‘celltracktech’ Data Analysis package

Jessica Gorzo

2025-05-06

Contents

Chapter 1

AOS2024 - 'celltracktech' Workshop

This RBook goes over the files, functions, and analyses from the 2024 American Ornithological Society (AOS) Workshop in Estes Park, CO on the 'celltracktech' R package. This package was developed by Dr. Jessica Gorzo, Dr. Sean Burcher, and Dr. Kristin Paxton.

This document will serve as a tutorial on how Cellular Tracking Technologies (CTT) registered users can download their data from our server, analyze the data using multilateration, and visualize the data using the built-in package functions.

This tutorial provides step-by-step instructions on how to obtain your data. This style is used to increase accessibility for absolute beginners in R, SQL, and data science.

1.1 Additional Libraries Needed (Linux users)

Note If you are using Linux (specifically Ubuntu), you may need to install the following libraries in the teriminal.

1.1.1 Install PostgreSQL libraries

```
sudo apt install libpq-dev libssl-dev
```

1.1.2 Installing R Spatial on Ubuntu

```
sudo add-apt-repository ppa:ubuntugis/ubuntugis-unstable
sudo apt update
sudo apt install libgdal-dev libgeos-dev libproj-dev libtbb-dev
```

Chapter 2

Project Setup

Here we will go over how to create a local project directory (or folder as it is commonly known) on your own personal computer.

2.1 Create an R Studio Project

1. Download the latest versions of R and RStudio. As of writing this tutorial, the latest version of R is 4.4.2, and RStudio is '2024-12.0.467'.
2. In RStudio, create a new project (.proj) in a new directory. By using an R project, you can create a new directory to store all of your code scripts and data files, as well as keep track of your workflow. You can find more information here.
    a. Click the 'Create git repository' and 'use renv' check boxes.

2.1.1 Git

Git is a version control system, which allows you keep track of changes to your project, and collaborate with colleagues if you use Github.

2.1.2 `renv`

Renv stands for 'R Virtual Environment', which creates reproducible environments for your R projects. Many new data scientists would download all their individual packages into one location, usually the R library. While this global (i.e. accessible anywhere on your computer) library is convenient, certain packages or package versions can conflict with each other, causing errors and preventing you from analyzing your data. With renv, you can download all the packages for your project in your project directory, which will lead to fewer problems due to conflicting packages/versions.

```
install.packages('renv')
library(renv)

renv::activate()
```

After creating a .R file and loading a library, use `renv::snapshot()` to update your `renv.lock` file:

```
renv::snapshot()
```

If you are working with collaborators, you'll then need to commit `renv.lock`, `.Rprofile`, `renv/settings.json` and `renv/activate.R` so they can work with the same packages and package versions as you.

2.2 Request an API token

To access the CTT Application Programming Interface (API) (i.e. the program to download your data from the CTT servers), you will need an API token. You can request one here.

2.3 Create a '.env' file

For NodeJs and Python projects, many data scientists use environment variables, or a user-defined variable, to store sensitive data such as passwords, API credentials, and other info that should not be publicly shared. These environment variables can then be accessed by the R project without displaying the sensitive information on your R script.

To store the environment variables, use a '.env' file and the R package 'dotenv'.

Run the following command in an R script or the R console to create a .env file in your project directory and store your CTT API key:

```
system('touch .env; echo "API_KEY=your_api_key" >> .env')
```

Replace "your_api_key" with your actual API key from your `account.celltracktech.com` profile.

2.3.1 Load the API key into your RStudio environment

```
# load the env file
load_dot_env(file='.env')

# get your api key from the env file
my_token <- Sys.getenv('API_KEY')
```

2.3.2 Add .env file to .gitignore

Finally, you will need to add the .env file to the .gitignore file, so that when you commit changes to your R project the .env file is not included. That way your environmental variables will stay on your computer until you share them with collaborators.

