

1 **Supplementary Material for *A Guide to***
2 ***Pre-processing High-Frequency Animal***
3 ***Tracking Data***

4 Pratik R. Gupte Christine E. Beardsworth Orr Spiegel
5 Emmanuel Lourie Sivan Toledo Ran Nathan
6 Allert I. Bijleveld

7 2021-02-16

8 Contents

9	1 Validating the Residence Patch Method with Calibration Data	3
10	1.1 Outline of Cleaning Steps	3
11	1.2 Install atlastools from Github	3
12	1.3 Prepare libraries	4
13	1.4 Access data and preliminary visualisation	4
14	1.5 Filter by bounding box	5
15	1.6 Filter trajectories	8
16	1.7 Smoothing the trajectory	9
17	1.8 Thinning the data	11
18	1.9 Residence patches	13
19	1.10 Compare patch metrics	15
20	1.11 Main text Figure 6	19
21	2 Processing Egyptian Fruit Bat Tracks	20
22	2.1 Prepare libraries	20
23	2.2 Read bat data	20
24	2.3 A First Visual Inspection	21
25	2.4 Prepare data for filtering	22
26	2.5 Filter by covariates	22
27	2.6 Filter by speed	23
28	2.7 Median smoothing	26
29	2.8 Making residence patches	26
30	3 References	32

31 **1 Validating the Residence Patch
32 Method with Calibration Data**

33 Here we show how the residence patch method (Barraquand and Benhamou 2008;
34 Bijleveld et al. 2016; Oudman et al. 2018) accurately estimates the duration of known
35 stops in a track collected as part of a calibration exercise in the Wadden Sea. These
36 data can be accessed from the data folder at this link: <https://doi.org/10.5281/zenodo.4287462>. These data are more fully reported in (Beardsworth et al. 2021).

38 **1.1 Outline of Cleaning Steps**

39 We begin by preparing the libraries we need, and installing atlastools from Github.
40 After installing atlastools, we visualise the data to check for location errors, and
41 find a single outlier position approx. 15km away from the study area (Fig. 1.1,
42 1.2). This outlier is removed by filtering data by the X coordinate bounds using
43 the function atl_filter_bounds; X coordinate bounds $\leq 645,000$ in the UTM 31N
44 coordinate reference system were removed ($n = 1$; remaining positions = 50,815; Fig.
45 1.2). We then calculate the incoming and outgoing speed, as well as the turning
46 angle at each position using the functions atl_get_speed and atl_turning_angle
47 respectively, as a precursor to targeting large-scale location errors in the form of
48 point outliers. We use the function atl_filter_covariates to remove positions
49 with incoming and outgoing speeds \geq the speed threshold of 15 m/s ($n = 13,491$,
50 26.5%; remaining positions = 37,324, 73.5%; Fig. 1.3; main text Fig. 7.b). This speed
51 threshold is chosen as the fastest boat speed during the experiment, 15 m/s. Finally,
52 we target small-scale location errors by applying a median smoother with a moving
53 window size $K = 5$ using the function atl_median_smooth (Fig. 1.4; main text Fig.
54 7.c). Smoothing does not reduce the number of positions. We thin the data to a 30
55 second interval leaving 1,803 positions (4.8% positions of the smoothed track)

56 **1.2 Install atlastools from Github**

57 atlastools is available from Github and is archived on Zenodo (Gupte 2020). It
58 can be installed using remotes or devtools. Here we use the remotes function
59 install_github.

```
install.packages("remotes")

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

60 **A Note on :=**

61 The `atlastools` package is based on `data.table`, to be fast and efficient (Dowle and
62 Srinivasan 2020). A key feature is modification in place, where data is changed
63 without making a copy. This is already implemented in R and will be familiar to
64 many users as `data_frame$column_name <- values`.
65 The `data.table` way of writing this assignment would be `data_frame[, column_name := values]`. We use this syntax throughout, as it provides
66 many useful shortcuts, such as multiple assignment:
67
68 `data_frame[, c("col_a", "col_b") := list(values_a, values_b)]`
69 Users can use this special syntax, and will find it convenient with practice, but
70 there are *no* cases where users *must* use the `data.table` syntax, and can simply
71 treat the data as a regular `data.frame`. However, users are advised to convert their
72 `data.frame` to a `data.table` using the function `data.table::setDT()`.

73 **1.3 Prepare libraries**

74 First we prepare the libraries we need. Libraries can be installed from CRAN if
75 necessary.

```
# for data handling
library(data.table)
library(atlastools)

# for recursion analysis
library(recurse)

# for plotting
library(ggplot2)
library(patchwork)

# making a colour palette
pal <- RColorBrewer::brewer.pal(5, "Set1")
pal[3] <- "seagreen"
```

76 **1.4 Access data and preliminary visualisation**

77 First we access the data from a local file using the `data.table` package (Dowle and
78 Srinivasan 2020). We look at the first few rows, using `head()`. We then visualise the
79 raw data.

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

# see raw data
head(data_raw)
```

80 Here we show how data can be easily visualised using the popular plotting package
81 `ggplot2`. Note that we plot both the points (`geom_point`) and the inferred path

82 between them (geom_path), and specify a geospatial coordinate system in metres,
83 suitable for the Dutch Wadden Sea (UTM 31N; ESPG code:32631; coord_sf). We
84 save the output to file for future reference.

85 Since plot code can become very lengthy and complicated, we omit showing further
86 plot code in versions of this document rendered as PDF or HTML; it can however
87 be seen in the online .Rmd version.

```
# plot data
fig_data_raw <-
  ggplot(data) +
  geom_path(aes(x, y),
            col = "grey", alpha = 1, size = 0.2
  ) +
  geom_point(aes(x, y),
            col = "grey", alpha = 0.2, size = 0.2
  ) +
  ggthemes::theme_few() +
  theme(
    axis.title = element_blank(),
    axis.text = element_blank()
  ) +
  coord_sf(crs = 32631)

# save figure
ggsave(fig_data_raw,
       filename = "figures/fig_calibration_raw.png",
       width = 185 / 25
)
```

88 1.5 Filter by bounding box

89 We first save a copy of the data, so that we can plot the raw data with the cleaned
90 data plotted over it for comparison.

```
# make a copy using the data.table copy function
data_unproc <- copy(data)

91 We then filter by a bounding box in order to remove the point outlier to the far south
92 east of the main track. We use the atl_filter_bounds functions using the x_range
93 argument, to which we pass the limit in the UTM 31N coordinate reference system.
94 This limit is used to exclude all points with an X coordinate < 645,000.
```

