286.2010-01-27 06:01:12    0      1 691823.8 5476112  11.832160
## 10286.2.5 10286.2010-01-27 06:01:12    0      1 691457.8 5475594   5.744563
##           canopycover disthighway distroad landcover
## 10286.2            63    1738.189 210.0000    forest
## 10286.2.1          72    1612.762 330.0000    forest
## 10286.2.2          37    1745.680 271.6616    forest
## 10286.2.3          76    2204.949 361.2478    forest
## 10286.2.4          63    1718.139 120.0000    forest
## 10286.2.5          76    1749.286 180.0000    forest
```

3.3 Possible extensions to the extract function

When extracting the environmental covariates, the most straightforward approach is to measure them at the endpoint of a step, just as we did in our example. For some ecological questions

it might though be desirable to analyze the selection of habitat structures along the step as well (see the grey box in Figure 2). Some animals might for example not visit a suitable patch of habitat because it is surrounded by unfavorable or even impassable structures. Fortunately, the `extract` function is capable of not just extracting values from single points but also along lines. The input data need to be "SpatialLines" features which means that you would first have to convert the "SpatialPoints" from your dataset. Suitable functions are included in the "sp" package. The extracted values could then be accumulated (e.g. terrain ruggedness), averaged or checked for criteria (e.g. "Does the step cross a cell classified as road?"). Implementing this feature might be a good aim for follow-up projects.

3.4 Checking for Multicollinearity

Before including all environmental factors in your analysis, you should check if two or even more variables are exact or highly correlated. The threshold for the correlation coefficiant is 0.7 or higher. To create a correlation matrix, we first need to convert the "Spatial Points Data Frame" into a data frame. Furthermore the column `landcover` needs to be changed from "factor" into "numeric".

```
xmpl.steps.df <- as.data.frame(xmpl.steps.spdf)

Z <- subset(xmpl.steps.df, select=c(ruggedness,canopycover,disthighway,distroad))

Z$landcover <- as.numeric(
    ifelse(xmpl.steps.df$landcover == "forest",0,
    ifelse(xmpl.steps.df$landcover == "open", 1,NA)))

#head(Z)

cor(Z, use="pairwise.complete.obs")

##             ruggedness canopycover disthighway    distroad   landcover
## ruggedness  1.00000000  0.04038767  0.19524678  0.18801050  0.05078682
## canopycover 0.04038767  1.00000000  0.11754087  0.14915085 -0.89037976
## disthighway 0.19524678  0.11754087  1.00000000  0.60060231 -0.05192774
## distroad    0.18801050  0.14915085  0.60060231  1.00000000 -0.06204133
## landcover   0.05078682 -0.89037976 -0.05192774 -0.06204133  1.00000000
```

Just to get a better overview we created a nice plot by using the following function.

```
panel.cor <- function(x, y, digits = 2, prefix = "", cex.cor, ...)
{
    usr <- par("usr"); on.exit(par(usr))
    par(usr = c(0, 1, 0, 1))
    r <- abs(cor(x, y,use="pairwise.complete.obs"))
    txt <- format(c(r, 0.123456789), digits = digits)[1]
    txt <- paste0(prefix, txt)
    if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
    text(0.5, 0.5, txt, cex = cex.cor * r)
}

panel.hist <- function(x, ...)
{
    usr <- par("usr"); on.exit(par(usr))
```

```
    par(usr = c(usr[1:2], 0, 1.5) )
    h <- hist(x, plot = FALSE)
    breaks <- h$breaks; nB <- length(breaks)
    y <- h$counts; y <- y/max(y)
    rect(breaks[-nB], 0, breaks[-1], y, col = "cyan", ...)
}
```

