Cellular Tracking Technologies: Data Tools

Jessica Gorzo

2024-07-09

Contents

Instructions **5**

1 API **7**
1.1 Your Token . . . 7
1.2 R script: api_run.R . . . 7
1.3 Terminal . . . 8
1.4 Local Database Option . . . 8
1.5 Data Cleaning . . . 10
1.6 Incorporating Node Data Into Your Database . . . 10

2 Tools to Work with Data **11**
How to use GitHub . . . 11
2.1 Pulling Data into R from your Local Database . . . 12

3 About the Functions **13**
3.1 Data Manager . . . 13
3.2 Node Health . . . 14

4 Calibration **19**
4.1 Description . . . 19
4.2 Tag Calibration . . . 19
4.3 Node Calibration . . . 19
4.4 How to Use the Output . . . 19

5 Localization Methods **21**
5.1 Primitive: Weighted Average . . . 21
5.2 Triangulation Functions . . . 22

Instructions

A RStudio tutorial is beyond the scope of this readme, but there are great resources to get you started with installing R and RStudio.

To install the R package, please run:

```
library(devtools)
install_github('cellular-tracking-technologies/celltracktech')
```

Chapter 1

API

1.1 Your Token

Please request an API token through this form. The token will appear on your account page when the request is fulfilled.

1.2 R script: api_run.R

1. Assign the API token you found above to the "my_token" variable
2. Set your "outpath" variable to wherever your files will live. If you have already been manually downloading files, use that as your "outpath."

- The script will search that directory, and will only download files you haven't already downloaded.
- It will create a nested folder structure within that directory in the following order: project name, station(s), file types, files

3. If you do not want to create a database…

- do not set the "conn" variable
- remove the "conn" argument from the get_my_data() function
- do not run update_db()
- comment out the dbDisconnect() line

4. If you do want to create a database locally, set "db_name" to the name of the Postgres database you create (see section "Local Database Option")

An example script using the API tools to download files:

```
library(celltracktech)
library(DBI)
start <- Sys.time()
```

```
####SETTINGS#####
my_token <- "your token here"
db_name <- "mydb"
myproject <- "CTT Project Name" #this is your project name on your CTT account
conn <- dbConnect(RPostgres::Postgres(), dbname=db_name)
################
outpath <- "~/Documents/data/radio_projects/myproject" #where your downloaded files ar
get_my_data(my_token, "~/Documents/data/radio_projects/myproject", conn, myproject=myp
update_db(conn, outpath, myproject)
dbDisconnect(conn)

#findfiles(outpath, "directory path where you want your caught files to go")

time_elapse <- Sys.time() - start
print(time_elapse)
```

1.3 Terminal

Run "Rscript <path to your copy of api_run.R>" on the command line to run the script outside of RStudio (recommended)

1.4 Local Database Option

Specify which of your CTT projects will be loaded into the database with the "myproject" option set to the project name on your account. (To look up your project name by station, go to "My Stations" on your account and click on the station of interest to see which project it's affiliated with.) If you choose to create a database out of your data (fair warning: in the future, the analysis tools will be based on this structure) you will need to install PostgreSQL on your machine.
Disclaimer I am providing some basic instructions for getting setup with Postgres on Windows and creating a user and database here, but please use at your own discretion and do not contact me with Postgres install questions. If any of these steps fail, seek another tutorial for installing Postgres and cross-reference with the steps here.

1. [Download and install] (https://www.postgresql.org/download/) for your OS

2. If pgAdmin wasn't installed with your PostgreSQL installation, it's a nice GUI for interacting with and visualizing your database
3. For simplicity, create a Postgres user with the same name as your computer user name. Otherwise, you will need to pass it as an argument to the connection

4. Create a database in Postgres owned by that user name you created in the previous step. You may have to set a password, and you may have to pass that password as an argument to the connection

- you may need to update your pg_hba.conf file to use the "trust" method for your connections

- if so, you'll also need to reload the configuration/restart Postgres for the new settings to take

1.4.1 Turning on autovacuum for PostgreSQL

In your terminal, find the location of your Postgres config file named `postgresql.conf`
Here is an example of a Linux terminal command that would display the file location:

```
sudo -u postgres psql -c 'SHOW config_file'
```