95 We then plot the result of filtering, with the excluded point in black, and the points
96 that are retained in green.

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

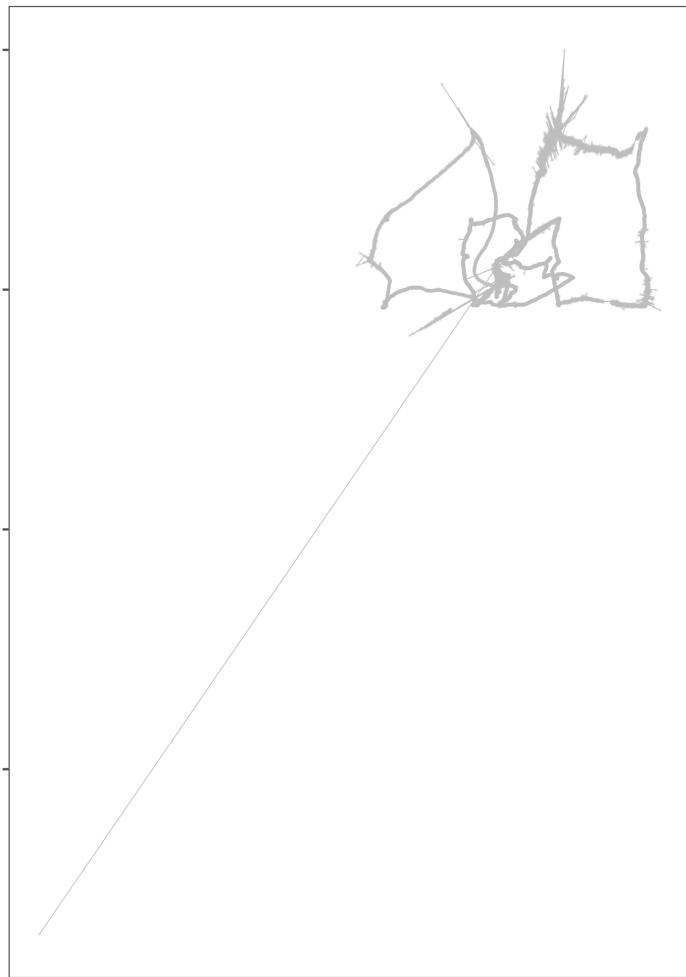


Figure 1.1: The raw data from a calibration exercise conducted around the island of Griend in the Dutch Wadden Sea. A handheld WATLAS tag was used to examine how ATLAS data compared to GPS tracks, and we use the WATLAS data here to demonstrate the basics of the pre-processing pipeline, as well as validate the residence patch method. It is immediately clear from the figure that the track shows location errors, both in the form of point outliers as well as small-scale errors around the true location.

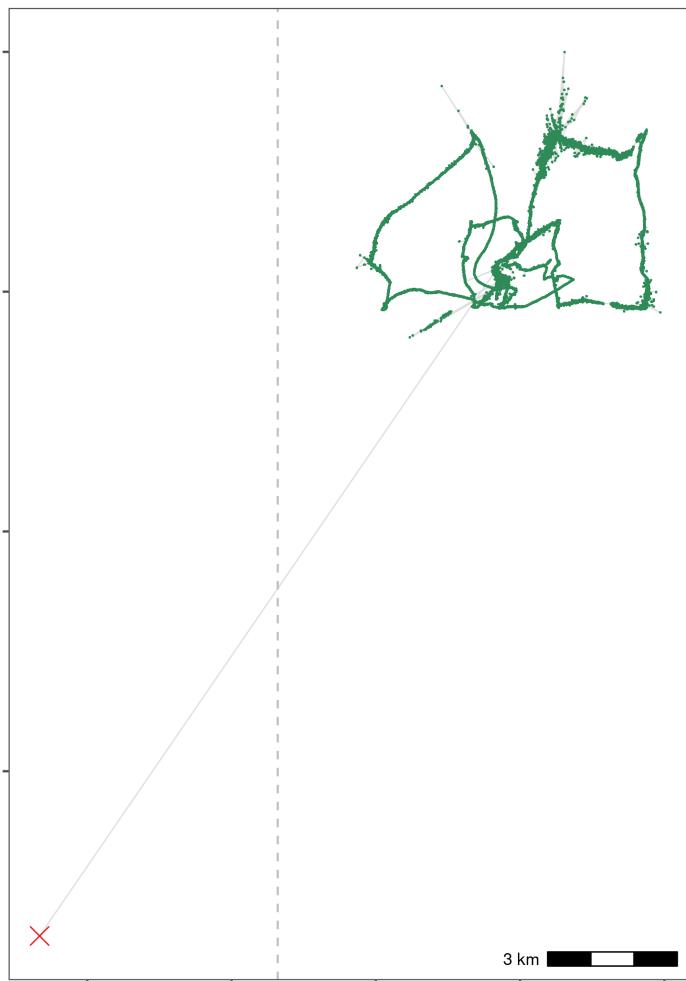


Figure 1.2: Removal of a point outlier using the function `atl_filter_bounds`. The point outlier (black point) is removed based on its X coordinate value, with the data filtered to exclude positions with an X coordinate $< 645,000$ in the UTM 31N coordinate system. Positions that are retained are shown in green.

97 1.6 Filter trajectories

98 1.6.1 Handle time

99 Time in ATLAS tracks is represented by 64-bit integers (type long) that specify
100 time in milliseconds, starting from the beginning of 1970 (the UNIX epoch). This
101 representation of time is called POSIX time and is usually specified in seconds, not
102 milliseconds.

103 Since about 1.6 billion seconds have passed since the beginning of 1970, current
104 POSIX times in milliseconds cannot be represented by R's built-in 32-bit integers.
105 A naive conversion results in truncation of out-of-range numbers leading to huge
106 errors (dates many thousands of years in the future).

107 R does not natively support 64-bit integers. One option is to use the bit64 package,
108 which adds 64-bit integer support to R.

109 A simpler solution is to convert the times to R's built in double data type (also called
110 numeric), which uses a 64-bit floating point representation. This representation can
111 represent integers with up to 16 digits without error; we only need 13 digits to
112 represent the number of milliseconds since 1970, so the conversion is error free.
113 We can also perform the conversion and then divide by 1000 so that times are
114 represented in seconds, not milliseconds; this simplifies speed estimation.

