

Supplementary Material for: A Rough Guide to Pre-processing High-Frequency Animal Tracking Data

Pratik R. Gupte Christine E. Beardsworth Orr Spiegel
Emmanuel Lourie Allert I. Bijleveld

2020-11-23

Contents

1	Processing calibration data	1
1.1	Prepare libraries	1
1.2	Access data and preliminary visualisation	1
1.3	Filter by bounding box	3
1.4	Filter trajectories	3
1.5	Smoothing the trajectory	5
1.6	Thinning the data	7
1.7	Residence patches	9
1.8	Compare patch metrics	11
2	Processing Egyptian fruit bat tracks	14
2.1	Prepare libraries	14
2.2	Install atlastools from Github.	14
2.3	Read bat data	14
2.4	A First Visual Inspection	15
2.5	Prepare data for filtering	16
2.6	Filter by covariates	16
2.7	Filter by speed	17
2.8	Median smoothing	18
2.9	Making residence patches	20
3	References	24

1 Processing calibration data

Here we show how the residence patch method (Barraquand and Benhamou 2008; Bijleveld et al. 2016; Oudman et al. 2018) accurately estimates the duration of known stops in a track collected as part of a calibration exercise in the Wadden Sea.

32 1.1 Prepare libraries

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

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

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

34 1.2 Access data and preliminary visualisation

35 First we access the data from a local file using the data.table package (Dowle and Srinivasan 2020). We then visualise the raw data.

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

37 1.3 Filter by bounding box

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

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

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

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

```
# 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)
```

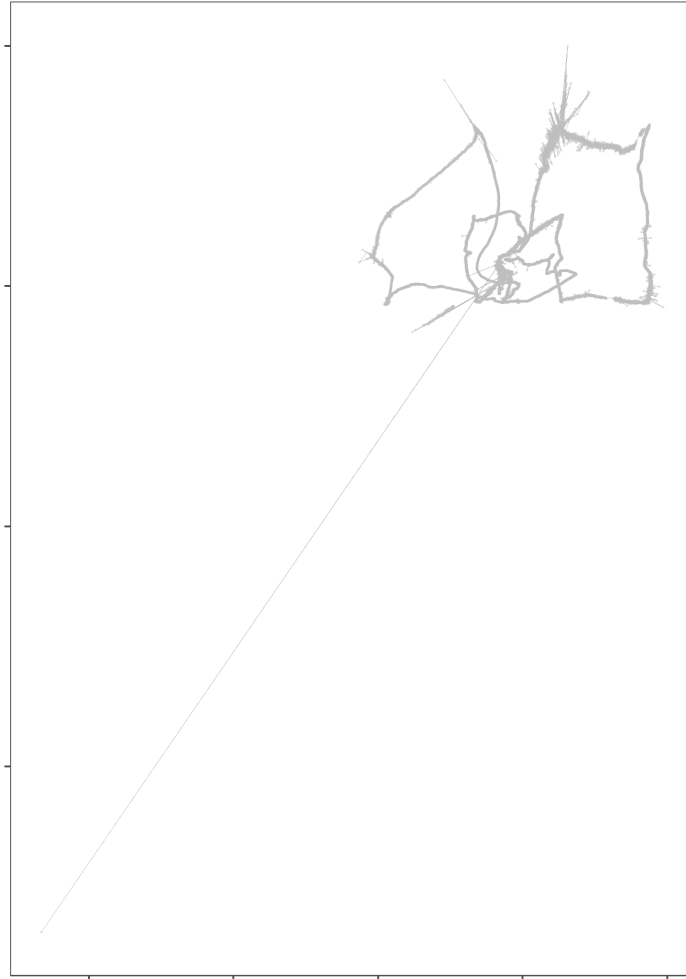


Figure 1: The raw data from a calibration exercise conducted around the island of Griend in the Dutch Wadden Sea. A handheld WATLAS tag was used to examine how ATLAS data compared to GPS tracks, and we use the WATLAS data here to demonstrate the basics of the pre-processing pipeline, as well as validate 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.

