# A Rough Guide to Pre-processing High-Frequency Animal Tracking Data

Pratik R. Gupte and others

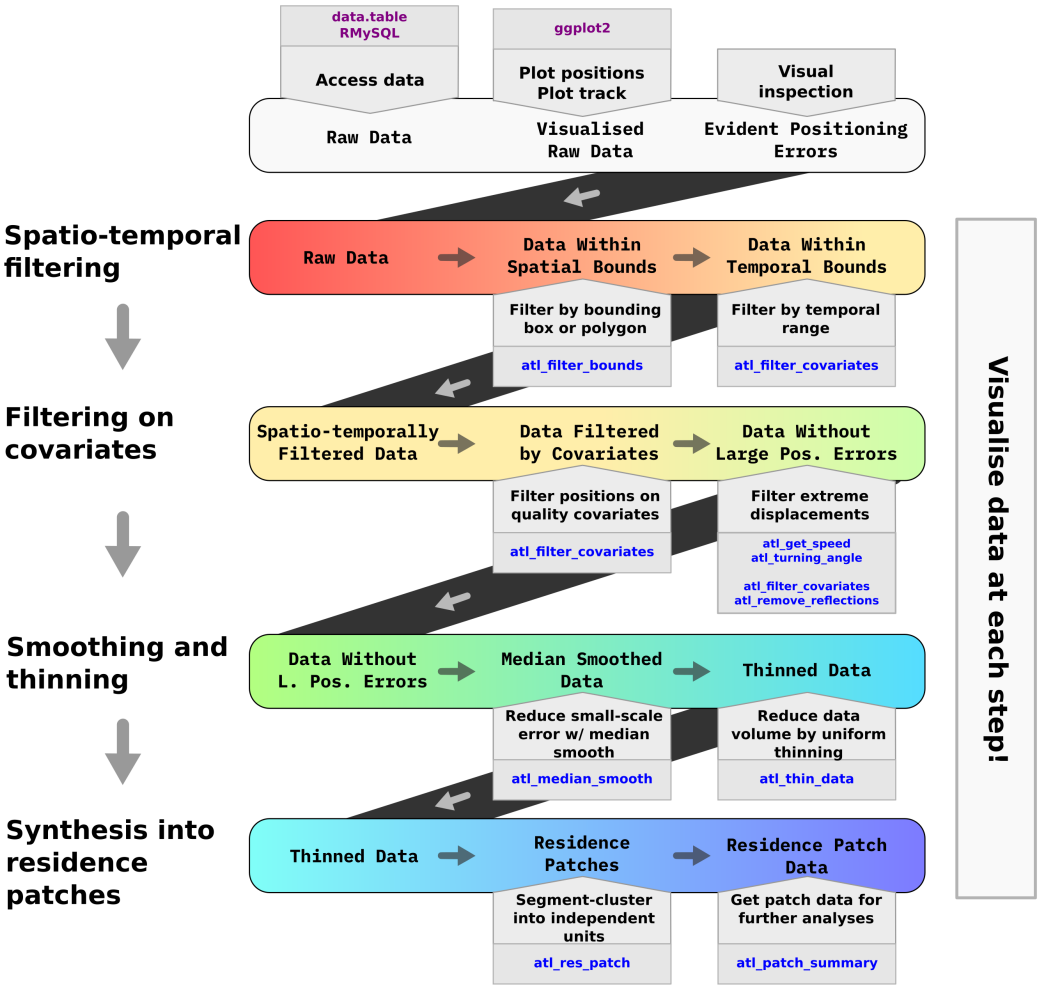
2020-10-29

# Contents

# 1 Abstract

Data cleaning is a ubiquitous pre-processing step in analysis pipelines, and its automation is essential for large data volumes such as those generated in animal tracking studies using high-frequency Time-of-Arrival (TOA) systems. Users of systems such as ATLAS must contend with two intertwined data cleaning challenges: (1) reducing positioning errors, and (2) the high volume of data itself. Making biological inferences from data with positioning errors is not straightforward, and processing large data volumes is computationally intensive. Though reducing positioning error is widely recommended, users are without uniform guidance on how to go about this, and lack a common set of computationally efficient tools. Further, many methods that synthesize movement tracks for ecological inference are either (1) not suited to very large datasets, or (2) not intuitive to understand in terms of the tracked animal's biology.

In this article we introduce a pipeline to pre-process high-frequency animal tracking data in order to prepare it for subsequent analysis. We demonstrate this pipeline on simulated movement data to which we have randomly added positioning errors. This pipeline is suited to any tracking study in which the high data volume combined with knowledge of the tracked individuals' biology can be used to smooth out positioning errors. We further suggest how large volumes of cleaned data may be synthesized into biologically meaningful 'residence patches', and demonstrate how this accurately captures animal space-use. Finally, we introduce the `R` package `atlastools` which provides fast implementations of the methods we describe. Though aimed at ATLAS systems, `atlastools` can be used with any time-series animal movement data, and we demonstrate its usage with both simulated and empirical examples.


data.table
RMySQL
Access data
ggplot2
Plot positions
Plot track
Visual
inspection
Raw Data
Visualised
Raw Data
Evident Positioning
Errors
Spatio-temporal
filtering
Raw Data
Data Within
Spatial Bounds
Data Within
Temporal Bounds
Filter by bounding
box or polygon
atl_filter_bounds
Filter by temporal
range
atl_filter_covariates
Filtering on
covariates
Spatio-temporally
Filtered Data
Data Filtered
by Covariates
Data Without
Large Pos. Errors
Filter positions on
quality covariates
atl_filter_covariates
Filter extreme
displacements
atl_get_speed
atl_turning_angle
atl_filter_covariates
atl_remove_reflections
Smoothing and
thinning
Data Without
L. Pos. Errors
Median Smoothed
Data
Thinned Data
Reduce small-scale
error w/ median
smooth
atl_median_smooth
Reduce data
volume by uniform
thinning
atl_thin_data
Synthesis into
residence
patches
Thinned Data
Residence
Patches
Residence Patch
Data
Segment-cluster
into independent
units
atl_res_patch
Get patch data for
further analyses
atl_patch_summary
Visualise data at each step!

# 2 Getting started

This section covers:

1. Installing the `R` package `atlastools`,
2. Simulating some realistic looking movement data using the `R` package `smoove` from (Gurarie et al. 2017), and
3. Introducing positioning errors into the simulated movement data.

## 2.1 Installing `atlastools`

This paper refers extensively to the `R` package `atlastools` (Gupte 2020), which can be installed from Github.

Releases of the package from Github can be found on Zenodo: (https://doi.org/10.5281/zenodo.4033155)

The code chunk below shows how to install `atlastools`.

```
# use either devtools or remotes to install
install.packages("devtools")

# installation using devtools
devtools::install_github("pratikunterwegs/atlastools")
```

## 2.2 Simulating movement data

Here, we simulate some movement data using the `smoove` `R` package from (Gurarie et al. 2017).

First, we load `smoove` and `data.table`, as well as some helper functions that make use of them to simulate data for use.

```
# load smoove and datatable
library(smoove)
library(data.table)

# source helper functions
source("R/helper_functions.R")

data <- do_smoove_data()
# save simulated data
fwrite(data, "data/data_sim.csv")
```

## 2.3 Simualte for residence patches

```r
# do smoove data using a RACVM
data <- smoove::simulateRACVM(dt = 0.1, Tmax = 500, omega = 5, v0 = 1, mu = 0.1)
data <- as.data.table(data$XY)

# assign id
data[, id := "test"]
data[, time := seq_len(nrow(data))]
# rename x and y
setnames(data, old = c("x", "y"), new = c("y", "x"))

# save data
fwrite(data, "data/data_for_res_patch.csv")
```

# 3 Reducing Large-Scale Positioning Error by Filtering Data

## 3.1 Prepare Libraries

Here we load some useful libraries, and the helper functions.

```
# to handle movement data
library(data.table)
library(atlastools)

# to plot
library(ggplot2)
library(patchwork)

# source helper functions
source("R/helper_functions.R")
```

## 3.2 Introducing Errors to the Data

Here we introduce three kinds of errors to the data:

1. Small-scale normally distributed errors at each position;
2. Large-scale error at a random 0.5% of positions;
3. A large-scale displacement of a sequence of 300 positions.

While the data are 10,000 positions at 1-second interval, we shall use only 5,000 of these.

```
# read in the data
data <- fread("data/data_sim.csv")[5000:10000, ]
```

We add outliers at random to the data to demonstrate their removal.

```
# make a copy
data_copy <- copy(data)

# add a prolonged spike or reflection to 300 positions
data_copy[500:800, `:=`(x = x + 0.25,
                        y = y + 0.25)]

# add normal error
data_copy[, `:=`(x = do_add_error(x, std_dev = 0.01),
                 y = do_add_error(y, std_dev = 0.005))]
```

```
# add 100 outliers
data_copy <- do_add_outliers(data_copy, p_data = 0.005, std_dev = 0.1)
```

Save the data to which errors have been added.

```
fwrite(data_copy, file = "data/data_errors.csv")
```

```
data_copy <- fread("data/data_errors.csv")
```

Define a palette with 4 colours for convenience.

```
# define a four colour palette
pal <- RColorBrewer::brewer.pal(4, "Set1")
```

We make a figure of the canonical data (grey line) along with the artificially added error (grey points).

## 3.3 Filtering by Spatial Bounds

Filtering by spatial bounds is a good way to begin reducing gross, large-scale positioning errors. There are two main ways of doing this:

1. Filtering by a bounding box: Compare the coordinates of observations against a range of acceptable coordinates, and retain those which fall within the range, or
2. Filtering by a spatial polygon: An explicit geometric intersection between the positions and a polygon representing an area of interest.

Here, we remove gross positioning errors using a bounding box filter using the function `atl_filter_bounds`. In this example, we show filtering only on the Y coordinate.