115 If second-resolution is accurate enough (it is for our purposes), the solution that we
116 use is to divide times by 1000 to reduce the resolution from milliseconds to seconds
117 and then to convert the time stamps to R integers. In the spirit of not destroying
118 data, we create a second lower-case column called time to store this

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

119 1.6.2 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)]
```

120 1.6.3 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
    )
}

121  1.6.4 Filter on speed

122  Here we use a speed threshold of 15 m/s, the fastest known boat speed. We then
123  plot the data with the extreme speeds shown in grey, and the positions retained
124  shown in green.

# 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")
)]

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

```

125 1.7 Smoothing the trajectory

126 We then apply a median smooth over a moving window ($K = 5$). This function
127 modifies in place, and does not need to be assigned to a new variable. We create a
128 copy of the data before applying the smooth so that we can compare the data before
129 and after smoothing.

```

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

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

```

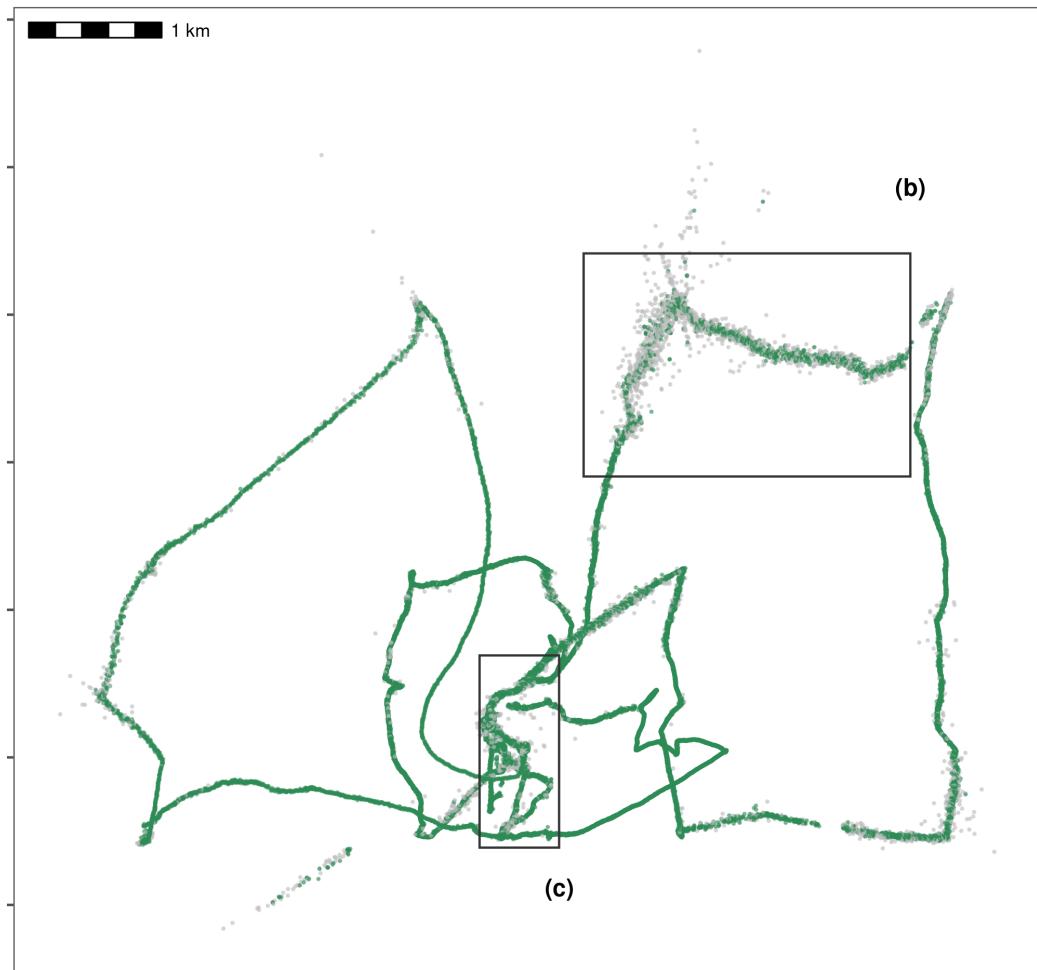


Figure 1.3: Improving data quality by filtering out positions that would require unrealistic movement. We removed positions with speeds ≥ 15 m/s, which is the fastest possible speed in this calibration data, part of which was collected in a moving boat around Griend. Grey positions are removed, while green positions are retained. Rectangles indicate areas expanded for visualisation in following figures.

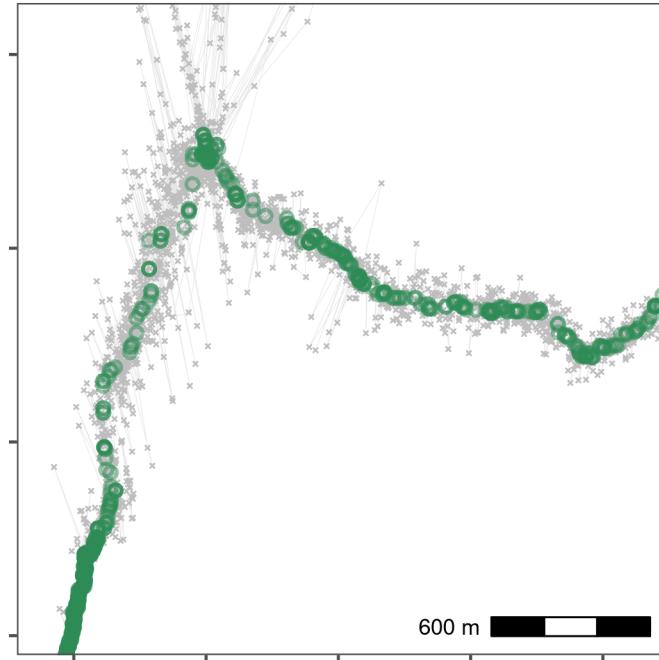


Figure 1.4: Reducing small-scale location error using a median smooth with a moving window $K = 5$. Median smoothed positions are shown in green, while raw, unfiltered data is shown in grey. Median smoothing successfully recovers the likely path of the track without a loss of data. The area shown is the upper rectangle from Fig. 1.3.

130 1.8 Thinning the data

- 131 Next we thin the data to demonstrate thinning by median smoothing. Following
132 this, we plot the median smooth and thinning by aggregation.

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

# remove columns we don't need
data <- data[, setdiff(
  colnames(data),
  c("tID", "Timestamp", "id", "TIME", "UTCtime"))
],
with = FALSE
]

# thin to a 30s interval
data_thin <- atl_thin_data(
  data = data,
  interval = 30,
  method = "aggregate",
  id_columns = "TAG"
)
```

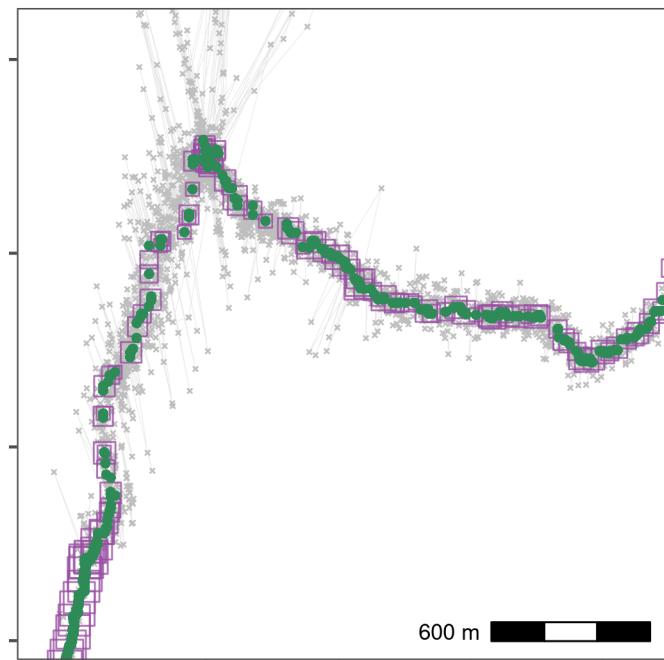


Figure 1.5: Thinning by aggregation over a 30 second interval (down from 1 second) preserves track structure while reducing the data volume for computation. Here, thinned positions are shown as purple squares, with the size of the square indicating the number of positions within the 30 second bin used to obtain the average position. Green points show the median smoothed data from Fig. 1.4, while the raw data are shown in grey. The area shown is the upper rectangle in Fig. 1.3.

133 1.9 Residence patches

134 1.9.1 Get waypoint centroids

135 We subset the annotated calibration data to select the waypoints and the positions
136 around them which are supposed to be the locations of known stops. Since each
137 stop was supposed to be 5 minutes long, there are multiple points in each known
138 stop.