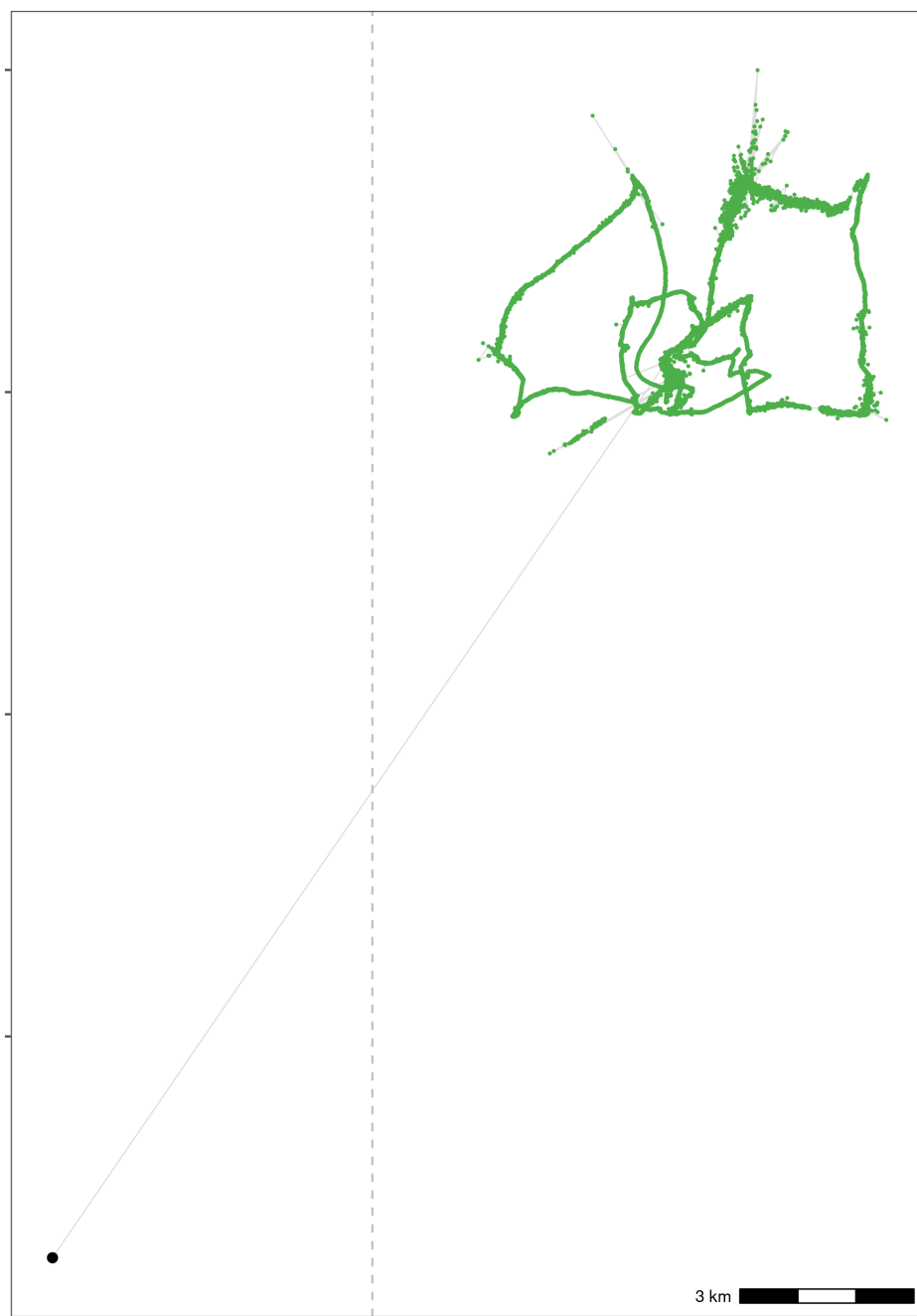


Figure 2: Removal of a point outlier using the function `atl_filter_bounds`. The point outlier (black point) is removed based on its X coordinate value, with the data filtered to exclude positions with an X coordinate $< 645,000$ in the UTM 31N system. Positions that are retained are shown in green.

46 1.4 Filter trajectories

47 Handle time

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

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

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

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

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

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

63 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)]
```

64 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)]
```

65 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)
```

66 Filter on speed

67 Here we use a speed threshold of 15 m/s, the fastest known boat speed. We then plot the
68 data with the extreme speeds shown in grey, and the positions retained 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)]
```

69 1.5 Smoothing the trajectory

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

```
# 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)
```

73 1.6 Thinning the data

74 Next we thin the data to demonstrate thinning by median smoothing. Following this, we
75 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
```



Figure 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.

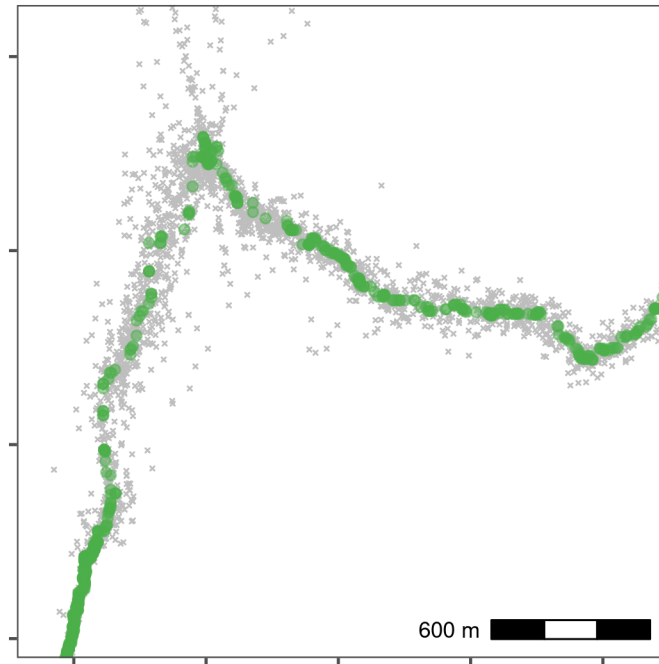


Figure 4: Reducing small-scale location error using a median smooth with a moving window $K = 5$. Median smoothed positions are shown in green, while raw, unfiltered data is shown in grey. Median smoothing successfully recovers the likely path of the track without a loss of data. The area shown is the upper rectangle from Figure 3.