Once you find your config file, edit it if needed to un-comment the line `autovacuum = on` if it begins with a #
You may also want to edit the remaining configuration options to set how your database handles `autovacuum`

1.4.2 Using the API to populate your empty database

Populating a Postgres database through the API includes the following data checks & structures:

- attempt to correct files missing headers
- removal of bad time records (bounded by station deployment dates)
- beep data
  - file the beep came from attached to the record
  - unique id for each beep record
  - filtering: removal of records without…
    - radio id
    - time stamp
- node health
  - re-coding battery > 9 to NA
  - filtering: removal of records without…
    - unique combination of radio id, node id, time and station id kept (i.e. removal of duplicates)
    - time stamp

- salvaging corrupt rows where possible

In development:

- tag ID validation

1.5 Data Cleaning

To remove duplicate records from your database (more than one record that has the same time stamp of a tag ID "beeping" on a node) the `db_cleanup()` function may be used. Additionally, if there are erroneous records of the same time stamp, tag ID and node ID with varying RSSI values, all records for that "beep" (combination of timestamp, tag ID and node ID) will be removed.

1.6 Incorporating Node Data Into Your Database

To include data pulled from your node SD card(s) create a folder called "nodes" and populate it with folders named for each node. Place the data pulled from each node in its respective folder. If you have already pulled sensor station data via the API, place the "nodes" folder in the folder that is auto-created by `get_my_data()` named for your project. In this case, the "outpath" and "myproject" arguments remain the same as for the functions above. In absence of a project name, "outpath" will be the folder containing the "nodes" folder. Run `import_node_data(conn, outpath, myproject="your project name")` to read in your node data and add it to the database.

Chapter 2

Tools to Work with Data

This is a manual for the R tools hosted at our GitHub repository.

How to use GitHub

Getting Set Up

1. Create an account.
2. Work through chapters 6-12 here if you need to install git, and connect it all with RStudio:
3. Choose your own adventure from here: do you want the working branch you created synced with the main repository, or do you want your main branch synced? Once you've decided, move onto the next step.

- having your working branch synced makes sure you can easily pull the latest files into your work space, but to work around that you should make your own copies of files you alter to make sure files don't conflict when you pull updates

- having your main branch synced is a bit more of a conventional structure, and means that changes pulled won't automatically propagate to your working branch. you could e.g. pull changes to the main branch, and use that as a reference to see what changes you want to pull into your working copy, and resolve conflicts before merging

4. Follow the instructions (at least through 5) here under "How to do this using RStudio and GitHub?"

- the repository to fork
- you don't need to enter the back ticks in the shell

- this example is a bit misleading because it doesn't include the .git, copy the link to the clipboard like before
- RESTART RSTUDIO BEFORE MOVING ONTO STEP 6 IN THIS TUTORIAL

5. If you want to pull updates from here to your copy, see chapter 31.

Result

By following these instructions, you should now…

- have a local copy of the repository
- be working on your own branch
- have an upstream connection to the main CTT repository
- "example.R" shows you example implementations of the data management and node health functions (also read comments, functions that produce files are commented out)
- "locate_example.R" is a template script for running the location functions

I suggest making your own copy of these scripts, renaming them, and modifying them with your file path inputs.

2.1 Pulling Data into R from your Local Database

This is the preferred option; please see postgres_example.R in your repository

Chapter 3

About the Functions

There is a sub-folder within this repo named "functions" which is full of, well, scripts that contain functions! You'll notice they're often called (via source()) at the top of the example scripts. This loads in the custom functions that I have written to handle CTT data. Ultimately, these will be rolled into an R package.

3.1 Data Manager

3.1.1 load_data

Description
Loads data

Usage

```
load_data(directory_name=NULL, starttime=NULL, endtime=NULL, tags=NULL)
```

Arguments

- directory_name: the input folder can contain a miscellany of raw downloaded files from the sensor station (beep data, node health, GPS) all in the same folder or subfolders. Zipped folders need to be unzipped, but compressed files do not (i.e. csv.gz files are just fine as they are).
- starttime: start time in POSIXct
- endtime: end time in POSIXct
- tags: a vector of tag IDs

Value

The function will return a nested list where each item corresponds to:
1. beep data
2. node health
3. GPS

Within each list item, there is a list for a data frame and the hardware version. Also, a column "v" has been added to each data frame indicating the hardware version.