```
library(stringi)
data_res <- data_unproc[stri_detect(tID, regex = "(WP)")]
139 From this data, we get the centroid of known stops, and determine the time differ-
140 ence between the first and last point within 50 metres, and within 10 minutes of the
141 waypoint positions' median time.
142 Essentially, this means that the maximum duration of a stop can be 20 minutes, and
143 stops above this duration are not expected.
```

```
# 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 = TRUE
)
```

144 1.9.2 Prepare data

145 First, we filter data where we know the animal spent some time at or near a position,
146 as this is the first step to identify residence patches. One way of doing this is to
147 filter out positions with speeds above which the animal is likely to be in transit, but

148 a better way of doing this by filtering on a metric such as residence time (Bracis,
149 Bildstein, and Mueller 2018).

150 1.9.3 Calculate residence time

151 First we calculate the residence time with a radius of 50 metres. For this, we need a
152 dataframe with coordinates, the timestamp, and the animal id. We save this data to
153 file for later use.

```
# get 4 column data
data_for_patch <- data_thin[, list(x, y, time, TAG)]  
  
# get recurse data for a 10m radius
recurse_stats <- getRecursions(data_for_patch,
  radius = 50, timeunits = "mins"
)  
  
# assign to recurse data
data_for_patch[, res_time := recurse_stats$residenceTime]  
  
# save recurse data
fwrite(data_for_patch, file = "data/data_calib_for_patch.csv")
```

154 1.9.4 Run residence patch method

155 We subset data with a residence time > 5 minutes in order to construct residence
156 patches. From this subset, we construct residence patches using the parameters:
157 buffer_radius = 5 metres, lim_spat_indep = 50 metres, lim_time_indep = 5 minutes,
158 and min_fixes = 3.

```
# assign id as tag
data_for_patch[, id := as.character(TAG)]  
  
# on known residence points
patch_res_known <- atl_res_patch(data_for_patch[res_time >= 5, ],
  buffer_radius = 5,
  lim_spat_indep = 50,
  lim_time_indep = 5,
  min_fixes = 3
)
```

159 A note on summary statistics

160 Users specifying a summary_variable should make sure that the variable for which
161 they want a summary statistic is present in the data. For instance, requesting mean
162 speed by passing summary_variable = "speed" and summary_function = "mean" to
163 atl_res_patch, should make sure that their data includes a column called speed.

164 **1.9.5 Get spatial and summary objects**

165 We get spatial and summary ouput of the residence patch method using the
166 atl_patch_summary function using the options which_data = "spatial" and
167 which_data = "summary". We use a buffer radius here of 20 metres for the spatial
168 buffer, despite using a buffer radius of 5 metres earlier, simply because it is easier to
169 visualise in the output figure.

```
# 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"
)
```

170 At this stage, users have successfully pre-processed their data from raw positions to
171 residence patches. Residence patches are essentially sf objects and can be visualised
172 using the sf method for plot; for instance plot(patch_sf_data). Further sections
173 reproduce the analyses in the main manuscript.

174

175 **1.9.6 Prepare to plot data**

176 We read in the island's shapefile to plot it as a background for the residence patch
177 figure.

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

178 **1.10 Compare patch metrics**

179 We then merge the annoated, known stop data with the calculated patch duration.
180 We filter this data to exclude one exceedingly long outlier of about an hour (WP080),
181 which how

```
# 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)
), ]
```

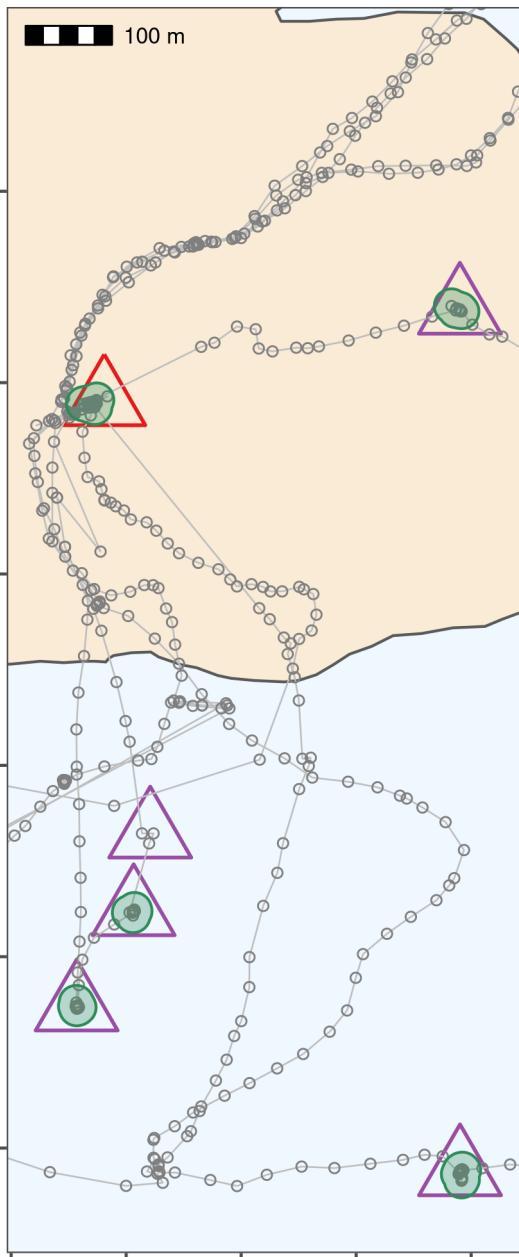


Figure 1.6: Classifying thinned data into residence patches yields robust estimates of the duration of known stops. The island of Griend (53.25°N , 5.25°E) is shown in beige. Residence patches (green polygons; function parameters in text) correspond well to the locations of known stops (purple triangles). However, the algorithm identified all areas with prolonged residence, including those which were not intended stops ($n = 12$; green polygons without triangles). The algorithm also failed to find two stops of 6 and 15 seconds duration, since these were lost in the data thinning step (triangle without green polygon shows one of these). The area shown is the lower rectangle in Fig. 1.3.