```
data_thin <- atl_thin_data(data = data,
                           interval = 30,
                           method = "aggregate",
                           id_columns = "TAG")
```

76 1.7 Residence patches

77 Get waypoint centroids

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

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

81 From this data, we get the centroid of known stops, and determine the time difference be-
 82 tween the first and last point within 50 metres, and within 10 minutes of the waypoint posi-
 83 tions' median time.

84 Essentially, this means that the maximum duration of a stop can be 20 minutes, and stops
 85 above this duration are not expected.

```
# get centroid
data_res_summary <- data_res[, list(x_median = round(median(x), digits = -2),
                                   y_median = round(median(y), digits = -2)),
```

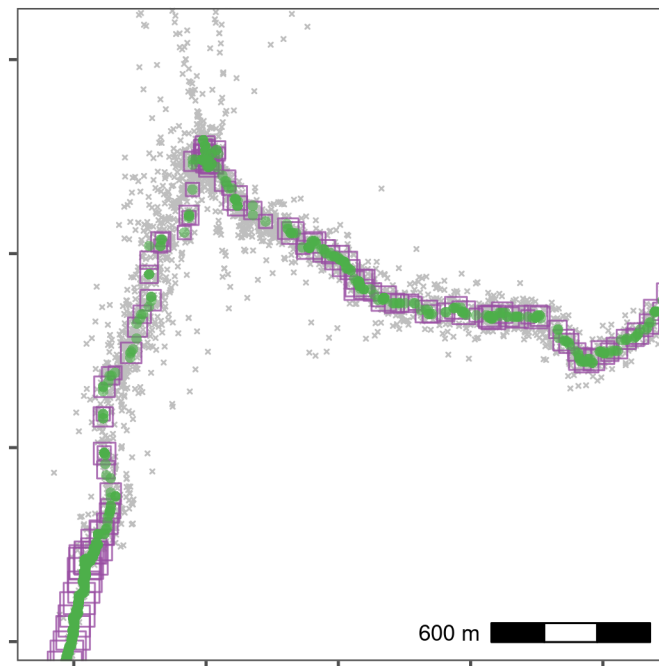



Figure 5: Thinning by aggregation over a 30 second interval (down from 1 second) preserves track structure while reducing the data volume for computation. Here, thinned positions are shown as purple squares, with the size of the square indicating the number of positions within the 30 second bin used to obtain the average position. Green points show the median smoothed data from Figure 4, while the raw data are shown in grey. The area shown is the upper rectangle in Figure 3.

```

                                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)

```

86 Prepare data

87 An indicator of individual residence at or near a position can be useful when attempting
 88 to identify residence patches. Positions can be filtered on a metric such as residence time
 89 (Bracis, Bildstein, and Mueller 2018).

90 Calculate residence time

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

```

# load recurse
library(recurse)

# 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")

```

93 Run residence patch method

94 We subset data with a residence time > 5 minutes in order to construct residence patches.
95 From this subset, we construct residence patches using the parameters: buffer_radius =
96 5 metres, lim_spat_indep = 50 metres, lim_time_indep = 5 minutes, and min_fixes
97 = 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)
```

98 Get spatial and summary objects

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

```
# for the known and unknown 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")
```