```
pairs(Z, lower.panel=panel.smooth,
      upper.panel=panel.cor,diag.panel=panel.hist)
```

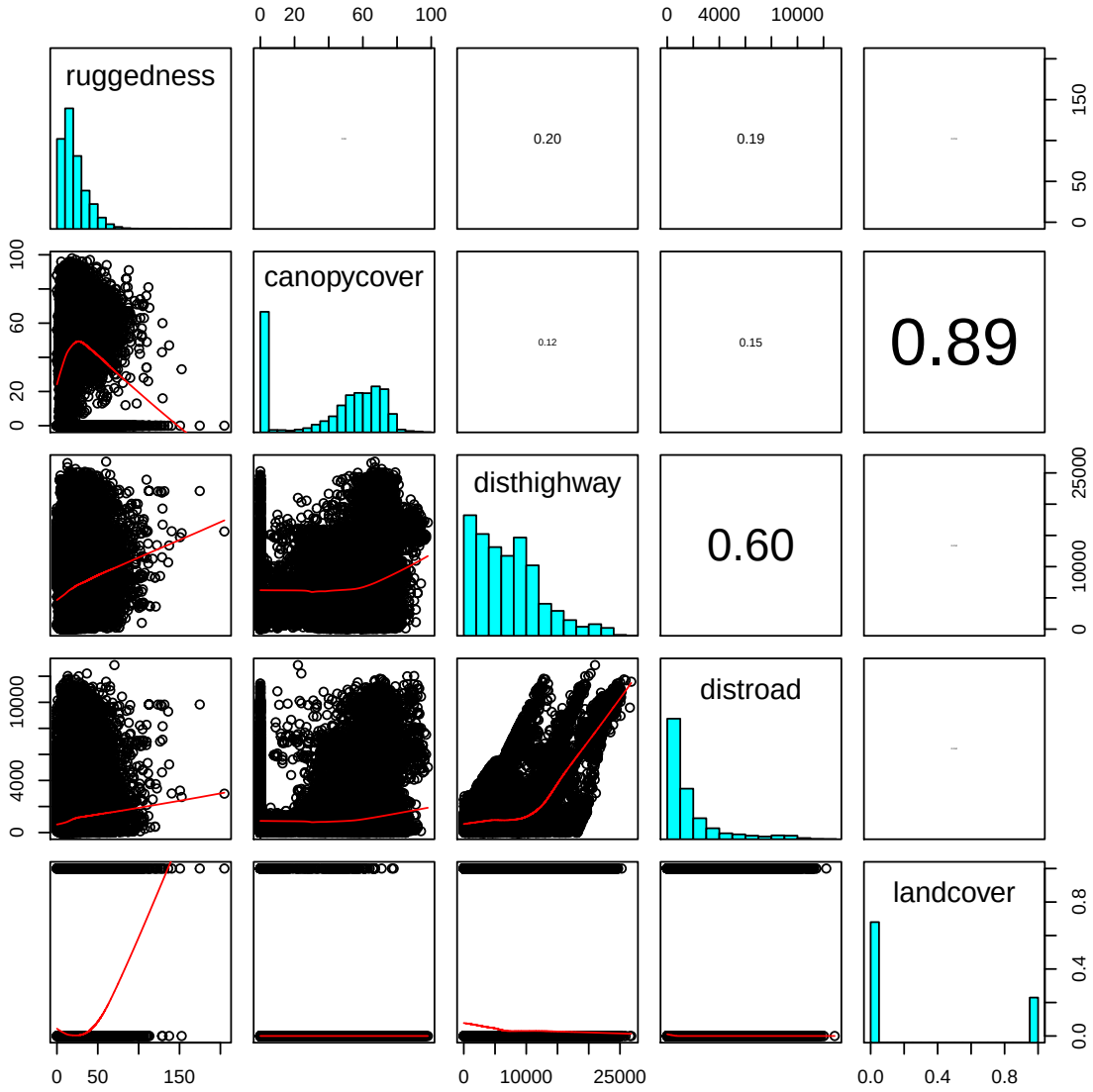

Figure 8: Beautiful plot to examine correlations and the structure of each parameter. The lower panel shows the scatter plots for each combination, the upper panel provides the correlation coefficients and the diagonal panel shows a histogram for each parameter.

As you can see, the canopycover and the landcover are highly correlated. The correlation coefficient for the distance to roads and the distance to highways is very high as well. For our further

analysis we therefore only use landcover and the distance to roads.

4 Final SSF Model

There are several options to analyze the generated data. As Thurfjell et al. 2014 [3] describe in their review of SSFs, conditional logistic regression has been the most commonly used procedure. Recently, researchers have tried to account for among-individual heterogeneity in their dataset. Thurfjell et al. 2014 [3] recommend two R packages providing the necessary functionality: The `lme4` package and the `mclogit` package. For giving two simple examples, we will demonstrate data analysis with a mixed conditional logistic regression `(mclogit)` and a generalized linear mixed model `(lme4)`. We will test for effects of ruggedness, canopy cover, land cover, distance to road and distance to highway. The geographic covariates will be included as quadratic terms, resulting in the equation:

$$case\ ruggedness + ruggedness^2 + disthighway + disthighway^2 + distroad + distroad^2 + landcover$$

Both models should produce rather similar outputs. Certainly, you are free to implement any type of model you find more elaborate or more suitable.
In a second step, we will use one of the fitted models to generate predictions and plot them.

4.1 Generalized Linear Mixed Model

family binomial (with a binomial distribution of error) nested random effect for ID and strata ...where the random effect takes the form of (1—id/strata).
structure of pseudo replication: id, stratum