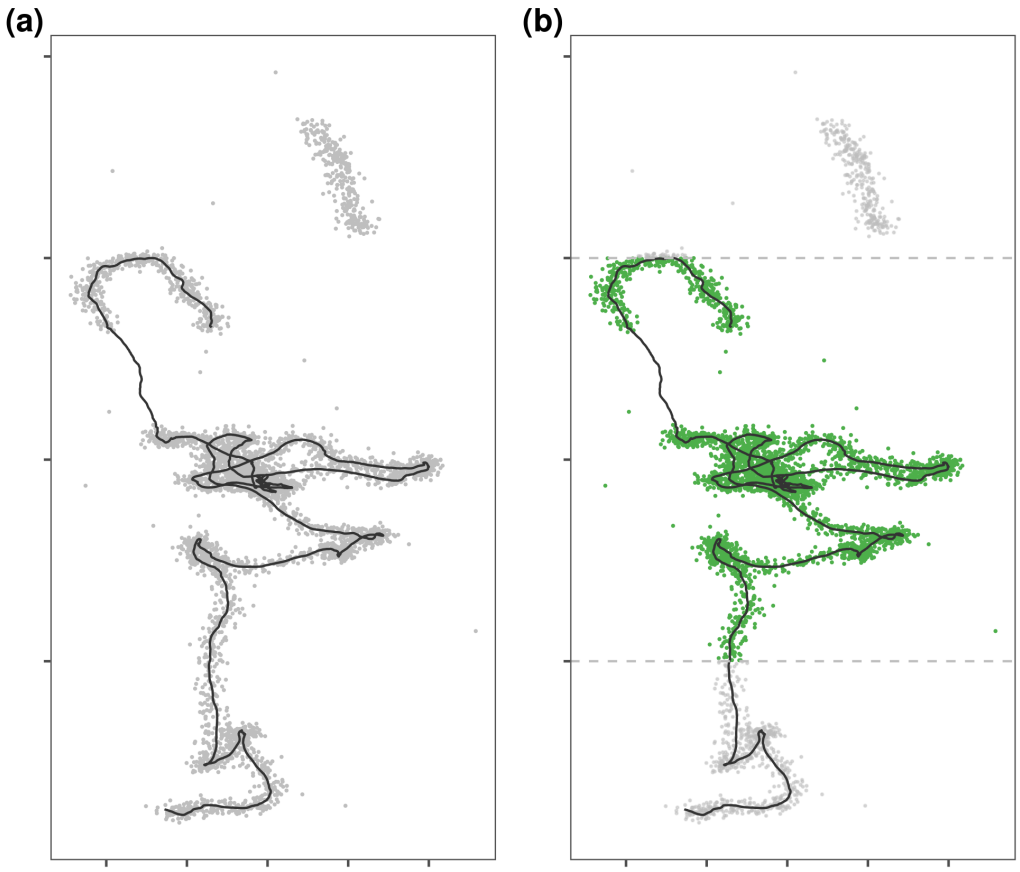
`atl_filter_bounds` takes coordinate ranges as two-element vectors of the lower and higher bound. It is possible to pass one of the bounds as `NA`, in which case, only the other bound is used for filtering, i.e., `y_range = c(NA, 1)` is equivalent to selecting all Y coordinates $< 1$.

`atl_filter_bounds` was initially designed to remove positions inside a specific range, hence the argument `remove_inside`. The default value of the argument is `FALSE`, and `atl_filter_bounds` is thus a bounding box filter.

```
# remove positions outside a bounding box
# NB: set remove_inside to FALSE
data_inside_bbox <- atl_filter_bounds(data = data_copy,
                                      y_range = c(0.5, 1),
                                      remove_inside = FALSE)
```

`atl_filter_bounds` is not vectorised, and if there are two or more bounds per coordinate (for instance, $X_1 \ldots X_2$, and $X_3 \ldots X_4$), they must be passed in two different function calls. The same is true for filtering by an `sf` polygon. `atl_filter_bounds` also supports filtering by multi-polygon objects.

Having filtered the data we shall prepare it for plotting.




(a) A velocity-autocorrelated movement track simulated for 5,000 positions (black line) using the `smoove` package (Gurarie et al. 2017). Three kinds of errors have been artificially added: (1) each position (grey points) is offset from the canonical track with the addition of normally distributed small-scale error, (2) large-scale error has been added to 0.5% of positions, and (3) 300 positions (indices 500 – 800) have been displaced to the top-right of the track to simulate a gross distortion that affects a continuous subset of the track. The goal of pre-processing such datasets is to get the estimated positions (grey points) to match the canonical track (solid black line) as closely as possible. (b) Tracks can be quickly filtered by spatial bounds ($0.5 \leq Y \leq 1.0$; dashed grey lines) using the `atlastools` function `atl_filter_bounds`. Setting the function argument `remove_inside = FALSE` retains positions within user supplied bounds (blue points), and excludes those outside (red points).

### 3.3.1 A Note on Filtering by Bounds

The filter on spatial bounds is only for demonstration, and is not applied to the data.

## 3.4 Filtering Unrealistic Movement

Large-scale positioning error can affect both point locations as well as entire subsets of a track, leading to a track appearing to show unrealistic movement for the study species (Bjørneraas et al. 2010).

The best way to examine whether a track contains such positioning errors is to plot the data, and especially to join the dots, i.e., to connect the positions with lines rather than simply plot points.

Users with extensive experience of their study system will readily recognise segments of a track where the movement appears to be unrealistic.

Briefly, the two main kinds of errors are mentioned here, and named based on their appearance in a tack: those that affect single positions lend a spiky appearance to a track, and are referred to as point outliers or spikes, while errors affecting continuous subsets of a track cause it to appear as though it has been reflected along a plane, and are hence called reflections, or prolonged spikes.

### 3.4.1 Filtering Point Outliers or Spikes

We begin by removing unrealistic movement in the form of point outliers, or spikes.

The first step is to determine how spikes should be identified. The non-movement approach prescribes determining whether movement metrics associated with each position are realistic or not, and targeting those positions where the movement metrics are unrealistic for the study species.

In this case we shall calculate only two metrics, speed and turning angle. These are conveniently implemented in `atlastools` using the functions `atl_get_speed` and `atl_turning_angle`.

```
# get speed and turning angle
data_copy[, `:=`(in_speed = atl_get_speed(data_copy,
                                          type = "in"),
                 out_speed = atl_get_speed(data_copy,
                                           type = "out"),
                 angle = atl_turning_angle(data_copy))]
```

Having calculated speed and turning angle, the next step is to remove positions with extremely high incoming and outgoing speeds. This means using the `atl_filter_covariates` function to remove positions with speed $\geq$ a plausible speed threshold. The use of a turning angle filter is optional, and not necessary in this case.

One approach is to define a speed cutoff based on expert knowledge, and this is best suited to well studied species.

For simulated data, there is no plausible speed threshold from prior knowledge — a reasonable choice here is to use the 90^th or 95^th percentile of speed and turning angle (when this metric is used).

This is also more general than identifying the limits of implausibility for each individual (since there may be inter-individual differences), let alone each species in a large dataset.

```r
# get 90 and 95 percentile of speed and turning angle
sapply(data_copy[, c("in_speed", "angle")], function(z) {
  quantile(z, probs = c(0.9, 0.95), na.rm = TRUE)
})
```

Finally, we shall remove positions whose incoming and outgoing speeds are both greater than the 95th speed percentile using `atl_filter_covariates`.

```r
# filter the copy by the 95th percentile
data_filtered <- atl_filter_covariates(data_copy,
    filters = c("(in_speed < 0.024 & out_speed < 0.024) | angle < 40"))
```

We prepare the data for plotting.

### 3.4.2 Filtering Reflections or Track Subsets

When entire track subsets are affected by positioning error, they are more difficult to remove. This is the case when positions are reflected along a plane in the coordinate axis — the simplest way in which this can be explained is to examine the way in which we simulated the reflection earlier in this text, by simply adding an offset to the X and Y coordinates of each of 300 consecutive positions (indices 500 – 800).

An explanation of why these reflections occur is found in WATLAS PAPER: Bijleveld et al. in prep. In short, they are results of an error in the ATLAS localisation algorithm, and are usually removed in the localisation step's quality control procedures. However, some reflections may remain to confront users.

Reflections and other issues affecting track subsets cannot be resolved by targeting single positions, since there are rarely any position-specific covariates that can be used to identify a position as part of a larger subset that should be removed.

However, it is often possible to identify the bounds of problematic subsets, such as reflections, and remove positions between them. The way this is done is conceptually similar to the point outlier algorithm.

The `atl_remove_reflection` function implements one method remove reflected subsets of a track. The working is described:

1. Remove point outliers,
2. Re-calculate speed and turning angles
3. Identify the first unrealistic movement position (fast speed and high turning angle)
4. Setting this point as an anchor, identify the next position with unrealistic movement (as above)
5. Remove all positions between these two points,
6. Search for the next unrealistic movement, and repeat the process.

This is a minimal algorithm which can be developed further, and iterated multiple times to comprehensively remove reflections.

One important function argument is the estimated reflection length, i.e., how many positions are estimated to be reflected. This argument controls how many positions after the anchor are candidates for the reflection's end.