Open the .gitignore file in RStudio and add '.env' (without the quotes) to the file and save it.

```
*_cache/
/cache/

# Temporary files created by R markdown
*.utf8.md
*.knit.md

# R Environment Variables
.Renviron

# pkgdown site
docs/

# translation temp files
po/*~

# RStudio Connect folder
rsconnect/

### R.Bookdown Stack ###
# R package: bookdown caching files
/*_files/

# Custom files and directories
.env


# End of https://www.toptal.com/developers/gitignore/api/r

```

You are now ready to start downloading data!

2.4 Organize your Project Directory

In your project directory create the following folders to organize your files:

- src - where you store your `.R` scripts
- data - store downloaded data from CTT and modified dataframes
- results - store plots made with `ggplot2`

Chapter 3

Download Data

You can download the following file types:

- raw: 434 MHz tag detections
- blu: 2.4 GHz tag detections
- gps: Sensor Station lat/lon with timestamps
- node_health: data on node battery, temperature, tag detections, etc.
- telemetry: ???
- sensorgnome: 166 MHz tag detections (Motus will need to translate the data into something usable)
- log: Sensor Station log files

Trying to load all the data files into RStudio's memory will lead to issues, namely maxing out on memory usage. Instead, you should create an SQL relational database. Using a database uses much less memory when combining or cleaning data frames compared to loading data directly into RStudio, which leaves more memory for analyses.

If you are new to relational databases, you should use DuckDB, a simple database management systems (DMBS) that is easy to install and use in RStudio.

```
# activate renv environment
renv::activate()
```

The `celltracktech` package includes all the packages you need. Download it from github using `renv`.

!NOTE The DuckDB install will take ~ 30 min. Please allocate time accordingly.

```
# install the celltracktech package using renv
library(renv)
```

```
renv::install('cellular-tracking-technologies/celltracktech')
```

If you want to download just your data files (in .csv.gz format), you can run the following script:

```
# load the celltracktech library
library(celltracktech)

# load env file into environment
load_dot_env(file='.env')

# Settings ------------------------------------------------------------------
my_token <- Sys.getenv('API_KEY') # load env variable into my_token
myproject <- "Meadows V2" # this is your project name on your CTT account, here we are

# Create your data directory if it does not exist
outpath <- "./data/" # where your downloaded files are to go

# Create project name folder
create_outpath(paste0(outpath, myproject, '/'))

# Download just the data files onto your computer
get_my_data(my_token = my_token,
            outpath = outpath,
            db_name = NULL,
            myproject = myproject,
            begin = as.Date("2023-08-01"),
            end = as.Date("2023-12-31"),
            filetypes=c("raw", "blu", "gps", "node_health", "sensorgnome", "telemetry"
)
```

3.1 Create a DuckDB database

Below is a sample script to create a DuckDB database

```
# Connect to Database using DuckDB ------------------------------------------
con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = "./data/Meadows V2/meadows.duckdb",
                      read_only = FALSE)
```

3.2 Download data from the CTT API

You are now connected to your DuckDB database, but so far nothing is in the database. The script below will download your data from the CTT servers. If

you do not want to use a database, you can still use the `get_my_data()` function to download the .csv.gz files.

NOTE! Your Sensor Station must be set to upload data to our servers. If your station does not upload data, skip this block and go to section 2.2.1.

```
get_my_data(my_token = my_token,
            outpath = outpath,
            db_name = con,
            myproject = myproject,
            begin = as.Date("2023-08-01"),
            end = as.Date("2023-12-31"),
            filetypes=c("raw", "blu", "gps", "node_health", "sensorgnome", "telemetry", 'log')
)
```

You may get this error message:

```
Error in post(endpoint = endpoint, payload = payload) :
  Gateway Timeout (HTTP 504).
```

It is fine, just run the `get_my_data()` function again and it should work properly.