3.2 Node Health

3.2.1 node_channel_plots

Description
This function is the "engine" behind the export function. You can run it standalone with the following parameters, but you don't have to if your sole goal is to output image files.

Usage

```
node_channel_plots(health, freq, ids, lat=NULL, lon=NULL)
```

Arguments

- health: the 2nd list item output by the load_data() function
- freq: the time interval for which you want variables to be summarized
- ids: a vector of IDs; the ID is of the format "_"
- lat: latitude (optional to produce day/night shading)
- lon: longitude (optional to produce day/night shading)

Value
The output is a nested list, where the top level is each combination of channel and node, and each item is a list of the following plots:

1. battery
2. RSSI
3. number of check-ins
4. scaled number of check-ins as line plot over scaled RSSI
5. box plot of node RSSI
   THE FOLLOWING ONLY FOR V2

6. latitude
7. longitude
8. scaled RSSI
9. dispersion

3.2.2 node_plots

Description
A set of diagnostic plots per node

Usage

```
node_plots(health, nodes, freq, lat = NULL, lon = NULL)
```

Arguments

- health: the 2nd data frame output by the load_data() function
- nodes: list of nodes
- freq: the time interval for which you want variables to be summarized
- lat: latitude
- lon: longitude

Value
The output is a nested list for each node, with the following plots for each:

1. RSSI
2. number of check-ins
3. battery
   THE FOLLOWING ONLY FOR V2
4. time mismatches
5. small time mismatches

3.2.3 gps_plots

Description
Plots to visualize some GPS data. ONLY FOR V2 HARDWARE

Usage

```
gps_plots(gps, freq)
```

Arguments

- gps: the 3rd data frame from the load_data() function
- freq: the time interval of summary

Value
A list of the following plots:

1. altitude
2. number of fixes

3.2.4 export_node_channel_plots

Description
Export plots of node x channel data

Usage

```
export_node_channel_plots(plotlist=NULL,health,freq="1 hour",out_path=getwd(),whichplot
```

Arguments

- plotlist: allows you to pass the output of node_channel_plots() if you prefer
- health: the 2nd data frame output by the load_data() function
- freq: the time interval for which you want variables to be summarized
- out_path: where you want your plots to go
- whichplots: an index vector of of the available plots

Output
This outputs a png for each input combination of node and channel.

3.2.5 export_node_plots

Description
Same as above; index for the plots can be chosen from the list under the `node_plots()` description
Usage

```
export_node_plots(plotlist = NULL, health,freq,out_path=getwd(), x=2, y=3, z=1)
```

Arguments
To assign x, y and z, look at the description for `node_channel_plots()` and select those plot index in the order you want them on the page.

Output
This outputs a png for each input node

Chapter 4

Calibration

4.1 Description

In order to best use the triangulation approach, calibration needs to be performed on your equipment. The goal of the calibration is to come up with a tag-specific relationship between RSSI and distance that is appropriate for your study site. An example of how to calibrate your system can be found in the supplementary material of Bircher et al. (2020). Notice this calibration experiment demonstrates calibration of tags and receivers (in our case, nodes).

4.2 Tag Calibration

- transects to cover differently vegetated areas of the study site
- varying tag height at each distance along the transect
- simulations of movement and different orientations

4.3 Node Calibration

Tags at varying orientations a fixed distance from each node.

4.4 How to Use the Output

There are a few routes to go from here. To extend the application by Bircher et al. (2020), the latter could be used to adjust RSSI values per node. Alternatively, one could derive a simple RSSI ~ distance relationship for each tag using the data above to measure N and input custom relationships (see 5.2.2).

Chapter 5

Localization Methods

5.1 Primitive: Weighted Average

Description
This is simply a weighted average based on number of beeps on a node and max. RSSI values.