103 Prepare to plot data

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

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

105 1.8 Compare patch metrics

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

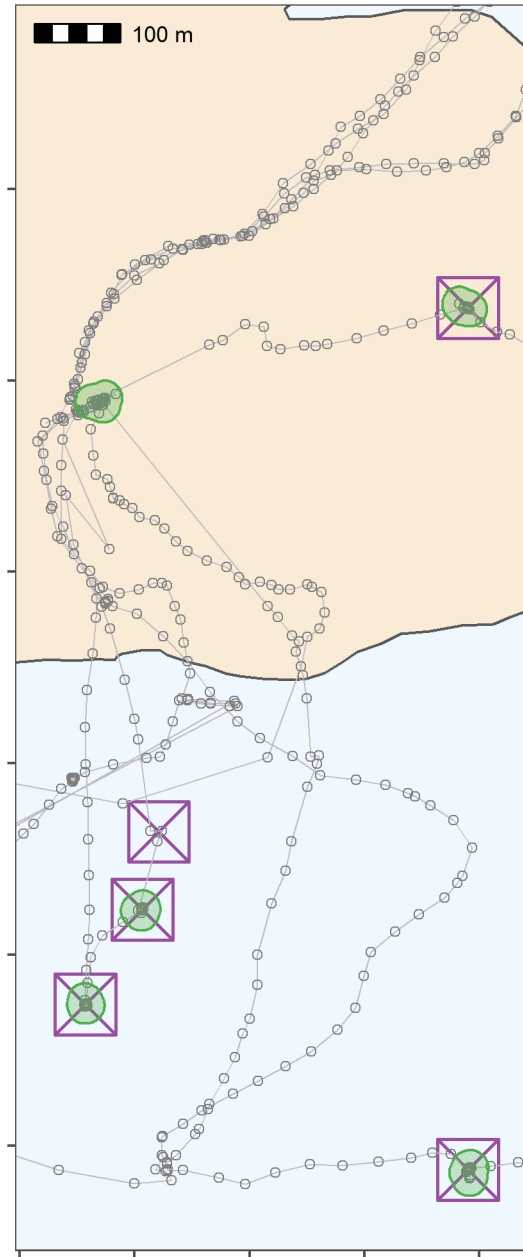


Figure 6: Classifying thinned data into residence patches yields robust estimates of the duration of known stops. The island of Griend (53.25°N , 5.25°E) is shown in beige. Residence patches (green polygons; function parameters in text) correspond well to the locations of known stops (purple crossed-squares). However, the algorithm identified all areas with prolonged residence, including those which were not intended stops ($n = 12$; green polygons without crossed-squares). The algorithm also failed to find two stops of 6 and 15 seconds duration, since these were lost in the data thinning step (crossed-square without green polygon shows one of these). The area shown is the lower rectangle in Figure 3.

```

# 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 := 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", ]

```

108 7 patches are identified where there are no waypoints, while 2 waypoints are not identified as
 109 patches. These waypoints consisted of 6 and 15 (WP098 and WP092) positions respectively,
 110 and were lost when the data were aggregated to 30 second intervals.

111 Linear model durations

112 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"
)

```

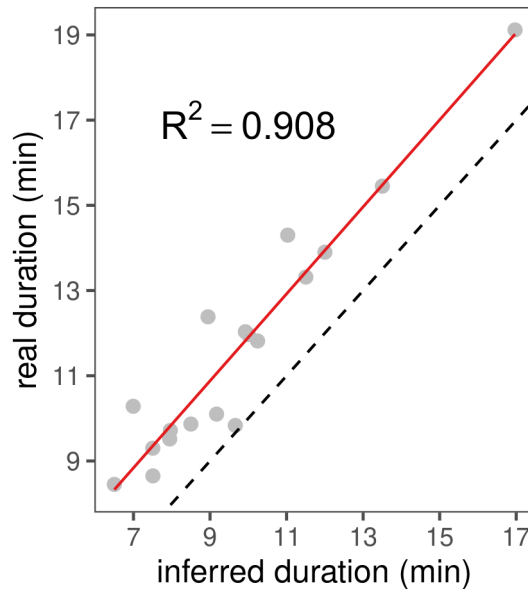


Figure 7: The inferred duration of residence patches corresponds very closely to the real duration (grey circles, red line shows linear model fit), with an underestimation of the true duration of around 2%. The dashed black line represents $y = x$ for reference.

113 Linear model summary

```

cat(
  readLines(
    con = "data/model_output_residence_patch.txt"
  ), sep = "\n"
)

```

114 2 Processing Egyptian fruit bat tracks

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

117 2.1 Prepare libraries

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

```

# load libs
library(data.table)
library(RSQLite)
library(ggplot2)

```

```
library(patchwork)

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

119 2.2 Install atlastools from Github.

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

```
install.packages("remotes")

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

123 2.3 Read bat data

124 Read the bat data from an SQLite database local file and convert to a plain text csv file.
 125 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)
```