```
xmpl.glmm.fit <- glmer(case ~
    landcover +
    ruggedness + I(ruggedness^2) +
    distroad + I(distroad^2) +
    (1|id/strata),
    family = binomial, data=xmpl.steps.df)
```

```
summary(xmpl.glmm.fit)

## Error in summary(xmpl.glmm.fit): error in evaluating the argument 'object' in
selecting a method for function 'summary': Error: object 'xmpl.glmm.fit' not found
```

The model does not run and we get error messages suggesting us to scale the variables. Using the function `scale()` we center and scale the data so we get comparable values for each predictor. The according equation is:

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

```
library(lme4)
library(effects)

xmpl.glmm.fit.sc <- glmer(case ~
    landcover +
    scale(ruggedness) + I(scale(ruggedness)^2) +
    scale(distroad) + I(scale(distroad)^2) +
    (1|id/strata),
    family = binomial, data=xmpl.steps.df)
```

```
summary(xmpl.glmm.fit.sc)

## Generalized linear mixed model fit by maximum likelihood (Laplace
##   Approximation) [glmerMod]
##  Family: binomial  ( logit )
## Formula: case ~ landcover + scale(ruggedness) + I(scale(ruggedness)^2) +
##     scale(distroad) + I(scale(distroad)^2) + (1 | id/strata)
##    Data: xmpl.steps.df
##
##      AIC      BIC   logLik deviance df.resid
##  32715.2  32786.3 -16349.6  32699.2    53401
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -0.3346 -0.3296 -0.3255 -0.2955  6.8536
##
## Random effects:
##  Groups    Name        Variance Std.Dev.
##  strata:id (Intercept) 0.00e+00 0.000e+00
##  id        (Intercept) 1.59e-13 3.988e-07
## Number of obs: 53409, groups:  strata:id, 4940; id, 7
##
## Fixed effects:
##                          Estimate Std. Error z value Pr(>|z|)
## (Intercept)            -2.215426   0.022127 -100.12  < 2e-16 ***
## landcoveropen          -0.216703   0.035502   -6.10 1.03e-09 ***
## scale(ruggedness)       0.011641   0.021362    0.54  0.58580
## I(scale(ruggedness)^2) -0.029234   0.010682   -2.74  0.00621 **
## scale(distroad)        -0.025928   0.030423   -0.85  0.39408
## I(scale(distroad)^2)    0.005637   0.011432    0.49  0.62191
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##             (Intr) lndcvr scl(r) I(scl(r)^2) scl(d)
## landcoverpn -0.338
## scl(rggdns)  0.179  0.012
## I(scl(r)^2) -0.365 -0.082 -0.633
## scal(dstrd)  0.397  0.043 -0.252  0.092
## I(scl(d)^2) -0.486 -0.012  0.187 -0.073      -0.861
```