```

by = "tID"
]

# add real duration
patch_summary_real[, duration_real := 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
  )

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

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

182 7 patches are identified where there are no waypoints, while 2 waypoints are not
183 identified as patches. These waypoints consisted of 6 and 15 (WP098 and WP092)
184 positions respectively, and were lost when the data were aggregated to 30 second
185 intervals.

```

186 1.10.1 Linear model durations

187 We run a simple linear model.

```

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

# get R2
summary(model_duration)

# write to file
writeLines(

```

```

text = capture.output(
  summary(model_duration)
),
con = "data/model_output_residence_patch.txt"
)

```

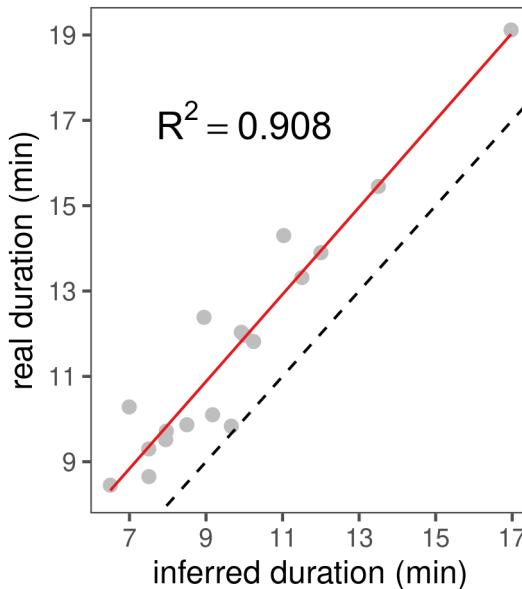


Figure 1.7: The inferred duration of residence patches corresponds very closely to the real duration (grey circles, red line shows linear model fit), with an underestimation of the true duration of around 2%. The dashed black line represents $y = x$ for reference.

188 1.10.2 Linear model summary

```

cat(
readLines(
  con = "data/model_output_residence_patch.txt",
  encoding = "UTF-8"
),
sep = "\n"
)
#>
#> Call:
#> lm(formula = duration_real ~ duration, data = patch_summary_compare)
#>
#> Residuals:
#>      Min       1Q   Median       3Q      Max
#> -103.237  -19.277   -2.917    7.003   93.431
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 101.42061   47.66936   2.128   0.0493 *
#> duration     1.02108    0.07876  12.965 6.66e-10 ***
#> ---

```

```
#> Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1  
#>  
#> Residual standard error: 50.35 on 16 degrees of freedom  
#> Multiple R-squared:  0.9131, Adjusted R-squared:  0.9077  
#> F-statistic: 168.1 on 1 and 16 DF,  p-value: 6.655e-10
```

¹⁸⁹ **1.11 Main text Figure 6**

¹⁹⁰ Plotting code is not shown in PDF and HTML form, see the .Rmd file.

191 2 Processing Egyptian Fruit Bat Tracks

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

194 2.1 Prepare libraries

195 Install the required R libraries that are required from CRAN if not already installed.

```
# libs for data
library(data.table)
library(RSQLite)
library(atlastools)

# libs for plotting
library(ggplot2)
library(patchwork)

# recursion analysis
library(recurse)

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

196 2.2 Read bat data

197 Read the bat data from an SQLite database local file and convert to a plain text csv
198 file. This data can be found in the “data” folder.

```
# 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)

199 Convert data to csv, and save a local copy in the folder "data".

# convert data to datatable
setDT(data)

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

```

200 2.3 A First Visual Inspection

201 Plot the bat data as a sanity check, and inspect it visually for errors (Fig. 2.1). The
 202 plot code is hidden in the rendered copy (PDF) of this supplementary material, but
 203 is available in the Rmarkdown file “06_bat_data.Rmd”. The saved plot is shown below
 204 as Fig. 2.1.

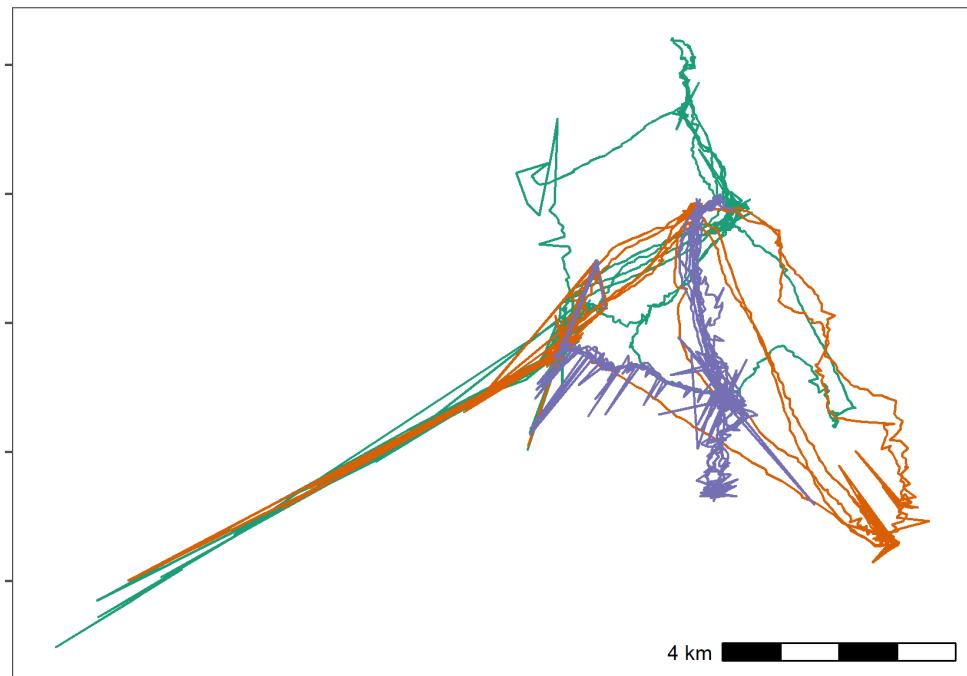


Figure 2.1: Movement data from three Egyptian fruit bats tracked using the ATLAS system (*Rousettus aegyptiacus*; (Toledo et al. 2020; Shohami and Nathan 2020)). The bats were tracked in the Hula Valley, Israel (33.1°N, 35.6°E), and we use three nights of tracking (5th, 6th, and 7th May, 2018), for our demonstration, with an average of 13,370 positions (SD = 2,173; range = 11,195 – 15,542; interval = 8 seconds) per individual. After first plotting the individual tracks, we notice severe distortions, making pre-processing necessary

205 2.4 Prepare data for filtering

206 Here we apply a series of simple filters. It is always safer to deal with one individual
207 at a time, so we split the data.table into a list of data.tables to avoid mixups among
208 individuals.