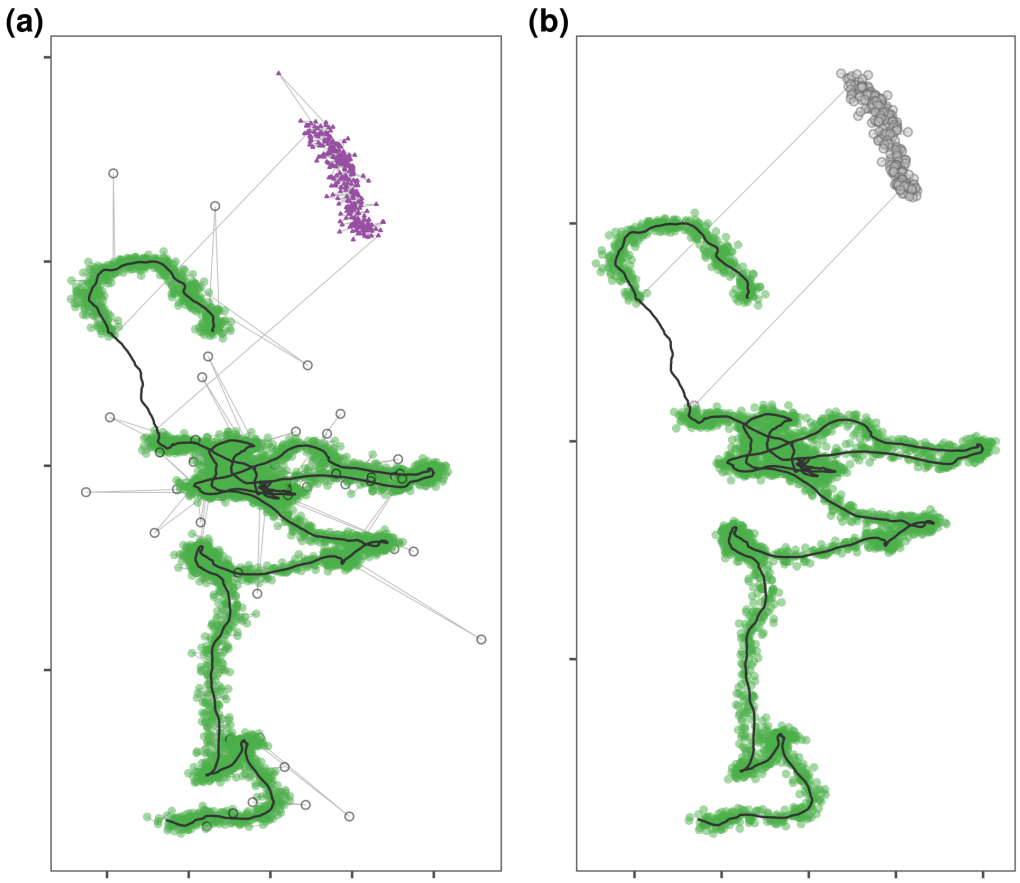
If the reflection does not end within this number of positions, the algorithm goes awry, and considers part of the reflection to be the valid data. Fortunately, this can be avoided by setting the length to the number of observations in the data.

However, if the reflection does not end at all, the algorithm will simply delete all positions from the anchor point onwards.

Thus, the algorithm is less of a double-edged sword and more a sword with no hilt — there are few safe ways to use it.

The difficulty in implementing a reasonable algorithm for dealing with this kind of track-subset wide positioning error reveals the general problem that algorithms are not easy to conceptualise or efficiently implement.

```
# attempt to remove reflections
data_no_reflection <- atl_remove_reflections(data_filtered,
                           point_angle_cutoff = 10,
                           reflection_speed_cutoff = 0.024)
# get reflections
reflection <- data_filtered[!data_no_reflection,
                                           on = c("x", "y")]
reflection <- na.omit(reflection)
```




Reducing large-scale positioning error in a movement track. (a) Positioning error may affect single observations resulting in point outliers or 'spikes' (red points), but it may also affect continuous susbets of a track, which we call a `prolonged spike' (purple triangles). While the former may be targeted by filtering on appropriate covariates such as speed and turning angle using theatlastoolsfunctionatl_filter_covariates, the latter cannot be effectively corrected by targeting each coordinate pair in isolation. **(b)** Theatlastoolsfunctionatl_remove_reflections` to identify prolonged spikes (red triangles) in tracking data is an illustrative example of targeting positioning errors that affect track subsets. While this method returns the canonical track without the prolonged spike (blue points) in this example, users are cautioned to frequently check this and similar semi-supervised algorithms' results.

Finally, we export the data without spikes and reflections to be used later.

```
fwrite(data_no_reflection, file = "data/data_no_reflection.csv")
```

# 4 Reducing Small-Scale Positioning Error and Aggregating Data

Once large-scale positioning errors have been removed using filters on bounds or on unrealistic movement, small-scale positioning errors may remain. Why is this the case?

The simple reason is that high-frequency tracking has a large component of error relative to the real movement of an individual, and the error / movement ratio increases with the frequency of tracking. (Noonan et al. 2019) has a better explanation, which users are strongly recommended to read.

The effect of these small-scale errors for high-frequency data is to cause overestimations of straight-line displacement (distance between two positions) and speed. If users were to apply a very strict speed filter, many of these positions would be lost. This loss of information would certainly reduce the sampling frequency, which would lead to better estimates of total distance and mean speed. However, while total distance and mean speed estimates would improve, the instantaneous distance and speed estimates would remain biased, which will most likely lead to erroneous inferences from the data.

However, users can turn the high-frequency of their data against itself by applying a simple smooth. The median smooth is recommended because it is robust to outliers, and is very well understood.

## 4.1 Preparing Libraries and Data

We first load some useful packages and define a colour palette.

```
# prep libs
library(data.table)
library(atlastools)
library(ggplot2)
library(patchwork)

# prepare a palette
pal <- RColorBrewer::brewer.pal(4, "Set1")
```

We load both the simulated, canonical data (which has no errors), as well as the data with errors which has been pre-processed to remove large-scale errors.

We also assign a variable, `window_size`, which will help identify the datasets when comparing median smooth moving window ($K$) sizes.

```
# read in the data and set the window size variable
data <- fread("data/data_sim.csv")[5000:10000, ]
data[, window_size := NA]

# data with small scale errors but no reflections or outliers
data_errors <- fread("data/data_no_reflection.csv")
data_errors[, window_size := 0]
```

## 4.2 Median Smoothing

We apply a median smooth with four different values of $K$, the moving window size — 3, 5, 11, and 21 positions. $K$ must be an odd number, but apart from that there is no correct magnitude of $K$. Very large $K$ will lead to unrealistic tracks, while small $K$ will not result in much reduction of error. Users are encouraged to plot their data before and after smoothing to examine the effect of different window sizes.

The `atlastools` function `atl_median_smooth` is quite fast and users can readily try multiple $K$ values as shown in the example below.

```
# smooth the data over four K values
list_of_smooths <- lapply(c(3, 5, 11, 21), function(K) {

  data_copy <- copy(data_errors)

  data_copy <- atl_median_smooth(data = data_copy,
                    x = "x",
                    y = "y",
                    time = "time",
                    moving_window = K)

  data_copy[, window_size := K]
})
```

We save the 11 point smoothed data.

```
fwrite(list_of_smooths[[3]], file = "data/data_smooth.csv")
```

We prepare the data for plotting.

```
# bind list after offset
data_plot <- mapply(function(df, offset) {
  df[, x := x + offset]
}, list_of_smooths, seq(0.4, 1.25, length.out = 4),
SIMPLIFY = F)

data_plot <- rbindlist(data_plot)
```

We prepare a plot of the smoothed data.

## 4.3 Thinning Data by Aggregation

Evenly thinning data is a good idea if statistical methods require even sampling, or if the volume of data is too large for statistical packages to efficiently handle it. In R, both may be true at once.

Here, we demonstrate thinning by aggregation on data that has been median smoothed using a $K$ of 11.

```
# choose the 11 point median smooth data
data_agg <- copy(list_of_smooths[[3]])

# get list of aggregated data
list_of_agg <- lapply(c(3, 10, 30, 120), function(z) {

  data_return <- atl_thin_data(data = data_agg,
                               interval = z,
                               method = "aggregate")

  data_return[, interval := z]

  return(data_return)
})

# get mean speed estimate and sd
speed_agg_smooth <-
  lapply(list_of_agg, function(df) {
    na.omit(df)
    df[, speed := atl_get_speed(df)]
    df[, list(median = median(speed, na.rm = T),
              sd = sd(speed, na.rm = T),
              interval = first(interval))]
  })

# bind
speed_agg_smooth <- rbindlist(speed_agg_smooth)
```

## 4.4 Aggregation Before Reducing Positioning Errors

Users may rightly wonder whether they can get away with aggregating their data, using a median aggregation function, and reduce data volumes, correct uneven sampling frequency, and reduce large-scale errors all in one go.

The answer to most questions in ecology is, "It depends". Median aggregation before correcting positioning errors can indeed be advantageous; for instance for faster visualisation while preserving the broad structure of a track.

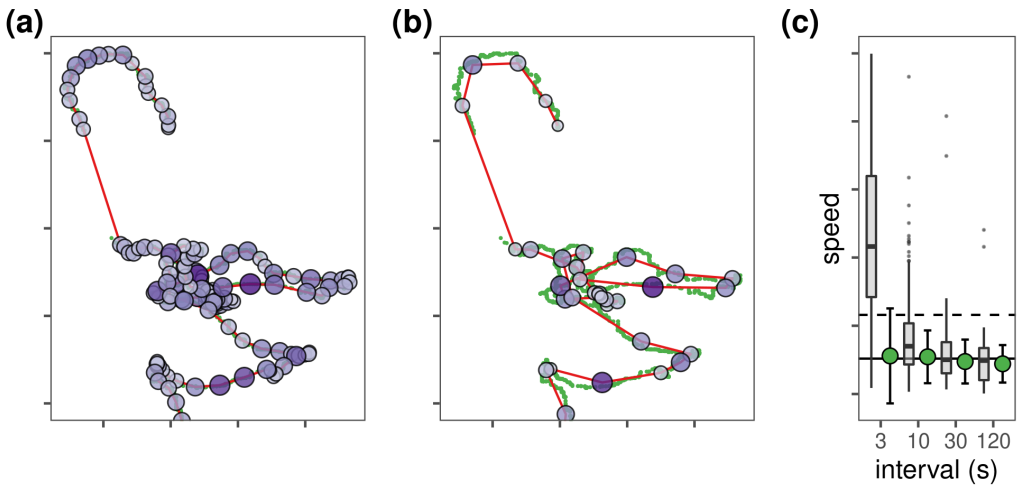
However, there are drawbacks. The main one is that information is lost in the aggregation process, lending less power to steps such as smoothing applied after aggregation. Further, aggregation (and indeed any kind of thinning) results in significantly different estimates of speed and distance from the real speed.