126 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")
```

127 2.4 A First Visual Inspection

128 Plot the bat data as a sanity check, and inspect it visually for errors (Figure 1). The plot code
 129 is hidden in the rendered copy (PDF) of this supplementary material, but is available in the
 130 Rmarkdown file “06_bat_data.Rmd”. The saved plot is shown below as Figure 1.

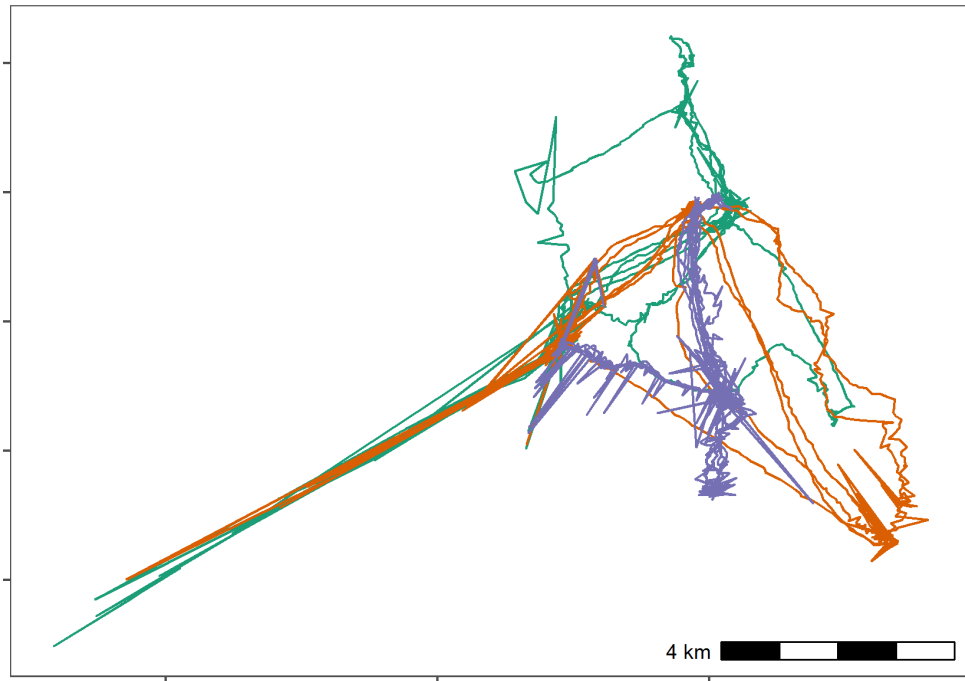


Figure 8: Movement data from three Egyptian fruit bats tracked using the ATLAS system (*Rousettus aegyptiacus*; (Toledo et al. 2020; Shohami and Nathan 2020)). The bats were tracked in the Hula Valley, Israel (33.1°N, 35.6°E), and we use three nights of tracking (5th, 6th, and 7th May, 2018), for our 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

131 2.5 Prepare data for filtering

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

134 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")
```

135 2.6 Filter by covariates

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

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

140 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))]  
  })  
)
```

141 Then we pass the filters to atl_filter_covariates. We apply the filter to each individ-
142 ual'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")  
  )  
})
```

143 **Sanity check: Plot filtered data**

144 We plot the data to check whether the filtering has improved the data (Figure 2). The plot
145 code is once again hidden in this rendering, but is available in the source code file.

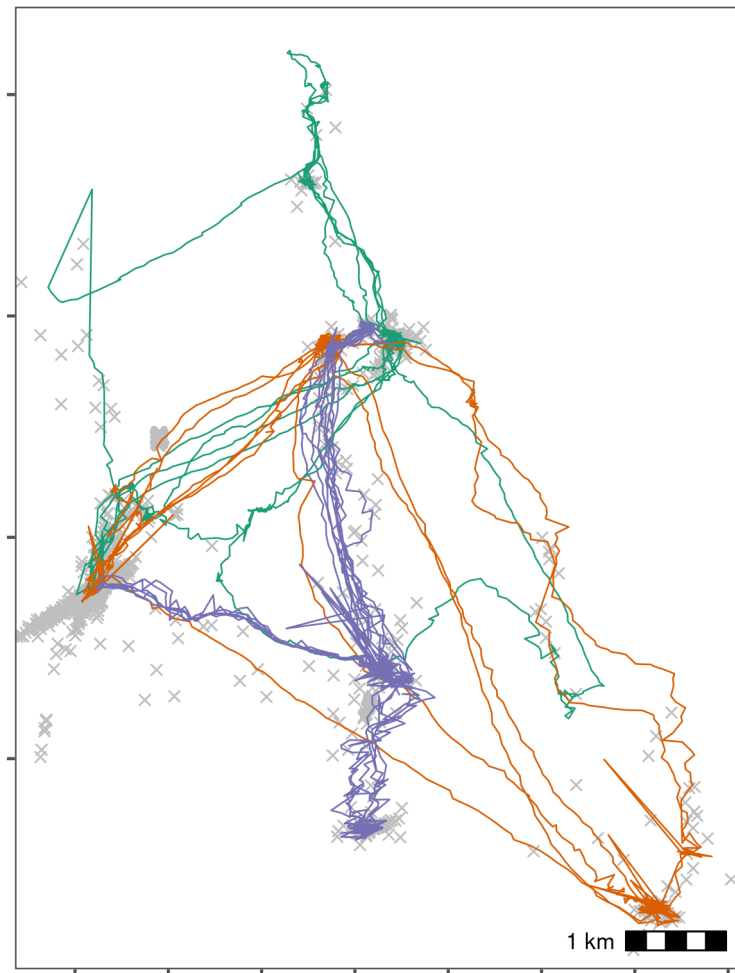


Figure 9: Bat data filtered for large location errors, removing observations with standard deviation > 20 . Grey crosses show data that were removed. Since the number of base stations used in the location process is a good indicator of error (Weiser et al. 2016), we also removed observations calculated 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.