209 2.4.1 Prepare data per individual

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

# now split
data_split <- split(data_split, by = "TAG")
```

210 2.5 Filter by covariates

211 No natural bounds suggest themselves, so instead we proceed to filter by covariates,
212 since point outliers are obviously visible.

213 We use filter out positions with $SD > 20$ and positions calculated using only 3 base
214 stations, using the function atl_filter_covariates.

215 First we calculate the variable SD.

```
# 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))]
  })
)
```

216 Then we pass the filters to atl_filter_covariates. We apply the filter to each
217 individual's data using an lapply.

```
# 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"
    )
  )
})
```

218 **2.5.1 Sanity check: Plot filtered data**

219 We plot the data to check whether the filtering has improved the data (Fig. 2.2). The
220 plot code is once again hidden in this rendering, but is available in the source code
221 file.

222 **2.6 Filter by speed**

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

224 First we calculate speeds, using atl_get_speed. We must assign the speed output to
225 a new column in the data.table, which has a special syntax which modifies in place,
226 and is shown below. This syntax is a feature of the data.table package, not strictly
227 of atlastools (Dowle and Srinivasan 2020).

```
# 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"
      )
    )]
  })
)
```

228 Now filter for speeds > 20 m/s (around 70 km/h), passing the predicate (a state-
229 ment return TRUE or FALSE) to atl_filter_covariates. First, we remove positions
230 which have NA for their speed_in (the first position) and their speed_out (last posi-
231 tion).

```
# 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"
```

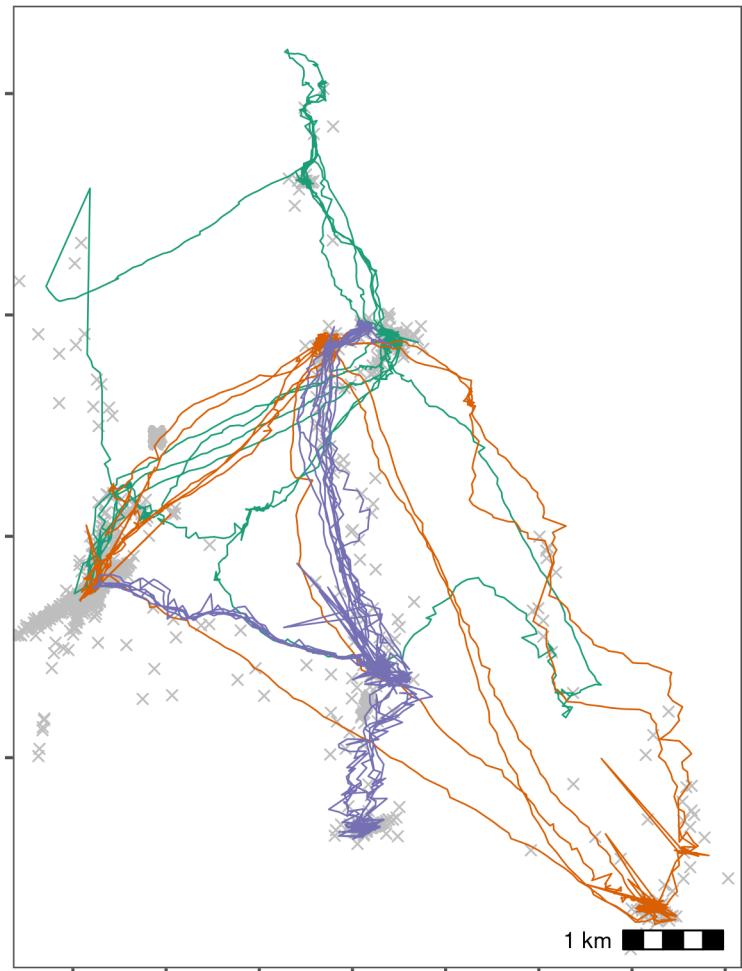


Figure 2.2: Bat data filtered for large location errors, removing observations with standard deviation > 20 . Grey crosses show data that were removed. Since the number of base stations used in the location process is a good indicator of error (Weiser et al. 2016), we also removed observations calculated using fewer than four base stations. Both steps used the function `atl_filter_covariates`. This filtering reduced the data to an average of 10,447 positions per individual (78% of the raw data on average). However, some point outliers remain.

```
)  
)  
})
```

232 2.6.1 Sanity check: Plot speed filtered data

233 The speed filtered data is now inspected for errors (Fig. 2.3). The plot code is once
234 again hidden.

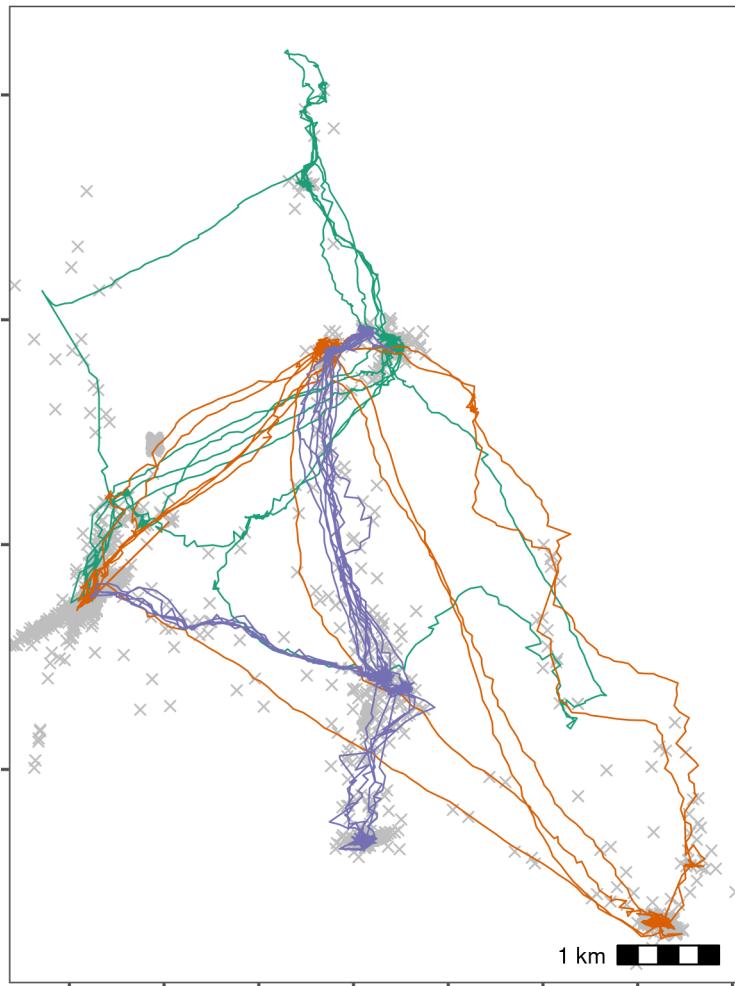


Figure 2.3: Bat data with unrealistic speeds removed. Grey crosses show data that were removed. We calculated the incoming and outgoing speed of each position using `atl_get_speed`, and filtered out positions with speeds $> 20 \text{ m/s}$ using `atl_filter_covariates`, leaving 10,337 positions per individual on average (98% from the previous step).

235 2.7 Median smoothing

- 236 The quality of the data is relatively high, and a median smooth is not strictly
237 necessary. We demonstrate the application of a 5 point median smooth to the
238 data nonetheless (Fig. 2.4).
- 239 Since the median smoothing function atl_median_smooth modifies in place, we first
240 make a copy of the data, using data.table's copy function. No reassignment is
241 required, in this case. The lapply function allows arguments to atl_median_smooth
242 to be passed within lapply itself.
- 243 In this case, the same moving window K is applied to all individuals, but modifying
244 this code to use the multivariate version Map allows different K to be used for
245 different individuals. This is a programming matter, and is not covered here further.