```
plot(allEffects(xmpl.glmm.fit.sc))
```

4.2 Mixed Conditional Logistic Regression

The conventional logistic regression models do not take into account that in wildlife telemetry data observations are not independent but rather linked. If for example the selection differs among individual animals, the model may be flawed by pseudo replication. The mixed conditional logistic regression allows for specifying a random effects structure and thereby is capable of handling matched observations from different animals. Here is how to implement it in R:

```
library(mclogit)
# Trasform dataset to a data frame
xmpl.steps.df <- as.data.frame(xmpl.steps.spdf)
```

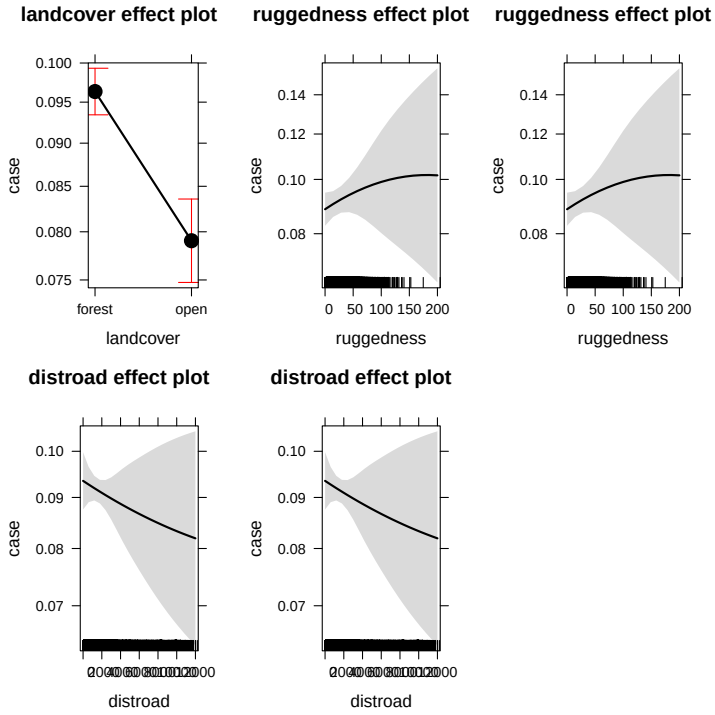

Figure 9: Effects plot of the GLMM.

```
# Recall the column names of the dataset:
#head(xmpl.steps.df)

#xmpl.steps.spdf$id <- as.integer(xmpl.steps.spdf$id)

# column "case" indicates whether a site was visited or not.
# "strata" indicates the burst number and "id" the individual animal.

# The actual logistic model:
xmpl.logit.fit <-
    mclogit(cbind(case, strata) ~
    landcover +
    ruggedness + I(ruggedness^2) +
    distroad + I(distroad^2), random=~1|id,
    data = xmpl.steps.df, start.theta=1000)

##
## Iteration 1 - Deviance = 23240.16
## Iteration 2 - Deviance = 23239.18
## Iteration 3 - Deviance = 23239.18
## Iteration 4 - Deviance = 23239.18
## converged
##
## Initial estimate of theta:  1000
## Iteration 1 - Deviance = 23239.18 criterion =  0
## converged

## when including random effects we run into problems.
# We get the error message that the data do not have enough residual variance.
```

```
#random=~1|id,
#disthighway + I(disthighway^2),
```

```
summary(xmpl.logit.fit)

##
## Call:
## mclogit(formula = cbind(case, strata) ~ landcover + ruggedness +
##     I(ruggedness^2) + distroad + I(distroad^2), data = xmpl.steps.df,
##     random = ~1 | id, start.theta = 1000)
##
##                    Estimate Std. Error z value Pr(>|z|)
## landcoveropen   -3.345e-01  4.425e-02  -7.559 4.07e-14 ***
## ruggedness       4.464e-03  3.936e-03   1.134   0.2567
## I(ruggedness^2) -1.326e-04  5.436e-05  -2.439   0.0147 *
## distroad        -1.290e-04  6.094e-05  -2.117   0.0342 *
## I(distroad^2)   -3.542e-09  8.579e-09  -0.413   0.6797
## Var(id)          5.000e+02  1.313e+07   0.732   0.4643
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Null Deviance:     23340
## Residual Deviance: 23240
## Number of Fisher Scoring iterations:  1
## Number of observations:
##   (942 observations deleted due to missingness)
```

4.3 Predictions

Formula

$$w(x) = exp(\beta 1 * x1 + \beta 2 * x2 + ... + \beta p * xp)$$

To see the results of your analysis you have to write final predictions of the habitat selection. We first provide the code for each model and then show you the plot.