We show the effect of aggregating before any error correction here.

```
# read data with errors
data_errors <- fread("data/data_errors.csv")

# aggregate before correction
list_of_agg <- lapply(c(3, 10, 30, 120), function(z) {
  data_return <- atl_thin_data(data = data_errors,
                               interval = z,
                               method = "aggregate")
  data_return[, interval := z]
  data_return[, speed := atl_get_speed(data_return)]
  return(data_return)
})

# get real speed
data[, speed := atl_get_speed(data)]
```

### 4.4.1 Comparing median aggregation with and without smoothing

```
# now plot distribution of speed
data_agg <- rbindlist(list_of_agg)
```




Thinning a movement track using median aggregation preserves track structure, but affects essential track metrics such as speed. (a, b) Movement tracks with a canonical interval of 1s aggregated over intervals of (a) 10 and (b) 30 seconds, without removing large- or small-scale positioning errors. All symbols represent positions in the aggregated track, with the size of the symbol representing the standard deviation at each position. Blue crosses represent positions with speed $\leq$ the 95^th percentile of canonical speeds, while red triangles represent positions with speed $\geq$ 95^th percentile of canonical speeds. (c) Boxplot of instantaneous speeds after median aggregation of a 1s interval track over intervals of 3, 10, 30, and 120 seconds, but without the removal of positioning errors. The mean and 95^th percentile of speed in the canonical track are shown as solid and dashed lines, respectively. Aggregation without reducing positioning errors can result in speed estimates that are substantially different from the true speed.

# 5 Residence patches and their construction

## Prepare libraries

```
library(data.table)
library(atlastools)
library(ggplot2)
library(patchwork)

# for residence time
library(recurse)

# prepare a palette
pal <- RColorBrewer::brewer.pal(4, "Set1")
```

## 5.1 An example with simulated data

### 5.1.1 Read data and classify

```
# read patch data
data <- fread("data/data_for_res_patch.csv")

# filter
data <- data[y < 21, ]

# do recurse
data_recurse <- getRecursions(data[, list(x, y, time, id)],
                              radius = 1)

# assign residence time
data[, residence_time := data_recurse$residenceTime]
```

We first plot a figure of residence time per positions, and residence time per timestamp.

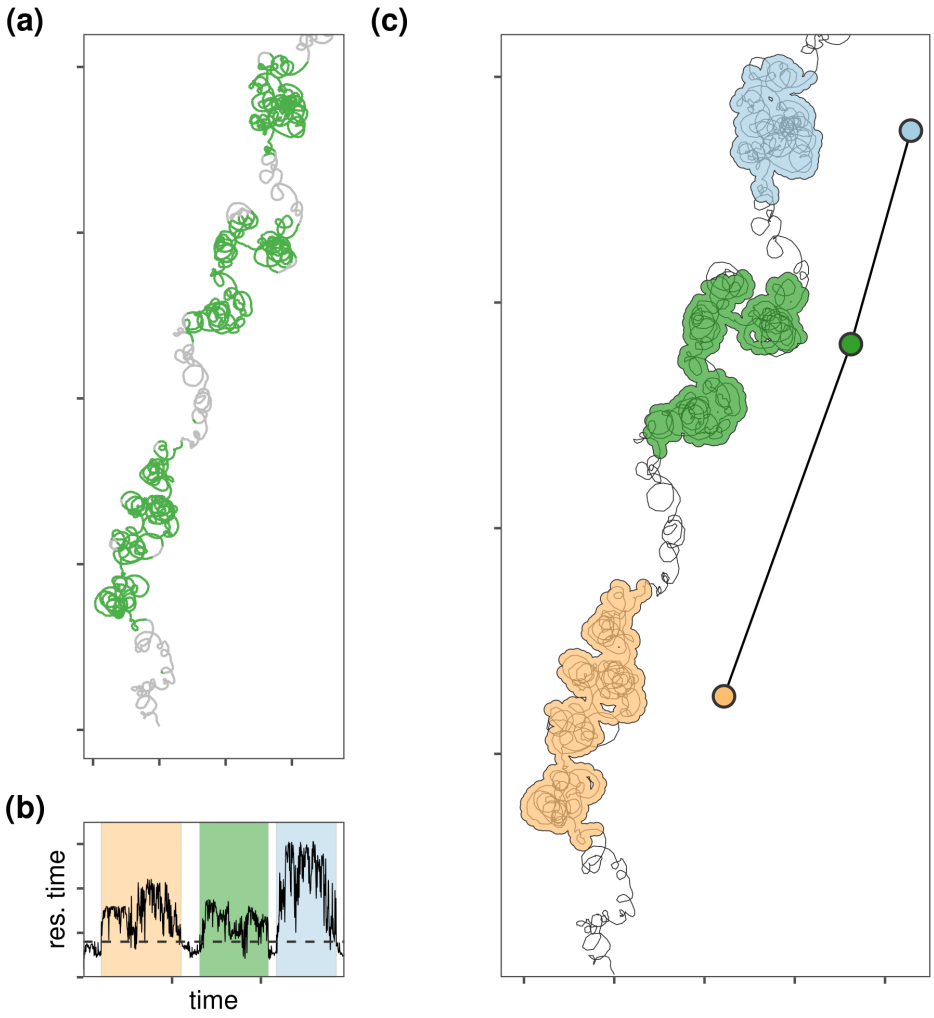
We construct residence patches from data where residence time is $> 0.15$.

```
# make residence patch
patch <- atl_res_patch(data[residence_time > 0.04, ],
                       buffer_radius = 0.1,
                       lim_spat_indep = 1,
                       lim_time_indep = 30)
```

```r
# get spatial representation
patch_sf <- atl_patch_summary(patch_data = patch,
                              which_data = "spatial",
                              buffer_radius = 0.15)

# get summary data
patch_summary <- atl_patch_summary(patch_data = patch,
                                   which_data = "summary")
```

### 5.1.2 Plot classified residence patches

We arrange the figures together.



# 6 Processing calibration data

## 6.1 Prepare libraries

```
# load libs
library(data.table)
library(atlastools)
library(ggplot2)
library(patchwork)

# prepare a palette
pal <- RColorBrewer::brewer.pal(4, "Set1")
```

## 6.2 Preliminary visualisation

```
# read and plot example data
data <- fread("data/atlas1060_allTrials_annotated.csv")
data_raw <- copy(data)
```

## 6.3 Filter by bounding box

Save an unprocessed copy.

```
data_unproc <- copy(data)
```

Filter by a bounding box.

```
# remove inside must be set to falses
data <- atl_filter_bounds(data = data,
                          x = "x", y = "y",
                          x_range = c(645000, max(data$x)),
                          remove_inside = FALSE)
```

## 6.4 Filter trajectories

### 6.4.1 Handle time

Time in ATLAS tracking is counted in milliseconds and is represented by a 64-bit integer (type `long`), which is not natively supported in R; it will instead be converted to a `numeric`, or `double`.

This is not what is intended, but it works. The `bit64` package can help handle 64-bit integers if you want to keep to intended type.

A further issue is that 64-bit integers (whether represented as `bit64` or `double`) do not yield meaninful results when you try to convert them to a date-time object, such as of the class `POSIXct`.

This is because `as.POSIXct` fails when trying to work with 64-bit integers (it cannot interpret this type), and returns a date many thousands of years in the future (approx. 52,000 CE) if the time column is converted to `numeric`.

There are two possible solutions. The parsimonious one is to convert the 64-bit number to a 32-bit short integer (dividing by 1000), or to use the `nanotime` package.

The conversion method loses an imperceptible amount of precision. The `nanotime` requires installing another package. The first method is shown here.

In the spirit of not destroying data, we create a second lower-case column called `time`.

```
# divide by 1000, convert to integer, then convert to POSIXct
data[, time := as.integer(TIME / 1000)]
```

