

Cellular Tracking Technologies: Data Tools

Jessica Gorzo

2021-05-19

Contents

Instructions	5
How to use GitHub	5
1 API	7
1.1 Install Postgres	7
1.2 Your Token	7
1.3 R script: api_run.R	7
1.4 Terminal	8
2 Start Here: Example Scripts	9
3 About the Functions	11
3.1 Data Manager	11
3.2 Node Health	12
4 Localization Methods	15
4.1 Primitive: Weighted Average	15
4.2 Triangulation Functions	15

Instructions



This is a manual for the R tools hosted at our GitHub repository. A RStudio tutorial is beyond the scope of this readme, but there are great resources to get you started with installing R and RStudio.

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?”
 - 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

Chapter 1

API

1.1 Install Postgres

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.

1. For simplicity, set your Postgres user name to be the same as your computer user name. Otherwise, you will need to pass it as an argument to the connection
2. Create a database in Postgres owned by that user name. You may have to set a password, and you may have to pass that password as an argument to the connection

1.2 Your Token

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

1.3 R script: `api_run.R`

1. As with the other R tools, I would suggest creating your own copy of “`api_run.R`” within your local repository, and modify that file.
2. Assign the API token you found above to the “`my_token`” variable (line 6)
3. 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
4. If you do not want to create a database...
 - comment out lines 8, 13-14
 - remove the “conn” argument from the `get_my_data()` function (line 11)
 5. If you do want to create a database locally, set “db_name” to the name of the Postgres database you created (line 7)

1.4 Terminal

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

Chapter 2

Start Here: Example Scripts

- “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.

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`: POSIXct
- `endtime`: 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 the raw files 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.

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: the ID is of the format “_”
- lat: latitude
- lon: longitude

Output 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
6. latitude

7. longitude
8. scaled RSSI
9. dispersion

3.2.2 `node_plots`

Description

NOTE: THIS ONLY WORKS FOR V2

Usage

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

Arguments

- `health`: the 2nd dataframe 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

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

1. RSSI
2. number of check-ins
3. battery
4. time mismatches
5. small time mismatches

3.2.3 `gps_plots`

Description

Plots to visualize some GPS data

Usage

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

Arguments

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

Output

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(),whichplots
= c(3,2,1))
```

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

3.2.5 export_node_plots**Description**

NOTE: THIS ONLY WORKS FOR V2

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 indices in the order you want them on the page.

Chapter 4

Localization Methods

4.1 Primitive: Weighted Average

This is simply a weighted average based on number of beeps on a node and max. RSSI values. It takes the following mandatory 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

```
locations <- weighted_average(freq = “3 min”, beep_data, node = nodes,  
node_health, 0, tags)
```

4.2 Triangulation Functions

4.2.1 Data Prep

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

```
input <- loc_prep(beep_data, nodes, freq)
```

4.2.2 Calibration

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`: this can be any identifier for the time period the tag was at this spot
- `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

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.

```
calibrated <- calibrate(beep_data, calibration, nodes, calibrate = TRUE)
```

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)

4.2.3 Custom Distance Function

You can pass a custom distance function, in the form of a string, that represents the relationship between RSSI and distance for your system. The string 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:

```
relation <- relate(calibrated[[2]], calibrated[[3]], calibrated[[4]])
```

4.2.4 Triangulation

This performs the triangulation with an input dataframe and defined distance relationship.

```
out <- triangulate(calibrated[[1]], distance = relation)
```