Usage

```
weighted_average(freq, beeps, node, node_health=NULL, MAX_NODES=0, tag_id=NULL, calibrate = NULL,
```

Arguments

- freq: this is the interval a localization should be summarized over, and is in the form of an interval (e.g. `"3 min"`)
- beep_data: raw beep data frame (e.g. `all_data[[1]][[1]]` from example.R)
- node: read in node file
- node_health: node health data frame
- MAX_NODES: the max number of nodes that should contribute to a localization. default = 0 means all nodes
- tag_id: a vector of tags to calculate locations for
- calibrate: the session ID if you want to calculate over the entire duration a tag was left at a point
- keep_cols: if there are valuable columns that shouldn't be dropped
- latlng: BUGGY DO NOT USE YET
- minRSSI: the minimum RSSI of data used for the location estimate

Value

A SpatialPointsDataFrame of estimated locations

5.2 Triangulation Functions

5.2.1 Calibration

Usage
NOTE: This function returns a single relationship for the entire input! The example code to date also demonstrates the output of 1 distance ~ RSSI relationship. In order to calibrate per tag, it will be necessary to modify your code to loop over (e.g. apply) each tag ID, sub-setting your input data accordingly.

You can also use this function if you e.g. left tags at a known location in your grid for a period of time. This function preps the beep data frame for input into the triangulation function, and also implements a calibration by fitting an asymptotic function for RSSI and distance.

The calibration data frame needs the following column names:

- pt: this can be any identifier for a given location used in the calibration
- session_id: a character row identifier
- start: the beginning of the time interval when the tag was placed at the point, in POSIXct UTC
- end: the end of the time interval when the tag was placed at the point, in POSIXct UTC
- TagId: the tag ID left at the point
- TagLat: latitude of the point
- TagLng: longitude of the point

```
calibrate(beep_data, calibration, nodes, calibrate = TRUE, freq = "3 min", max_nodes =
```

Arguments
The option `calibrate = TRUE` is the default, and means that summary stats will be calculated over the entire time interval for each calibration location. Otherwise, pass `calibrate = FALSE, freq = <interval>` for the time interval of interest.

- beep_data: beep data frame
- calibration: data frame described above
- nodes: node file

- calibrate: whether or not the entire time interval a tag was at a point should be used for the estimation
- freq: alternatively, specify an interval for location estimation
- max_nodes: how many nodes should contribute? default = 0 means all nodes

Value
This function returns a list, the items of which are...

1. data frame to be input into the triangulation
2. a (see below)
3. S (see below)
4. K (see below)

5.2.2 Custom Distance Function

Description
You can pass a custom distance function to the triangulation, in the form of a string, that represents the relationship between RSSI and distance for your system. The string that you pass is the right side of the formula, where the left side is distance. The string needs to contain `x` which refers to RSSI. An example of an asymptotic relationship can be generated by the following function and the output of the calibrate function:

Usage

```
relate(a, S, K)
```

Arguments
These are fitted coefficients from an `SSasymp()` model relating distance to RSSI

- a = R0 e.g. the 2nd item returned from the calibrate function
- S = exp(lrc) e.g. the 3rd item returned from the calibrate function
- K = Aysm e.g. the 4th item returned from the calibrate function

Value
Inspect that string if you would like to instead create your own (e.g. for tag-wise calibration)

5.2.3 Data Prep

Description
This function prepares beep data to be input into the triangulation.

Usage

```
loc_prep(beep_data, nodes, freq)
```

Arguments

- beep_data: beep data frame
- nodes: node file
- freq: interval to calculate locations over

Value
A data frame that can be used as input to the `triangulate()` function

5.2.4 Triangulation

Description
This performs the triangulation with an input data frame and defined distance relationship. You could e.g. fit a relationship between distance and RSSI based on your calibration work.

Usage

```
triangulate(all_data, rssi = -100, node = 3, distance = relation)
```

Arguments

- all_data: a formatted data frame, such as the output from `loc_prep()` or the data frame returned by `calibrate()`
- rssi: the minimum RSSI threshold to incorporate data into the location calculation
- node: the maximum number of nodes to contribute to the calculation
- distance: a string representing the right side of a formula relating RSSI to distance, where distance is the left side and RSSI is `x` in the string

Value
A data frame with estimated locations and error

Bibliography

Bircher, N., van Oers, K., Hinde, C. A., and Naguib, M. (2020). Extraterritorial forays by great tits are associated with dawn song in unexpected ways. *Behavioral Ecology*, 31(4):873–883.