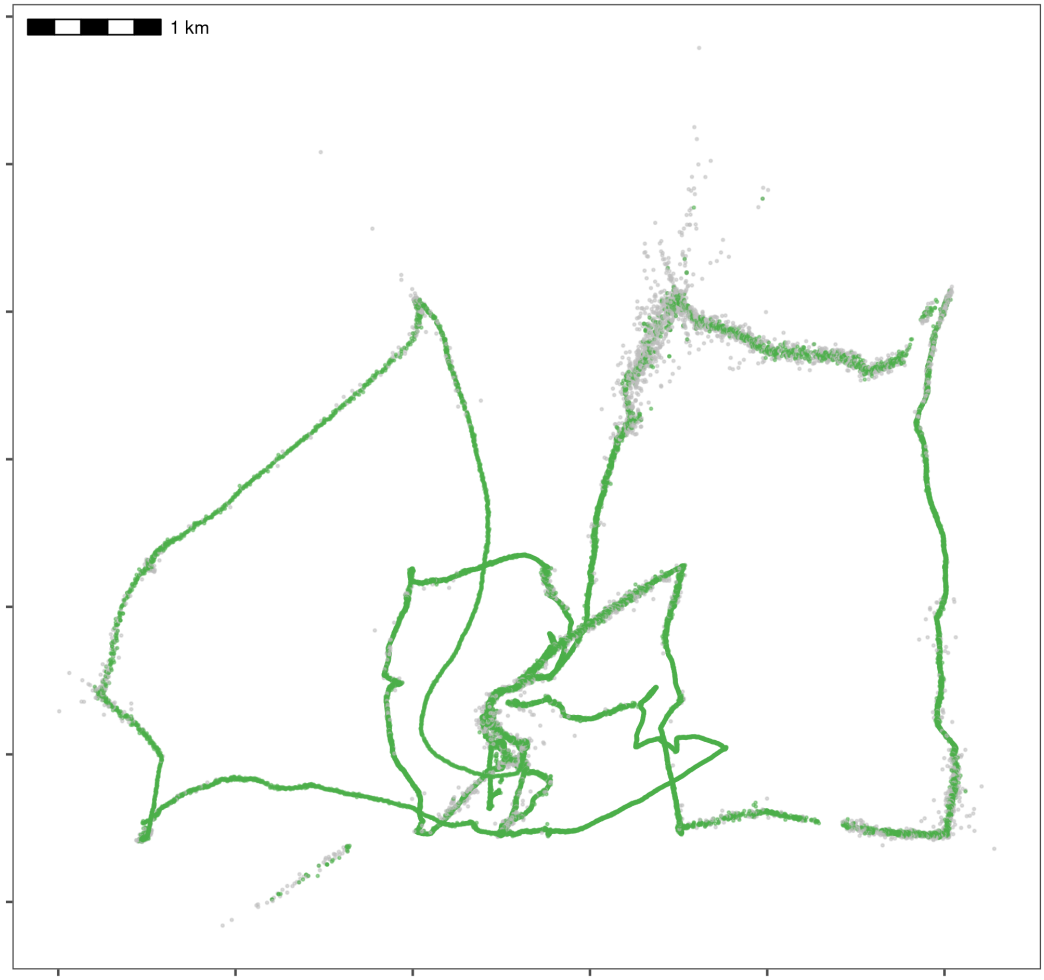
### 6.4.2 Add speed and turning angle

```
# add incoming and outgoing speed
data[, `:=` (speed_in = atl_get_speed(data,
                                      x = "x",
                                      y = "y",
                                      time = "time"),
             speed_out = atl_get_speed(data, type = "out"))]

# add turning angle
data[, angle := atl_turning_angle(data = data)]
```

### 6.4.3 Get 95th percentile of speed and angle

```
# use sapply
speed_angle_thresholds <-
  sapply(data[, list(speed_in, speed_out, angle)],
       quantile, probs = 0.9, na.rm = T)
```

### 6.4.4 Plot to see speeds




### 6.4.5 Filter on speed

Here we use a speed threshold of 15 m/s, the fastest known boat speed.

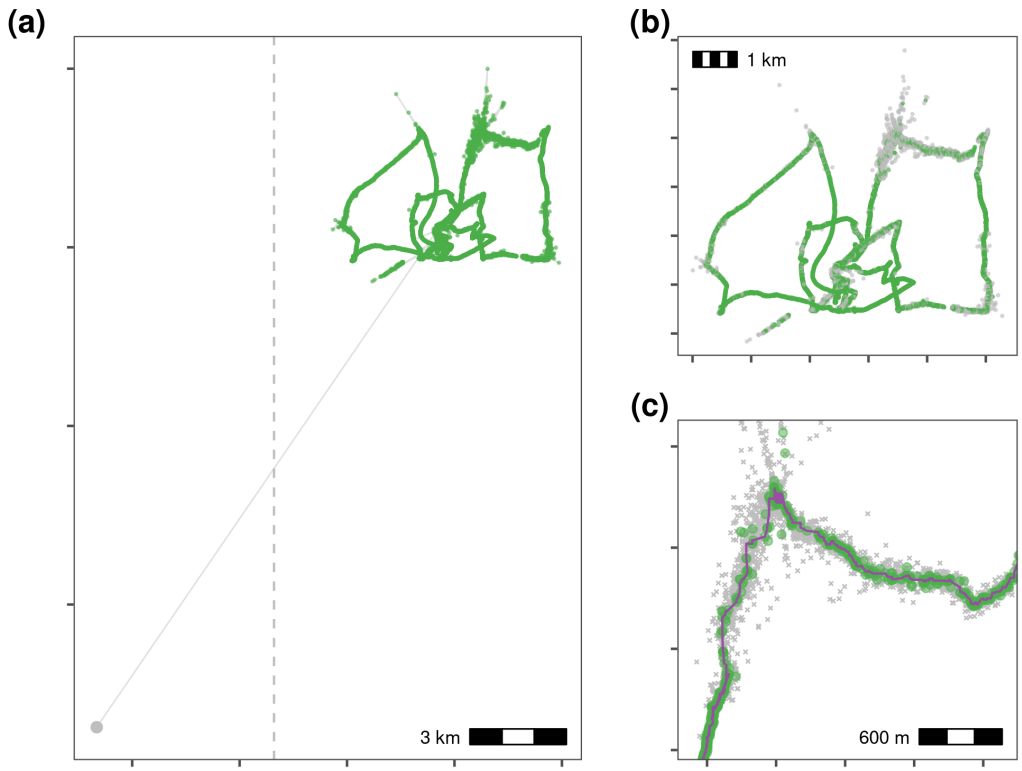
```
# make a copy
data_unproc <- copy(data)

# remove speed outliers
data <- atl_filter_covariates(data = data,
            filters = c("(speed_in < 15 & speed_out < 15)"))

# recalculate speed and angle
data[, `:=` (speed_in = atl_get_speed(data,
                                      x = "x",
                                      y = "y",
                                      time = "time"),
             speed_out = atl_get_speed(data, type = "out"))]
```

```r
# add turning angle
data[, angle := atl_turning_angle(data = data)]
```

## 6.5 Smoothing the trajectory

```r
# apply a 5 point median smooth, first make a copy
data_unproc <- copy(data)

# now apply the smooth
data <- atl_median_smooth(data = data,
                          x = "x", y = "y", time = "time",
                          moving_window = 5)
```

### 6.5.1 Plot pre-processing steps




## 6.6 Making residence patches

### 6.6.1 Prepare data

An indicator of individual residence at or near a position can be useful when attempting to identify residence patches. Positions can be filtered on a metric such as residence time (see Bracis et al. 2018).

In this dataset, residence positions are marked in the `tID` column as beginning with `WP`. These can be extracted and converted into residence patches.

```
# make a data copy
data_unproc <- copy(data)
```

### 6.6.2 Calculate residence time

Subset for boating segments: 2020-08-24 11:13:00 – 2020-08-24 12:36:11.

```
# load recurse
library(recurse)
data_recurse <- data_unproc[, list(x, y, time, TAG, UTCtime)]

# thin the data
data_recurse <- atl_thin_data(data = data_recurse,
                              interval = 30, id_columns = "TAG",
                              method = "resample")

# get 4 column data
data_recurse <- data_recurse[, list(x, y, time, TAG)]

# get recurse data for a 10m radius
recurse_stats <- getRecursions(data_recurse,
                               radius = 50, timeunits = "mins")

# assign to recurse data
data_recurse[, res_time := recurse_stats$residenceTime]

# save recurse data
fwrite(data_recurse, file = "data/data_calib_recurse.csv")
```

Read in the data.

```
data_recurse <- fread("data/data_calib_recurse.csv")
```

Subset waypoint data.

```
library(stringi)
data_res <- data_unproc[stri_detect(tID, regex = "(WP)")]
```

### 6.6.3 Plot residence time

### 6.6.4 Run residence patch method

```
# assign id as tag
data_recurse[, id := as.character(TAG)]

# subset on 5 minute residence time
data_recurse_subset <- data_recurse[res_time >= 5, ]

# on known residence points
patch_res_known <- atl_res_patch(data_recurse_subset,
                                 buffer_radius = 10,
```

```
                                lim_spat_indep = 100,
                                lim_time_indep = 5,
                                min_fixes = 3)
```

### 6.6.5 Get spatial and summary objects

```
# for the known and unkniwn patches
patch_sf_data <- atl_patch_summary(patch_res_known, which_data = "spatial",
                                   buffer_radius = 20)

# assign crs
sf::st_crs(patch_sf_data) <- 32631

# get summary data
patch_summary_data <- atl_patch_summary(patch_res_known, which_data = "summary")
```

### 6.6.6 Get waypoint centroids

```
# get centroid
data_res_summary <- data_res[, list(x_median = median(x),
                                    y_median = median(y),
                                    t_median = median(time)),
                             by = "tID"]

# now get times 10 mins before and after
data_res_summary[, `:=`(t_min = t_median - (10 * 60),
                        t_max = t_median + (10 * 60))]

# make a list of positions 10min before and after
wp_data <- mapply(function(l, u, mx, my) {
  tmp_data <- data_unproc[inrange(time, l, u)]
  tmp_data[, distance := sqrt((mx - x)^2 + (my - y)^2)]
  # keep within 50
  tmp_data <- tmp_data[distance <= 50, ]
  # get duration
  return(diff(range(tmp_data$time)))
}, data_res_summary$t_min, data_res_summary$t_max,
   data_res_summary$x_median, data_res_summary$y_median,
SIMPLIFY = FALSE)
```

### 6.6.7 Read Griend

```
# read griend
griend <- sf::st_read("data/griend_polygon/griend_polygon.shp")
```

We prepare a plot of the residence patches.