```
#plot(predict(xmpl.glmm.fit.sc, type="response"))
summary(xmpl.glmm.fit.sc)

## Generalized linear mixed model fit by maximum likelihood (Laplace
##   Approximation) [glmerMod]
##  Family: binomial  ( logit )
## Formula: case ~ landcover + scale(ruggedness) + I(scale(ruggedness)^2) +
##     scale(distroad) + I(scale(distroad)^2) + (1 | id/strata)
##    Data: xmpl.steps.df
##
##      AIC      BIC   logLik deviance df.resid
##  32715.2  32786.3 -16349.6  32699.2    53401
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -0.3346 -0.3296 -0.3255 -0.2955  6.8536
##
```

```
## Random effects:
##  Groups    Name        Variance Std.Dev.
##  strata:id (Intercept) 0.00e+00 0.000e+00
##  id        (Intercept) 1.59e-13 3.988e-07
## Number of obs: 53409, groups:  strata:id, 4940; id, 7
##
## Fixed effects:
##                         Estimate Std. Error z value Pr(>|z|)
## (Intercept)            -2.215426   0.022127 -100.12  < 2e-16 ***
## landcoveropen          -0.216703   0.035502   -6.10 1.03e-09 ***
## scale(ruggedness)       0.011641   0.021362    0.54  0.58580
## I(scale(ruggedness)^2) -0.029234   0.010682   -2.74  0.00621 **
## scale(distroad)        -0.025928   0.030423   -0.85  0.39408
## I(scale(distroad)^2)    0.005637   0.011432    0.49  0.62191
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##             (Intr) lndcvr scl(r) I(scl(r)^2) scl(d)
## landcoverpn -0.338
## scl(rggdns)  0.179  0.012
## I(scl(r)^2) -0.365 -0.082 -0.633
## scal(dstrd)  0.397  0.043 -0.252  0.092
## I(scl(d)^2) -0.486 -0.012  0.187 -0.073      -0.861

mean(xmpl.steps.df$ruggedness)

## [1] 21.39603

sd(xmpl.steps.df$ruggedness)

## [1] 15.22119

summary(scale(xmpl.steps.df$ruggedness))

##        V1
##  Min.   :-1.4057
##  1st Qu.:-0.7389
##  Median :-0.2735
##  Mean   : 0.0000
##  3rd Qu.: 0.4773
##  Max.   :12.0550

rm(mydata)

## Warning in rm(mydata): object 'mydata' not found

mydata = data.frame(ruggedness=seq(-1.5,15,0.1))
#mydata = data.frame(ruggedness=scale(xmpl.steps.df$ruggedness))

mydata$wruggforest = exp(
             xmpl.glmm.fit.sc@beta[3] * mydata$ruggedness  +
             xmpl.glmm.fit.sc@beta[4] * mydata$ruggedness^2 +
```

```
              xmpl.glmm.fit.sc@beta[5] * median(scale(xmpl.steps.df$distroad)) +
              xmpl.glmm.fit.sc@beta[6] * median((scale(xmpl.steps.df$distroad))^2))

mydata$wruggopen = exp(
              xmpl.glmm.fit.sc@beta[2] +
              xmpl.glmm.fit.sc@beta[3] * mydata$ruggedness  +
              xmpl.glmm.fit.sc@beta[4] * mydata$ruggedness^2 +
              xmpl.glmm.fit.sc@beta[5] * median(scale(xmpl.steps.df$distroad)) +
              xmpl.glmm.fit.sc@beta[6] * median((scale(xmpl.steps.df$distroad))^2))