146 **2.7 Filter by speed**

147 Some point outliers remain (Figure 2), and could be removed using a speed filter.

148 First we calculate speeds, using `atl_get_speed`. We must assign the speed output to a new
149 column in the `data.table`, which has a special syntax which modifies in place, and is shown

150 below. This syntax is a feature of the `data.table` package, not strictly of `atlastools`
151 ([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"))]

  })
)
```

152 Now filter for speeds > 20 m/s (around 70 km/h), passing the predicate (a statement return
153 TRUE or FALSE) to `atl_filter_covariates`. First, we remove positions which have
154 NA for their `speed_in` (the first position) and their `speed_out` (last position).

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

155 **Sanity check: Plot speed filtered data**

156 The speed filtered data is now inspected for errors (Figure 3). The plot code is once again
157 hidden.

158 **2.8 Median smoothing**

159 The quality of this data (Figure 3) is relatively high, and a median smooth is not strictly
160 necessary. We demonstrate the application of a 5 point median smooth to the data nonethe-
161 less.

162 Since the median smoothing function `atl_median_smooth` modifies in place, we first make
163 a copy of the data, using `data.table`'s `copy` function. No reassignment is required, in this
164 case. The `lapply` function allows arguments to `atl_median_smooth` to be passed within
165 `lapply` itself.

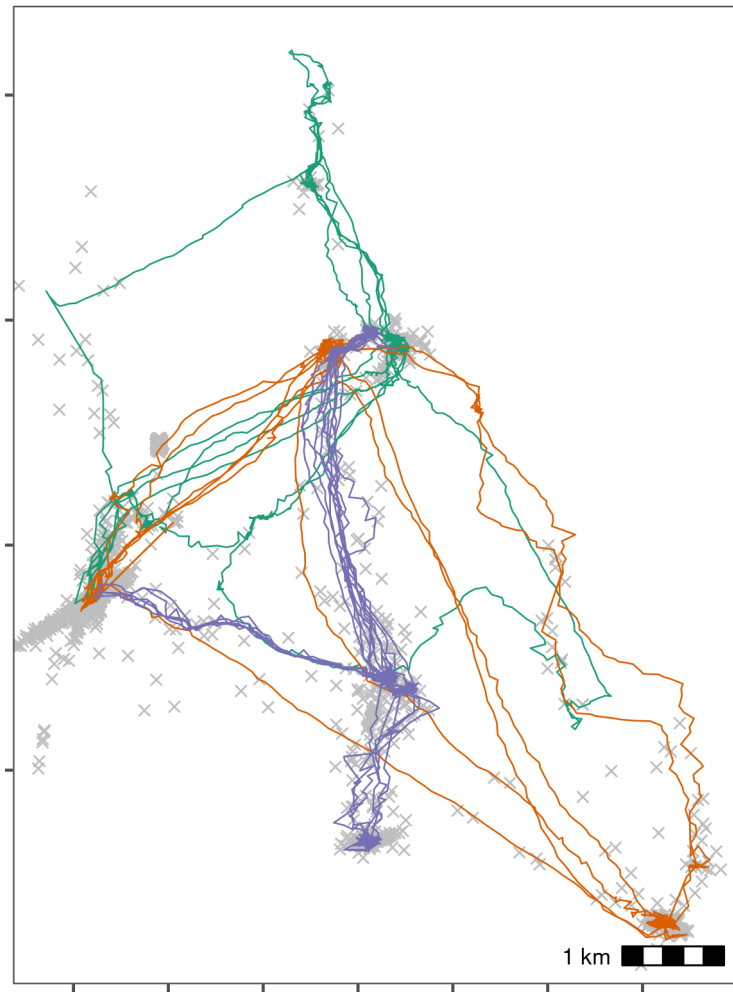


Figure 10: Bat data with unrealistic speeds removed. Grey crosses show data that were removed. We calculated the incoming and outgoing speed of each position using `atl_get_speed`, and filtered out positions with speeds > 20 m/s using `atl_filter_covariates`, leaving 10,337 positions per individual on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

166 In this case, the same moving window K is applied to all individuals, but modifying this code
167 to use the multivariate version Map allows different K to be used for different individuals.
168 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)
)
```