## 6.7 Compare patch metrics

### 6.7.1 Compare known patches

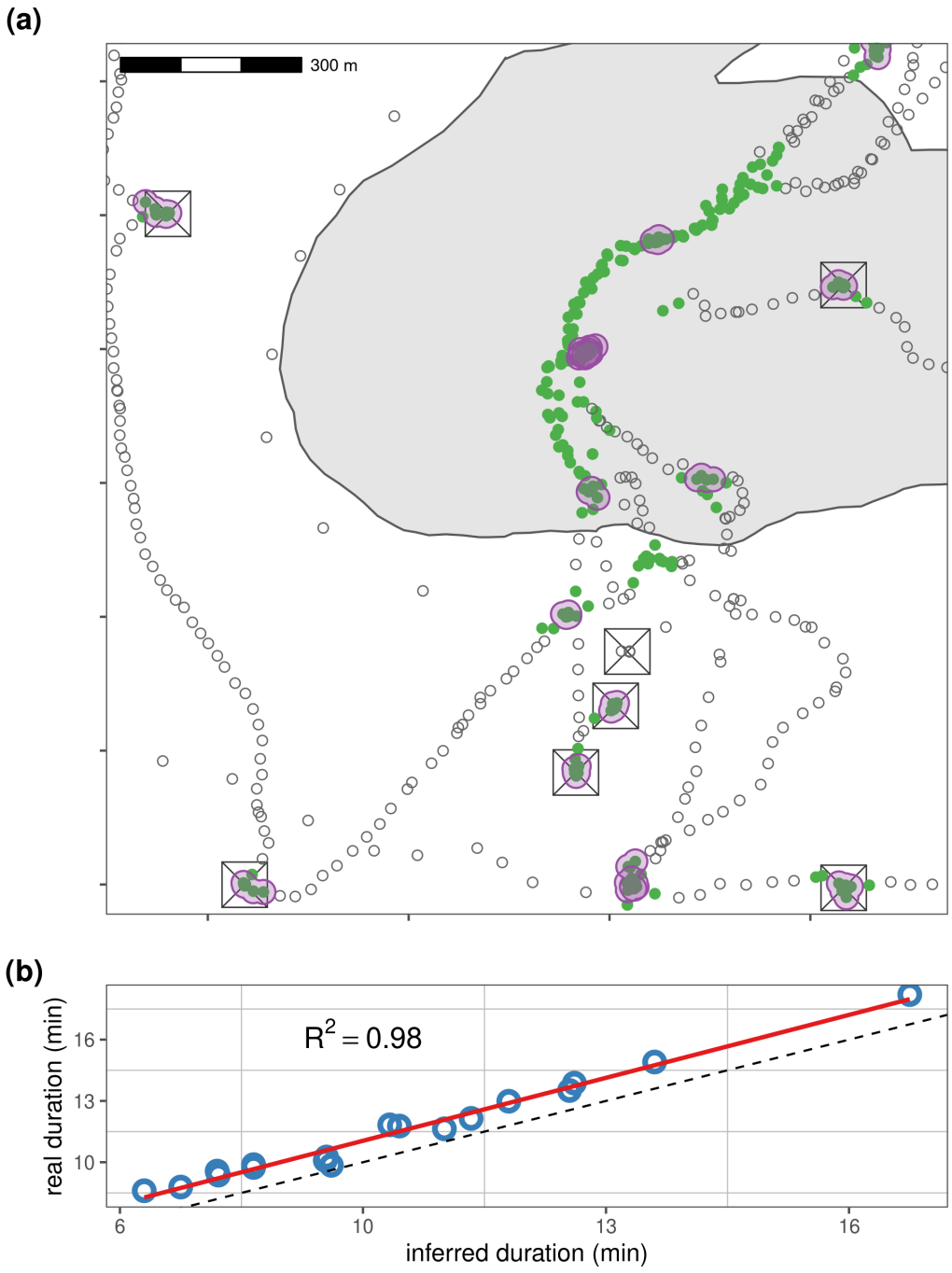
```
# get known patch summary
data_res <- data_unproc[stringi::stri_detect(tID, regex = "(WP)"), ]

# get waypoint summary
patch_summary_real <- data_res[, list(nfixes_real = .N,
                                      x_median = round(median(x), digits = -2),
                                      y_median = round(median(y), digits = -2)),
                               by = "tID"]

# add real duration
patch_summary_real[, duration_real := unlist(wp_data)]
# round median coordinate for inferred patches
patch_summary_inferred <-
  patch_summary_data[,
                     c("x_median", "y_median",
                       "nfixes", "duration", "patch")
                     ][, `:=`(x_median = round(x_median, digits = -2),
                              y_median = round(y_median, digits = -2))]

# join with respatch summary
patch_summary_compare <-
  merge(patch_summary_real,
        patch_summary_inferred,
        on = c("x_median", "y_median"),
        all.x = TRUE, all.y = TRUE)

patch_summary_compare[!is.na(tID)]

# drop nas
patch_summary_compare <- na.omit(patch_summary_compare)

# drop patch around WP080
patch_summary_compare <- patch_summary_compare[tID != "WP080", ]
```

12 patches are identified where there are no waypoints, while 2 waypoints are not identified as patches. These waypoints consisted of 6 and 15 (WP098 and WP092) positions respectively, and were lost when the data were aggregated to 30 second intervals.

### 6.7.2 Plot durations comparisons

```
# get linear model
model_duration <- lm(duration_real ~ duration,
                     data = patch_summary_compare)

# get R2
summary(model_duration)
```

### 6.7.3 Join patch map and duration comparison figure



# 7 Processing Egyptian fruit bat tracks

We show the pre-processing pipeline at work on the tracks of three Egyptian fruit bats (Rousettus aegyptiacus), and construct residence patches.

## 7.1 Prepare libraries

```
# load libs
library(data.table)
library(RSQLite)
library(atlastools)
library(ggplot2)
library(patchwork)

# prepare a palette
pal <- RColorBrewer::brewer.pal(4, "Set1")
```

## 7.2 Read bat data

Read the bat data and convert to a csv file.

```
# prepare the connection
con <- dbConnect(drv = SQLite(), dbname = "data/Three_example_bats.sql")

# list the tables
table_name <- dbListTables(con)

# prepare to query all tables
query <- sprintf('select * from \"%s\"', table_name)

# query the database
data <- dbGetQuery(conn = con, statement = query)

# disconnect from database
dbDisconnect(con)
```

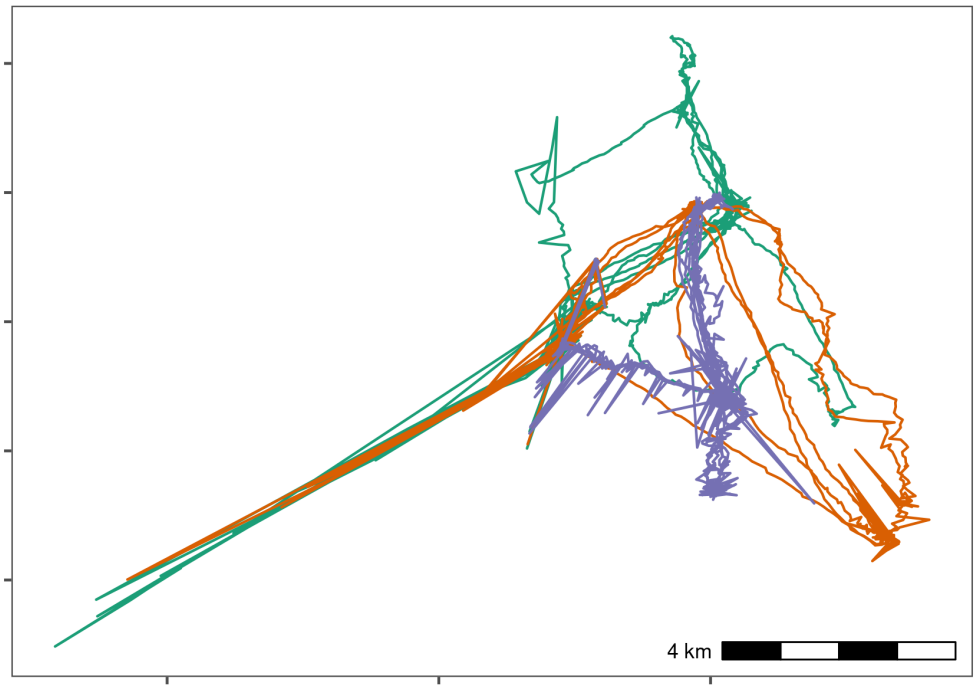
Convert data to csv.

```
# convert data to datatable
setDT(data)

# write data for QGIS
fwrite(data, file = "data/bat_data.csv")
```

## 7.3 Sanity check: Plot bat data

Save the raw data plot.

Show the plot.




## 7.4 Prepare data for filtering

### 7.4.1 Prepare data per individual

```
# split bat data by tag
# first make a copy using the data.table function copy
# this prevents the orignal data from being modified by atlastools
# functions which DO MODIFY BY REFERENCE!
data_split <- copy(data)

# now split
data_split <- split(data_split, data$TAG)
```

## 7.5 Filter by covariates

No natural bounds suggest themselves, and we proceed to filter by covariates, since point outliers are obviously visible.

We use filter out positions with `SD > 20` and positions calculated using only 3 base stations.

We use the function `atl_filter_covariates`.