#plot(mydata£ruggedness, mydata£wruggforest, type="l", col="darkgreen", lwd=2)
#lines(mydata£ruggedness, mydata£wruggopen, type="l", col="green", lwd=2)

## rescaling


mydata$ruggednessResc <- (mydata$ruggedness * sd(xmpl.steps.df$ruggedness) + mean(xmpl.steps


#head(mydata)
#str(mydata)
#mydata£ruggednessResc
```

```
plot(mydata$ruggednessResc, mydata$wruggforest,
     main="GLMM predictions for ruggedness", type="l",
     col="darkgreen", lwd=2, ylim=c(0,1), xlim=c(0,250),
     xlab="ruggedness", ylab="W", las=1)
lines(mydata$ruggednessResc, mydata$wruggopen, type="l", col="green", lwd=2)
```

4.3.1 Predictions for MCLOGIT

```
summary(xmpl.steps.df$ruggedness)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00   10.15   17.23   21.40   28.66  204.90

summary(xmpl.logit.fit)

##
## Call:
## mclogit(formula = cbind(case, strata) ~ landcover + ruggedness +
##     I(ruggedness^2) + distroad + I(distroad^2), data = xmpl.steps.df,
##     random = ~1 | id, start.theta = 1000)
##
##                   Estimate Std. Error z value Pr(>|z|)
## landcoveropen   -3.345e-01  4.425e-02  -7.559 4.07e-14 ***
## ruggedness       4.464e-03  3.936e-03   1.134   0.2567
## I(ruggedness^2) -1.326e-04  5.436e-05  -2.439   0.0147 *
## distroad        -1.290e-04  6.094e-05  -2.117   0.0342 *
```

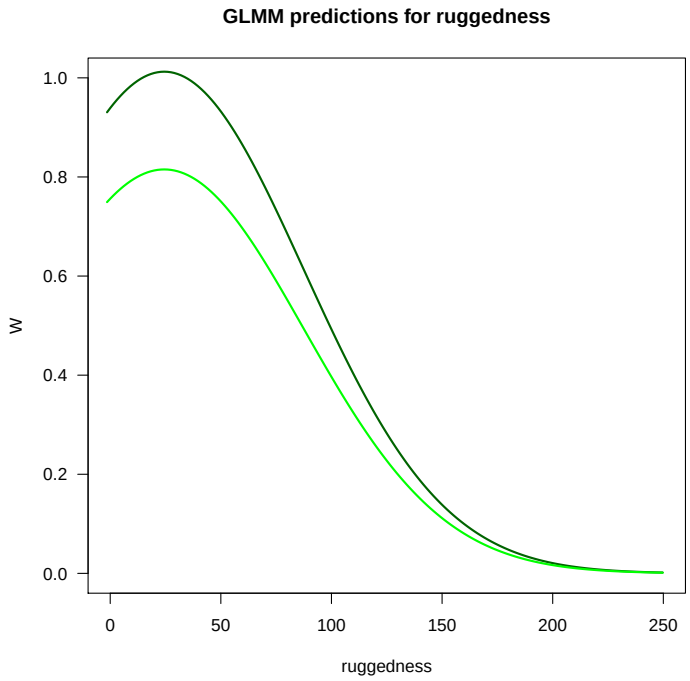

Figure 10: Prediction from the GLMM.

```
## I(distroad^2)   -3.542e-09  8.579e-09  -0.413   0.6797
## Var(id)          5.000e+02  1.313e+07   0.732   0.4643
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Null Deviance:     23340
## Residual Deviance: 23240
## Number of Fisher Scoring iterations:  1
## Number of observations:
##   (942 observations deleted due to missingness)