169 **Sanity check: Plot smoothed data**

170 **2.9 Making residence patches**

171 **Calculating residence time**

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

```
# load recurse
library(recurse)

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

174 We calculated residence time, but since bats may revisit the same features, we want to pre-
175 vent confusion between frequent revisits and prolonged residence.

176 For this, we stop summing residence times within Z metres of a location if the animal exited
177 the area for one hour or more. The value of Z (radius, in recurse parameter terms) was
178 chosen as 50m.

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

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

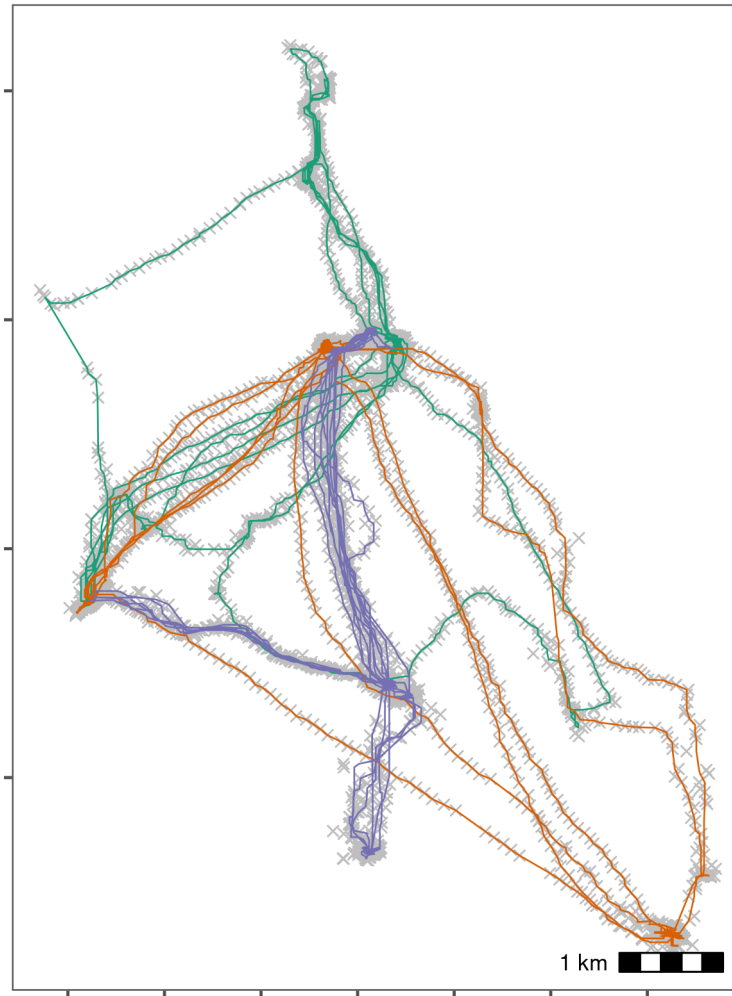


Figure 11: Bat data after applying a median smooth with a moving window $K = 5$. Grey crosses show data prior to smoothing. The smoothing step did not discard any data.

```

# 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")
})

```

184 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")]

```

185 Constructing residence patches

186 Some preparation is required. First, the function requires columns `x`, `y`, `time`, and `id`, which
187 we assign using the `data.table` syntax. Then we subset the data to only work with positions
188 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)
```

189 We apply the residence patch method, using the default argument values (`lim_spat_indep`
190 = 100 (metres), `lim_time_indep` = 30 (minutes), and `min_fixes` = 3). We change the
191 `buffer_radius` to 25 metres (twice the buffer radius is used, so points must be separated
192 by 50m to be independent bouts).

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

193 Getting residence patch data

194 We extract the residence patch data as spatial `sf`-MULTIPOLYGON objects. These are returned
195 as a list and must be converted into a single `sf` object. These objects and the raw movement
196 data are shown in Figure 5.

```
# get data spatial
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)
```

197 Write patch spatial representations

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


198 Write cleaned bat data.

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

199 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")
```

200 3 References

- 201 Barraquand, Frédéric, and Simon Benhamou. 2008. "Animal Movements in Heterogeneous
202 Landscapes: Identifying Profitable Places and Homogeneous Movement Bouts." *Ecol-*
203 *ogy* 89 (12): 3336–48. <https://doi.org/10.1890/08-0162.1>.
- 204 Bijleveld, Allert Imre, Robert B MacCurdy, Ying-Chi Chan, Emma Penning, Richard M.
205 Gabrielson, John Cluderay, Erik L. Spaulding, et al. 2016. "Understanding Spatial Dis-
206 tributions: Negative Density-Dependence in Prey Causes Predators to Trade-Off Prey
207 Quantity with Quality." *Proceedings of the Royal Society B: Biological Sciences* 283
208 (1828): 20151557. <https://doi.org/10.1098/rspb.2015.1557>.
- 209 Bracis, Chloe, Keith L. Bildstein, and Thomas Mueller. 2018. "Revisitation Analysis Un-
210 covers Spatio-Temporal Patterns in Animal Movement Data." *Ecography* 41 (11): 1801–
211 11. <https://doi.org/10.1111/ecog.03618>.
- 212 Dowle, Matt, and Arun Srinivasan. 2020. *Data.Table: Extension of 'data.Frame'.* Man-
213 ual.
- 214 Gupte, Pratik Rajan. 2020. "Atltools: Pre-Processing Tools for High Frequency Tracking
215 Data." Zenodo. <https://doi.org/10.5281/ZENODO.4033154>.
- 216 Oudman, Thomas, Theunis Piersma, Mohamed V. Ahmedou Salem, Marieke E. Feis, Anne
217 Dekinga, Sander Holthuijsen, Job ten Horn, Jan A. van Gils, and Allert I. Bijleveld. 2018.
218 "Resource Landscapes Explain Contrasting Patterns of Aggregation and Site Fidelity by
219 Red Knots at Two Wintering Sites." *Movement Ecology* 6 (1): 24–24. <https://doi.org/10.1186/s40462-018-0142-4>.
- 221 Shohami, David, and Ran Nathan. 2020. "Cognitive Map-Based Navigation in Wild Bats
222 Revealed by a New High-Throughput Tracking System." Dryad. <https://doi.org/10.5061/DRYAD.G4F4QRFN2>.
- 224 Toledo, Sivan, David Shohami, Ingo Schiffner, Emmanuel Lourie, Yotam Orchan, Yoav
225 Bartan, and Ran Nathan. 2020. "Cognitive MapBased Navigation in Wild Bats Revealed
226 by a New High-Throughput Tracking System." *Science* 369 (6500): 188–93. <https://doi.org/10.1126/science.aax6904>.