```
# 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, by = "TAG")

# 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(
    X = data_smooth,
    FUN = atl_median_smooth,
    time = "time", moving_window = 5
  )
)
```

246 2.7.1 Sanity check: Plot smoothed data

247 2.8 Making residence patches

248 2.8.1 Calculating residence time

- 249 First, the data is put through the recurse package to get residence time (Bracis,
250 Bildstein, and Mueller 2018).

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

- 251 We calculated residence time, but since bats may revisit the same features, we want
252 to prevent confusion between frequent revisits and prolonged residence.
- 253 For this, we stop summing residence times within Z metres of a location if the
254 animal exited the area for one hour or more. The value of Z (radius, in recurse
255 parameter terms) was chosen as 50m.

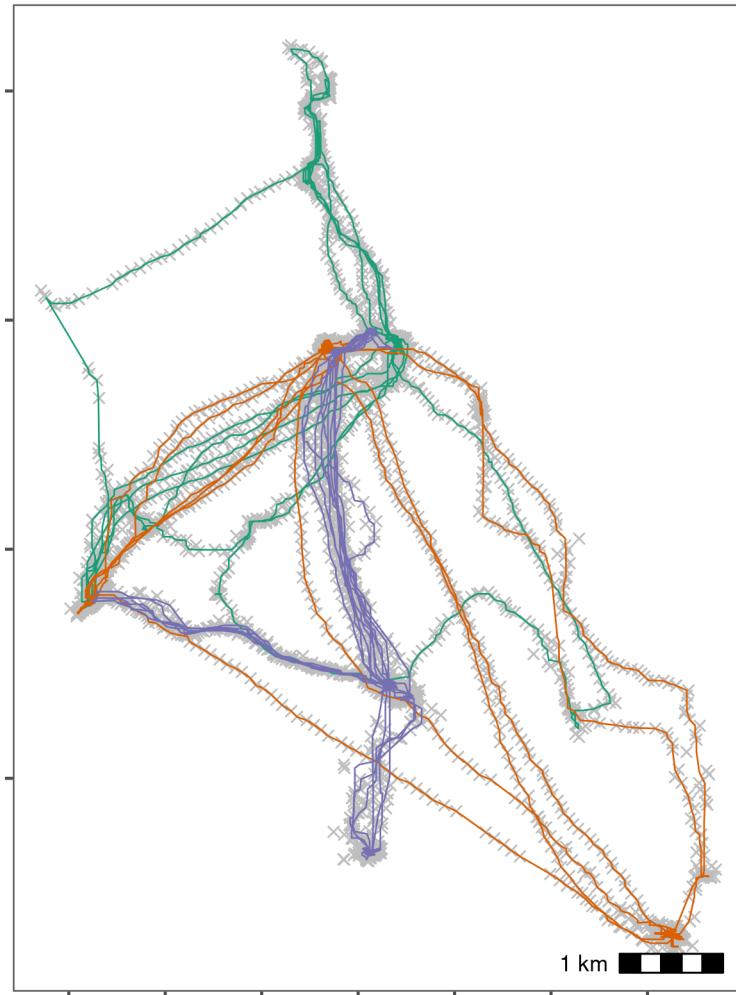


Figure 2.4: Bat data after applying a median smooth with a moving window $K = 5$. Grey crosses show data prior to smoothing. The smoothing step did not discard any data.

256 This step is relatively complicated and is only required for individuals which frequently return to the same location, or pass over the same areas repeatedly, and
257 for which revisits (cumulative time spent) may be confused for residence time in a
258 single visit.

260 While a simpler implementation using total residence time divided by the number
261 of revisits is also possible, this does assume that each revisit had the same residence
262 time.

```
# 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")
})
```

263 We bind the data together and assign a human readable timestamp column.

```

# bind the list
data_residence <- rbindlist(data_residence)

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

```

264 **2.8.2 Constructing residence patches**

265 Some preparation is required. First, the function requires columns x, y, time, and id,
 266 which we assign using the data.table syntax. Then we subset the data to only work
 267 with positions where the individual had a residence time of more than 5 minutes.

```

# 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)

```

268 We apply the residence patch method, using the default argument values
 269 (lim_spat_indep = 100 (metres), lim_time_indep = 30 (minutes), and min_fixes =
 270 3). We change the buffer_radius to 25 metres (twice the buffer radius is used, so
 271 points must be separated by 50m to be independent bouts).

```

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

```

272 **2.8.3 Getting residence patch data**

273 We extract the residence patch data as spatial sf-MULTIPOLYGON objects. These are
 274 returned as a list and must be converted into a single sf object. These objects and
 275 the raw movement data are shown in Fig. 2.5.

```

# 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)

276 2.8.4 Write patch spatial representations

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

277 Write cleaned bat data.

fwrite(rbindlist(data_smooth),
  file = "data/data_bat_smooth.csv"
)