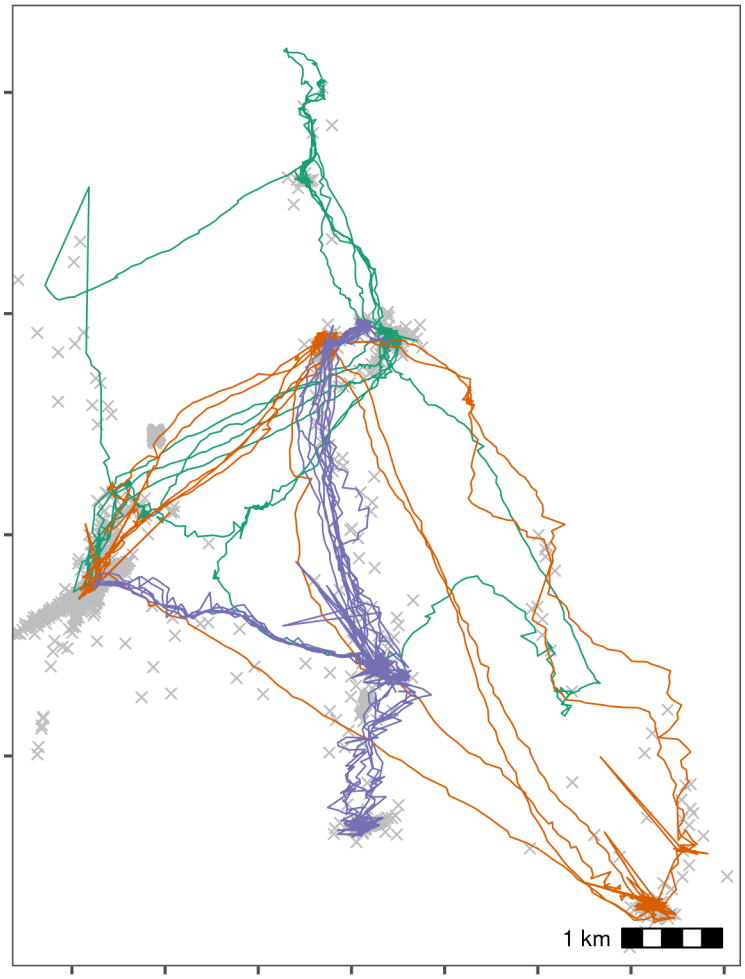
```r
# get SD.
# since the data are data.tables, no assignment is necessary
invisible(
  lapply(data_split, function(dt) {
    dt[, SD := sqrt(VARX + VARY + (2 * COVXY))]
  })
)

# filter for SD <= 20
# here, reassignment is necessary as rows are being removed
# the atl_filter_covariates function could have been used here
data_split <- lapply(data_split, function(dt) {
  dt <- atl_filter_covariates(
    data = dt,
    filters = c("SD <= 20",
                "NBS > 3")
  )
})

# check whether the filter has worked
invisible(
  lapply(data_split, function(dt) {
    assertthat::assert_that(min(dt$SD) <= 20,
                            msg = "some SDs above 20 remain")
    assertthat::assert_that(min(dt$NBS) > 3,
                            msg = "some NBS below 3 remain")
  })
)
```

### 7.5.1 Sanity check: Plot filtered data

```r
# bind all individuals together
data_split <- rbindlist(data_split)
```

Save the filtered data plot.

Show the plot.




## 7.6 Filter by speed

Some point outliers remain, and should be removed using a speed filter.

First we calculate speeds.

```
# calculate speed on split data once more
data_split <- split(data_split, data_split$TAG)

# get speeds as with SD, no reassignment required for columns
invisible(
  lapply(data_split, function(dt) {

    # first process time to seconds
    # assign to a new column
    dt[, time := floor(TIME / 1000)]

    dt[, `:=`(speed_in = atl_get_speed(dt,
```

```
                                        x = "X", y = "Y",
                                        time = "time",
                                        type = "in"),
                speed_out = atl_get_speed(dt,
                                        x = "X", y = "Y",
                                        time = "time",
                                        type = "out"))]
  })
)
```

Now filter for speeds $> 20$ m/s (around 70 km/h).

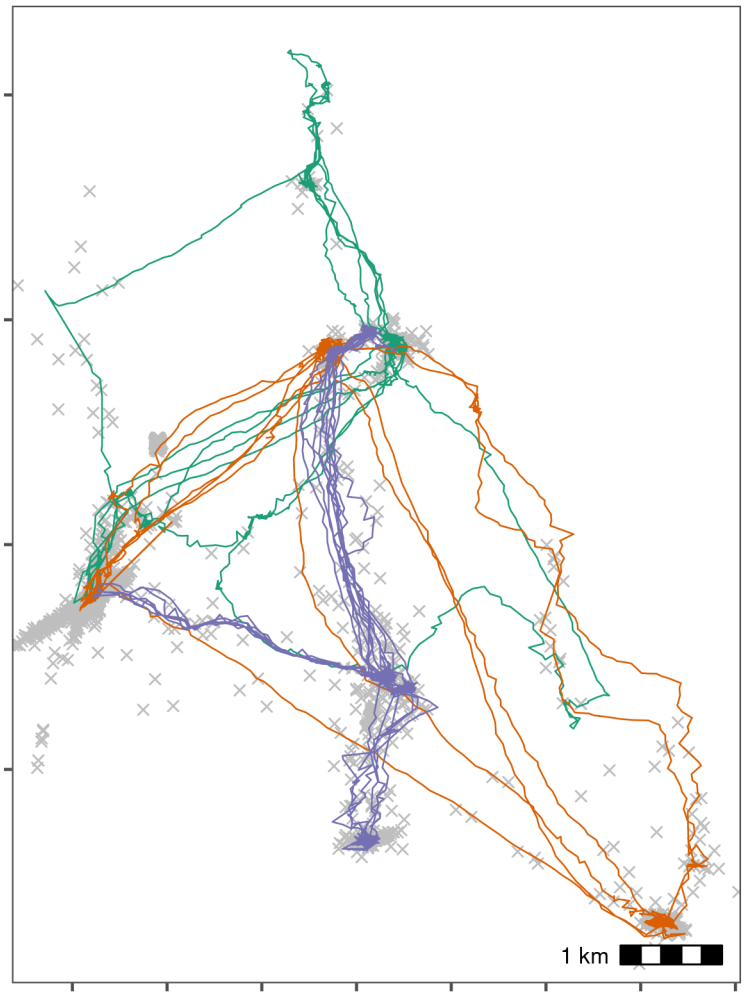
```
# filter speeds
# reassignment is required here
data_split <- lapply(data_split, function(dt) {
  dt <- na.omit(dt, cols = c("speed_in", "speed_out"))

  dt <- atl_filter_covariates(data = dt,
                              filters = c("speed_in <= 20",
                                          "speed_out <= 20"))
})
```

### 7.6.1 Sanity check: Plot speed filtered data

```
# bind all individuals together
data_split <- rbindlist(data_split)
```

Save the speed filtered data plot.

Show the plot.




## 7.7 Median smoothing

Apply a 5 point median smooth to the data.

```
# since the function modifies in place, we shall make a copy
data_smooth <- copy(data_split)

# split the data again
data_smooth <- split(data_smooth, data_smooth$TAG)
```

Remember, `atl_median_smooth` MODIFIES IN PLACE.

```
# apply the median smooth to each list element
# no reassignment is required as THE FUNCTION MODIFIES IN PLACE!
invisible(
  # the function arguments to atl_median_smooth
  # can be passed directly in lapply
  lapply(data_smooth, atl_median_smooth,
```

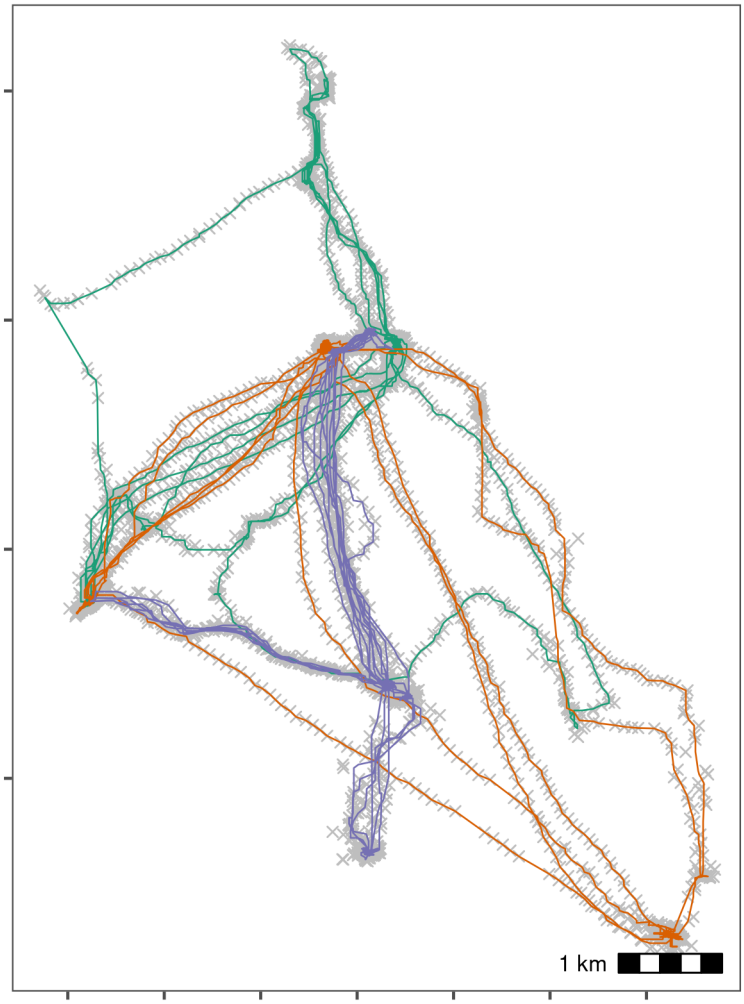
```
        time = "time", moving_window = 5)
)
```

### 7.7.1 Sanity check: Plot smoothed data

```
# recombine split up data that has been smoothed
data_smooth <- rbindlist(data_smooth)
```

Save the smoothed data plot.

Show the plot.




## 7.8 Making residence patches

### 7.8.1 Calculating residence time

First, the data is put through the `recurse` package to get residence time.

```r
# load recurse
library(recurse)

# split the data
data_smooth <- split(data_smooth, data_smooth$TAG)
```

Get residence time. Since bats may revisit the same features, we want to prevent confusion between frequent revisits and prolonged residence.

For this, we stop summing residence times within Z metres of a location if the animal exited the area for one hour or more.