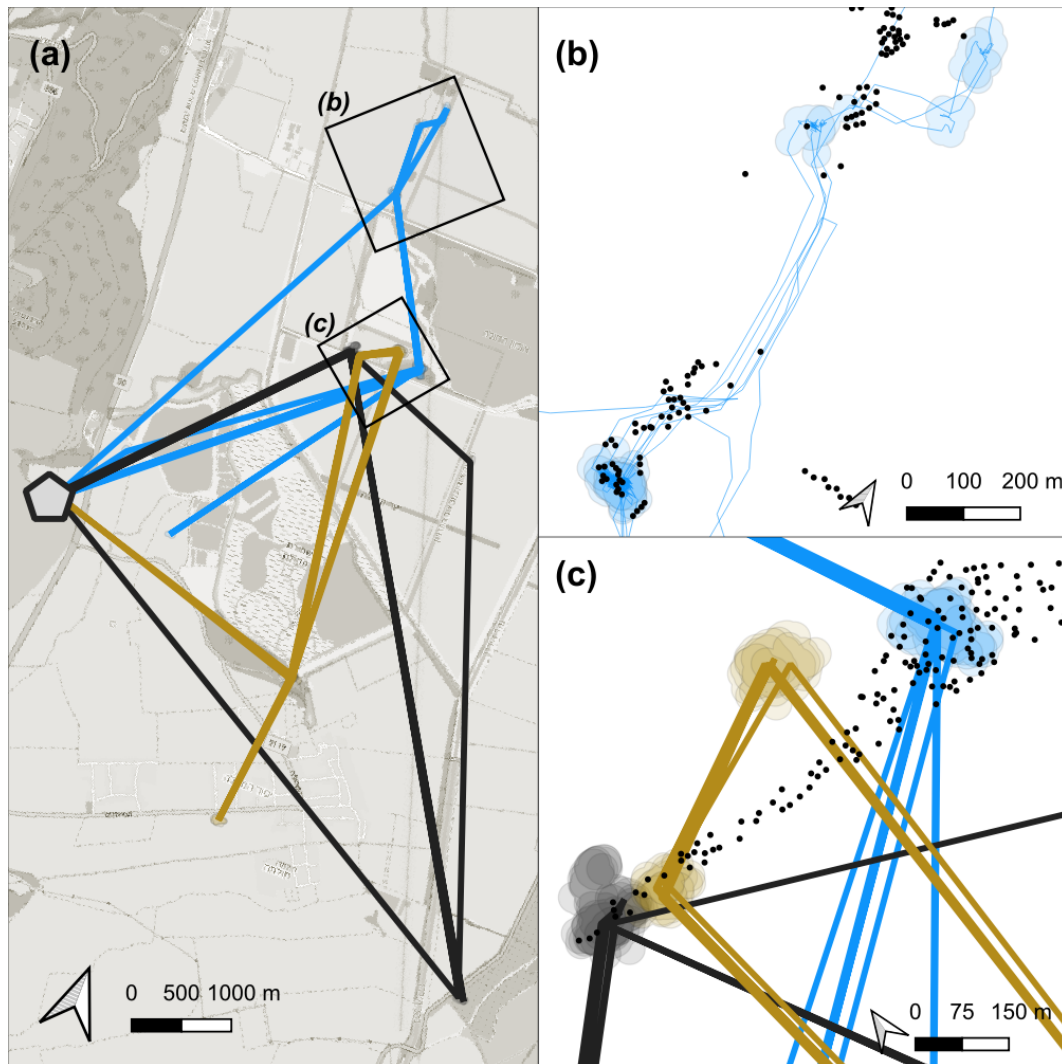


Figure 12: A visual examination of plots of the bats' residence patches and linear approximations of paths between them showed that though all three bats roosted at the same site, they used distinct areas of the study site over the three nights **(a)**. Bats tended to be resident near fruit trees, which 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)**.

228 Weiser, Adi Weller, Yotam Orchan, Ran Nathan, Motti Charter, Anthony J. Weiss, and
229 Sivan Toledo. 2016. “Characterizing the Accuracy of a Self-Synchronized Reverse-
230 GPS Wildlife Localization System.” In *2016 15th ACM/IEEE International Conference*
231 *on Information Processing in Sensor Networks (IPSN)*, 1–12. [https://doi.org/10.1109/](https://doi.org/10.1109/IPSN.2016.7460662)
232 [IPSN.2016.7460662](https://doi.org/10.1109/IPSN.2016.7460662).