278 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"
)

```

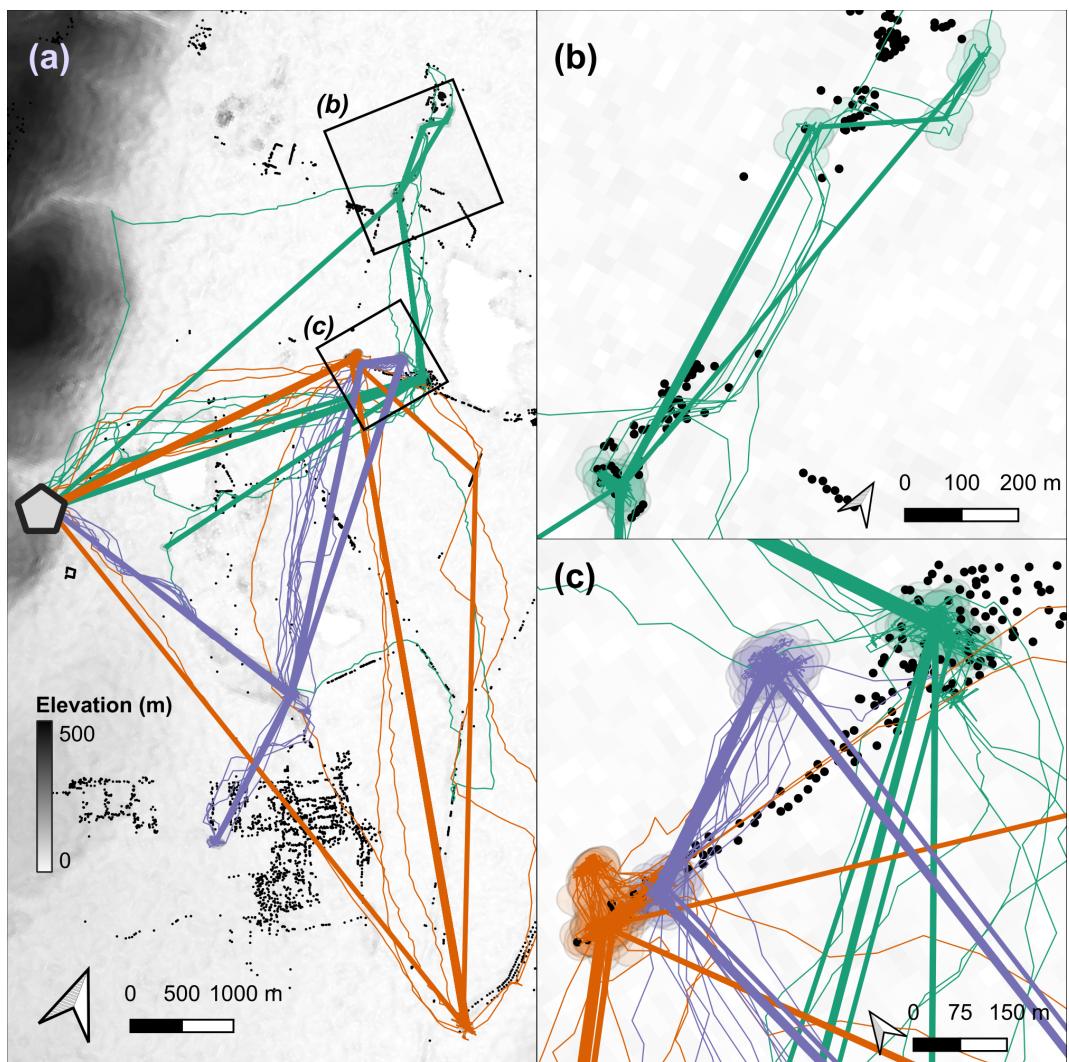


Figure 2.5: A visual examination of plots of the bats' residence patches and linear approximations of paths between them showed that though all three bats roosted at the same site, they used distinct areas of the study site over the three nights (a). Bats tended to be resident near fruit trees, which are their main food source, travelling repeatedly between previously visited areas (b, c). However, bats also appeared to spend some time at locations where no fruit trees were recorded, prompting questions about their use of other food sources (b, c). When bats did occur close together, their residence patches barely overlapped, and their paths to and from the broad area of co-occurrence were not similar (c). Constructing residence patches for multiple individuals over multiple activity periods suggests interesting dynamics of within- and between-individual overlap (b, c).

²⁷⁹ 3 References

- ²⁸⁰ Barraquand, Frédéric, and Simon Benhamou. 2008. "Animal Movements in Heterogeneous Landscapes: Identifying Profitable Places and Homogeneous Movement Bouts." *Ecology* 89 (12): 3336–48. <https://doi.org/10.1890/08-0162.1>.
- ²⁸³ Beardsworth, Christine E., Evy Gobbens, Frank van Maarseveen, Bas Denissen, Anne Dekkinga, Ran Nathan, Sivan Toledo, and Allert I. Bijleveld. 2021. "Validating a High-Throughput Tracking System: ATLAS as a Regional-Scale Alternative to GPS." *bioRxiv*, February, 2021.02.09.430514. <https://doi.org/10.1101/2021.02.09.430514>.
- ²⁸⁷ Bijleveld, Allert Imre, Robert B MacCurdy, Ying-Chi Chan, Emma Penning, Richard M. Gabrielson, John Cluderay, Erik L. Spaulding, et al. 2016. "Understanding Spatial Distributions: Negative Density-Dependence in Prey Causes Predators to Trade-Off Prey Quantity with Quality." *Proceedings of the Royal Society B: Biological Sciences* 283 (1828): 20151557. <https://doi.org/10.1098/rspb.2015.1557>.
- ²⁹² Bracis, Chloe, Keith L. Bildstein, and Thomas Mueller. 2018. "Revisitation Analysis Uncovers Spatio-Temporal Patterns in Animal Movement Data." *Ecography* 41 (11): 1801–11. <https://doi.org/10.1111/ecog.03618>.
- ²⁹⁵ Dowle, Matt, and Arun Srinivasan. 2020. *Data.Table: Extension of 'data.Frame'*. Manual.
- ²⁹⁷ Gupte, Pratik Rajan. 2020. "Atlastools: Pre-Processing Tools for High Frequency Tracking Data." Zenodo. <https://doi.org/10.5281/ZENODO.4033154>.
- ²⁹⁹ Oudman, Thomas, Theunis Piersma, Mohamed V. Ahmedou Salem, Marieke E. Feis, Anne Dekkinga, Sander Holthuijsen, Job ten Horn, Jan A. van Gils, and Allert I. Bijleveld. 2018. "Resource Landscapes Explain Contrasting Patterns of Aggregation and Site Fidelity by Red Knots at Two Wintering Sites." *Movement Ecology* 6 (1): 24–24. <https://doi.org/10.1186/s40462-018-0142-4>.
- ³⁰⁴ Shohami, David, and Ran Nathan. 2020. "Cognitive Map-Based Navigation in Wild Bats Revealed by a New High-Throughput Tracking System." Dryad. <https://doi.org/10.5061/DRYAD.G4F4QRFN2>.
- ³⁰⁷ Toledo, Sivan, David Shohami, Ingo Schiffner, Emmanuel Lourie, Yotam Orchan, Yoav Bartan, and Ran Nathan. 2020. "Cognitive Map-Based Navigation in Wild Bats Revealed by a New High-Throughput Tracking System." *Science* 369 (6500): 188–93. <https://doi.org/10.1126/science.aax6904>.
- ³¹¹ Weiser, Adi Weller, Yotam Orchan, Ran Nathan, Motti Charter, Anthony J. Weiss, and Sivan Toledo. 2016. "Characterizing the Accuracy of a Self-Synchronized Reverse-GPS Wildlife Localization System." In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 1–12. <https://doi.org/10.1109/IPSN.2016.7460662>.