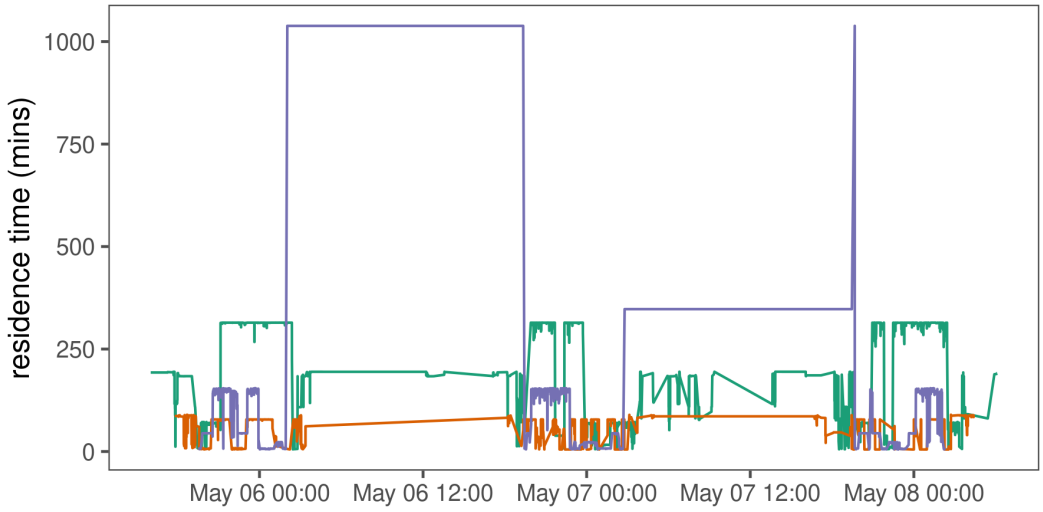
```r
# get residence times

data_residence <- lapply(data_smooth, function(dt) {
  # do basic recurse
  dt_recurse <- getRecursions(
    x = dt[, c("X", "Y", "time", "TAG")],
    radius = 50,
    timeunits = "mins"
  )

  # get revisit stats
  dt_recurse <- setDT(
    dt_recurse[["revisitStats"]]
  )

  # count long absences from the area
  dt_recurse[, timeSinceLastVisit :=
          ifelse(is.na(timeSinceLastVisit), -Inf, timeSinceLastVisit)]
  dt_recurse[, longAbsenceCounter := cumsum(timeSinceLastVisit > 60),
             by = .(coordIdx)
             ]
  # get data before the first long absence of 60 mins
  dt_recurse <- dt_recurse[longAbsenceCounter < 1, ]

  dt_recurse <- dt_recurse[, list(
    resTime = sum(timeInside),
    fpt = first(timeInside),
    revisits = max(visitIdx)
  ),
  by = .(coordIdx, x, y)
  ]

  # prepare and merge existing data with recursion data
  dt[, coordIdx := seq(nrow(dt))]

  dt <- merge(dt,
              dt_recurse[, c("coordIdx", "resTime")],
              by = c("coordIdx"))

  setorderv(dt, "time")
})
```

### 7.8.2 Sanity check: Residence-time time-series

```
# bind the list
data_residence <- rbindlist(data_residence)

# get time as human readable
data_residence[, ts := as.POSIXct(time, origin = "1970-01-01")]
```

Save the speed filtered data plot.

Show the plot.




### 7.8.3 Constructing residence patches

Split the data and construct residence patches.

Some preparation is required. First, the function requires columns `x`, `y`, `time`, and `id`.

```
# add an id column
data_residence[, `:=`(id = TAG,
                      x = X, y = Y)]

# filter for residence time > 5 minutes
data_residence <- data_residence[resTime > 5, ]

# split the data
data_residence <- split(data_residence, data_residence$TAG)
```

Now segment-cluster into residence patches.

```
# segment into residence patches
data_patches <- lapply(data_residence, atl_res_patch,
                       buffer_radius = 25)
```

### 7.8.4 Getting residence patch data

We get the residence patch data as spatial `sf-MULTIPOLYGON` objects.

```
# get data spatials
data_spatials <- lapply(data_patches, atl_patch_summary,
                        which_data = "spatial",
                        buffer_radius = 25)

# bind list
data_spatials <- rbindlist(data_spatials)

# convert to sf
library(sf)
data_spatials <- st_sf(data_spatials, sf_column_name = "polygons")

# assign a crs
st_crs(data_spatials) <- st_crs(2039)
```

### 7.8.5 Write patch spatial representations

```
st_write(data_spatials,
         dsn = "data/data_bat_residence_patches.gpkg")
```

Write cleaned bat data.

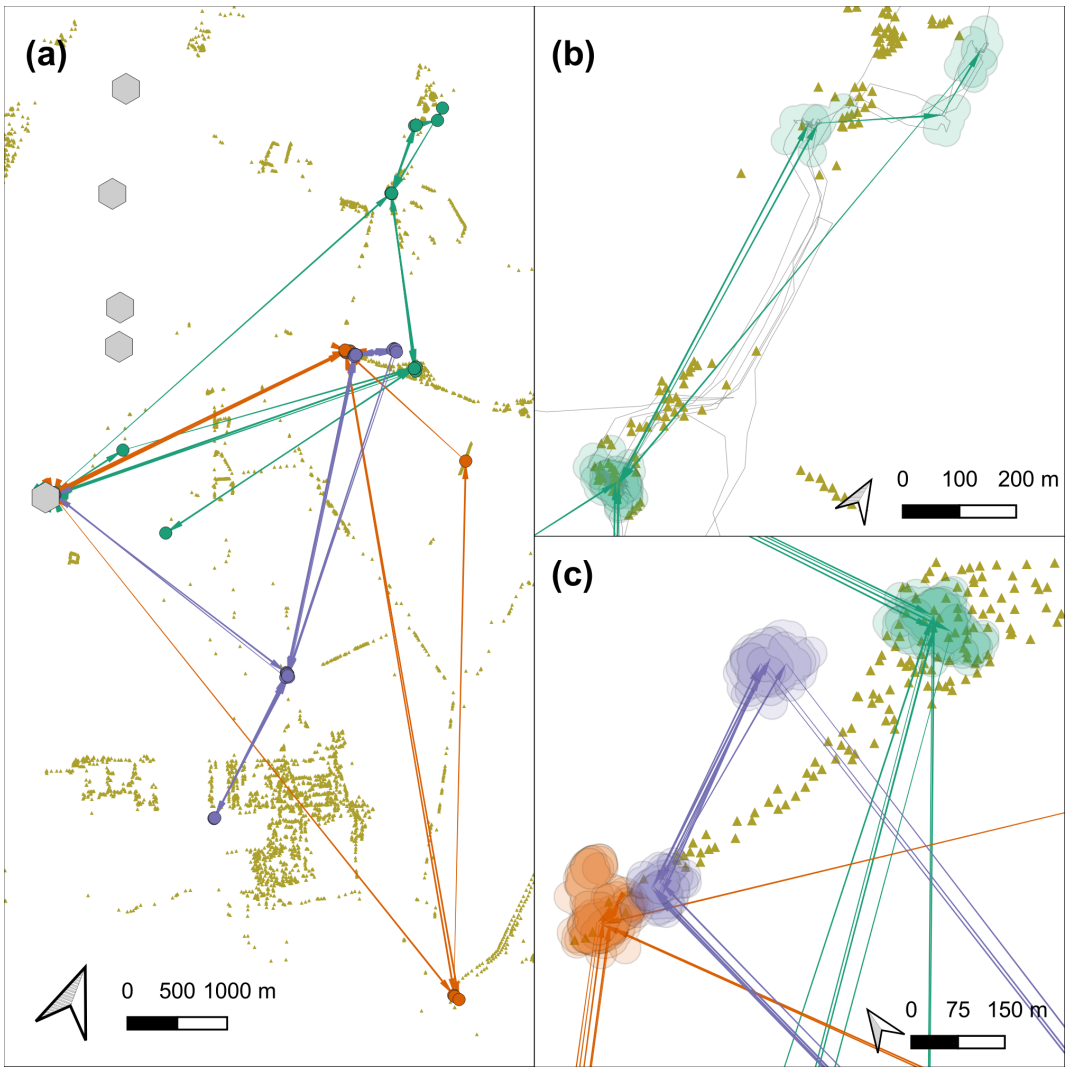
```
data_clean <- fwrite(rbindlist(data_smooth),
                     file = "data/data_bat_smooth.csv")
```

Write patch summary.

```
# get summary
patch_summary <- lapply(data_patches, atl_patch_summary)

# bind summary
patch_summary <- rbindlist(patch_summary)

# write
fwrite(patch_summary,
       "data/data_bat_patch_summary.csv")
```

## 7.9 Processed bat patches

This figure made in QGIS.


(a)
0 500 1000 m
(b)
0 100 200 m
(c)
0 75 150 m

# 8 References

Bjørneraas, Kari, Bram Van Moorter, Christer Moe Rolandsen, and Ivar Herfindal. 2010. "Screening Global Positioning System Location Data for Errors Using Animal Movement Characteristics." The Journal of Wildlife Management 74 (6): 1361–6. https://doi.org/10.1111/j.1937-2817.2010.tb01258.x.

Gurarie, Eliezer, Christen H. Fleming, William F. Fagan, Kristin L. Laidre, Jesús Hernández-Pliego, and Otso Ovaskainen. 2017. "Correlated Velocity Models as a Fundamental Unit of Animal Movement: Synthesis and Applications." Movement Ecology 5 (1): 13. https://doi.org/10.1186/s40462-017-0103-3.

Noonan, Michael J., Christen H. Fleming, Thomas S. Akre, Jonathan Drescher-Lehman, Eliezer Gurarie, Autumn-Lynn Harrison, Roland Kays, and Justin M. Calabrese. 2019. "Scale-Insensitive Estimation of Speed and Distance Traveled from Animal Tracking Data." Movement Ecology 7 (1): 35. https://doi.org/10.1186/s40462-019-0177-1.