3.2.1 Create Database from Files on your Computer

If you already have your Sensor Station files on your computer (i.e. your sensor station is not connected to the internet), you can use the code block below to create a database and add those files to it.

```
celltracktech::create_database(my_token = my_token,
                               outpath = outpath,
                               myproject = myproject,
                               db_name = con)
```

3.3 Updating the database

Upload the compressed data (i.e. the '.csv.gz' files) into your DuckDB database:

```
update_db(con, outpath, myproject)
```

3.4 Disconnecting from the database

The benefit of using a relational database is that it does not need to be loaded into your computer memory the entire time you are analyzing your data. You can connect to it, filter/clean the data you want, and then disconnect once you are done, freeing up computer memory for more intensive data analysis tasks.

To disconnect from the database, run the following command:

```
DBI::dbDisconnect(con)
```

3.5 Uploading Node Data from the SD Card

Note! This step is optional! If you are not uploading data from the Node SD cards, you can skip to chapter 3!

3.5.1 Create Nodes directory

To upload Node data from the SD cards, you will need to create a 'nodes' folder in the CTT Project Name folder. For example, we will create a folder in the 'Meadows V2' folder in the './data/meadows/' directory:

```
create_outpath('./data/Meadows V2/nodes/')
```

3.5.2 Create individual directories for each Node

Then, create a folder for each Node. In the example below, we are creating a folder for Node 3B8845:

```
create_outpath('./data/Meadows V2/nodes/3B8845/')
```

If you are only uploading data from the node SD cards frequently, you should create a parent folder for the date you removed the data from the SD card. The nodes save files incrementally (i.e. 434_mhz_beep_0, 434_mhz_beep1, etc.). If you remove the files, it will start back at 0 again. When you put the files with the same name in the same folder, one of them will be overwritten.

```
create_outpath('./data/Meadows V2/nodes/20250423/3B8845)')
```

3.5.3 Upload Node data to your DuckDB database

Remember, we need to re-connect to the DuckDB database to upload the Node data:

```
con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = "./data/Meadows V2/meadows.duckdb",
                      read_only = FALSE)

# Import node data into your database
import_node_data(d = con,
                 outpath = outpath,
                 myproject = myproject,
                 station_id = '6CA25D375881')

# disconnect from the database
DBI::dbDisconnect(con)
```

If you get `Error in files_loc[1, ] : incorrect number of dimensions` you did not create the `nodes` folder in the correct location.

The `import_node_data` will import the node csv files into their own 'node' tables

- node_raw
- node_blu
- node_gps
- node_health_from_node

and insert the csv files into the regular "raw", "blu", and "node_health" tables.

You have finished uploading data to your database!

3.6 Create Postgres Database

If you or your lab are familiar with PostgreSQL, you can connect to your database with this:

```
# connect to Postgres database
con <- DBI::dbConnect(
  RPostgres::Postgres(),
  dbname="meadows"
)
```

Creating and editing a Postgres database is vastly different from a DuckDB database. You can find more information on Postgres and R here.

You should be able to do all the above as long as you use `RPostgres::Postgres()` in your database connection.

Chapter 4

Obtaining Data from Database

4.1 Duckplyr

If you are new to database queries but familiar with `dplyr` and `tidyverse`, we recommend using `duckplyr`. You can find more information about `duckplyr` here but the main takeaway is that you can query your database using dplyr phrases and pipes, while also saving memory on loading data.

```
library(celltracktech)

con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = "./data/Meadows V2/meadows.duckdb",
                      read_only = FALSE)

# load raw data table and find unique tags
unique_tags = tbl(con, 'raw') |>
  group_by(tag_id) |>
  summarize(num_detect = n()) |>
  select(tag_id, num_detect) |>
  arrange(desc(num_detect)) |>
  collect()

DBI::dbDisconnect(con)
```

4.2 SQL Queries

If you are more comfortable with the SQL syntax, you can use SQL queries to get data from different tables.