#plot(predict(xmpl.logit.fit))

mydata = data.frame(ruggedness=seq(0,250,1))

mydata$wruggforest = exp(xmpl.logit.fit$coefficients[2] * mydata$ruggedness  +
                xmpl.logit.fit$coefficients[3] * mydata$ruggedness^2 +
                xmpl.logit.fit$coefficients[4] * median(xmpl.steps.df$distroad) +
                xmpl.logit.fit$coefficients[5] * (median(xmpl.steps.df$distroad))^2)

mydata$wruggopen = exp(xmpl.logit.fit$coefficients[1] +
                xmpl.logit.fit$coefficients[2] * mydata$ruggedness  +
                xmpl.logit.fit$coefficients[3] * mydata$ruggedness^2 +
                xmpl.logit.fit$coefficients[4] * median(xmpl.steps.df$distroad) +
                xmpl.logit.fit$coefficients[5] * (median(xmpl.steps.df$distroad))^2)
```

```
plot(mydata$ruggedness,mydata$wruggforest,
     main="MCLOGIT predictions for ruggedness",
     type="l", col="darkgreen", lwd=2, ylim=c(0,1),
     xlab="ruggedness", ylab="W", las=1)
lines(mydata$ruggedness,mydata$wruggopen,type="l", col="green", lwd=2)
```

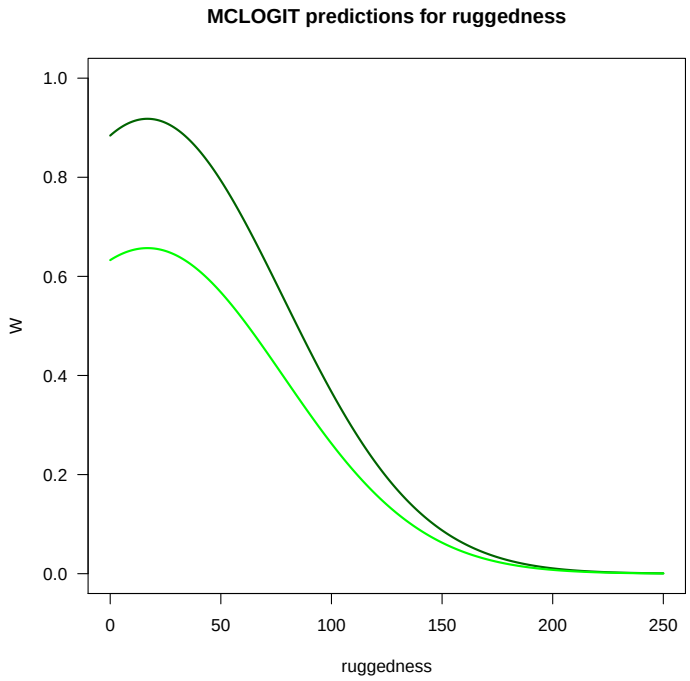

Figure 11: Prediction from the MCLOGIT.

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 3.1.1 (2014-07-10)
## Platform: i386-w64-mingw32/i386 (32-bit)
##
## locale:
## [1] LC_COLLATE=German_Germany.1252  LC_CTYPE=German_Germany.1252
## [3] LC_MONETARY=German_Germany.1252 LC_NUMERIC=C
## [5] LC_TIME=German_Germany.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] rgdal_0.9-1         effects_3.0-2       lme4_1.1-7
##  [4] Rcpp_0.11.3         mclogit_0.3-1       Matrix_1.1-4
##  [7] raster_2.3-12       adehabitatHS_0.3.9  hab_1.17
## [10] adehabitatHR_0.4.11 deldir_0.1-6        adehabitatLT_0.3.16
## [13] CircStats_0.2-4     boot_1.3-11         MASS_7.3-33
## [16] ade4_1.6-2          adehabitatMA_0.3.8  sp_1.0-15
## [19] knitr_1.7
##
## loaded via a namespace (and not attached):
##  [1] colorspace_1.2-4 evaluate_0.5.5   formatR_1.0      grid_3.1.1
##  [5] highr_0.4        lattice_0.20-29  minqa_1.2.4      nlme_3.1-117
##  [9] nloptr_1.0.4     nnet_7.3-8       parallel_3.1.1   pbkrtest_0.4-2
## [13] splines_3.1.1    stringr_0.6.2    tools_3.1.1
```

References

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

[2] Clement Calenge. Analysis of animal movements in r. 2011.

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