This chapter is a quick summary on how to use SQL in R. If you would like more info on how to run different queries, use this tutorial: https://solutions.posit.co/connections/db/getting-started/database-queries/

4.3 List Tables

List the tables in your database. If everything was run correctly, each data file type (raw, blu, node-health, etc.) should be in its own data table.

Remember to reconnect to the database!

```
library(celltracktech)

con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = "./data/Meadows V2/meadows.duckdb",
                      read_only = FALSE)

# list tables in database
DBI::dbListTables(con)

# disconnect from database
DBI::dbDisconnect(con)
```

4.4 Find unique tags in `raw` table

```
# connect to database
con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = "./data/Meadows V2/meadows.duckdb",
                      read_only = FALSE)

# list last 10 records in raw
raw = DBI::dbGetQuery(con, "SELECT * FROM raw
                            ORDER BY time DESC
                            LIMIT 10")
raw

# find unique tags in the raw table
unique_tag = dbGetQuery(con,
                        'SELECT tag_id, COUNT(*) AS num_detect
                        FROM raw
```

```
                        GROUP BY tag_id
                        ORDER BY num_detect DESC')
DBI::dbDisconnect(con)
```

Note: you do not need to load each table into RStudio. You should only load the tables you need to for your analysis.

Chapter 5

Node Check

With your data you can run the following analyses:

- Presence/absence using detection times
- Activity budget using the change in signal strength over time
- Habitat Use using localization
- Home range/territory size
- Movement patterns

Before you can do any of that, it is a good idea to check if your nodes are working properly, and filter out any that are malfunctioning.

5.1 Node Health

Things to look at:

- Recent health records and detections
- Battery Level
- Are they charging?
- GPS fixes
- Synchronized clocks?

For example, debris on solar panels can lead to the node not to charge, which means the battery voltage will drop, which stops the GPS, leading to an out of sync clock.

Another example is that foliage cover leads to no charging, low battery voltage, stops the GPS, and leads to an out of sync clock.

5.1.1 Node Health - Check Health Records

5.1.1.1 Load Data from Database

```
library(celltracktech)

# load env file into environment
load_dot_env(file='.env')

# Settings - ----------------------------------------------------------------
# These were created in Chapter 2. If you do not have these in your project directory,
my_token <- Sys.getenv('API_KEY') # load env variable into my_token
myproject <- "Meadows V2" # this is your project name on your CTT account, here we are
outpath <- "./data/" # where your downloaded files are to go

# Specify the time range of node data you want to import for this analysis
start_time <- as.POSIXct("2023-08-01 00:00:00", tz = "GMT")
stop_time <- as.POSIXct("2023-08-07 00:00:00", tz = "GMT")

# Connect to Database using DuckDB ------------------------------------------------------
con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = "./data/Meadows V2/meadows.duckdb",
                      read_only = FALSE)

# load node_health table into RStudio and only load the data between the set start and
node_health_df <- tbl(con, "node_health") |>
  filter(time >= start_time & time <= stop_time) |>
  collect()

# disconnect from database
DBI::dbDisconnect(con)

# filter the node_health_df for unique node ids
node_health_df <- node_health_df |>
  distinct(node_id,
           time,
           recorded_at,
           .keep_all = TRUE)
```

5.1.1.2 Check if nodes are operating properly

```
# Look at the number of health records received from each node
node_record_counts <- node_health_df %>% count(node_id)

# sort the node_record_counts by decreasing number
```

```
node_record_counts <- node_record_counts[order(node_record_counts$n, decreasing = TRUE),]

# plot the number of node health records based on node id
ggplot(node_record_counts,
       aes(x = factor(x = node_id,
                      levels = node_id),
           y = n)) +
  geom_bar(stat = "identity") +
  coord_flip() +
  labs(x = "Health Record Count",
       y = "Node Id") +
  tag_hist_plot_theme()
```

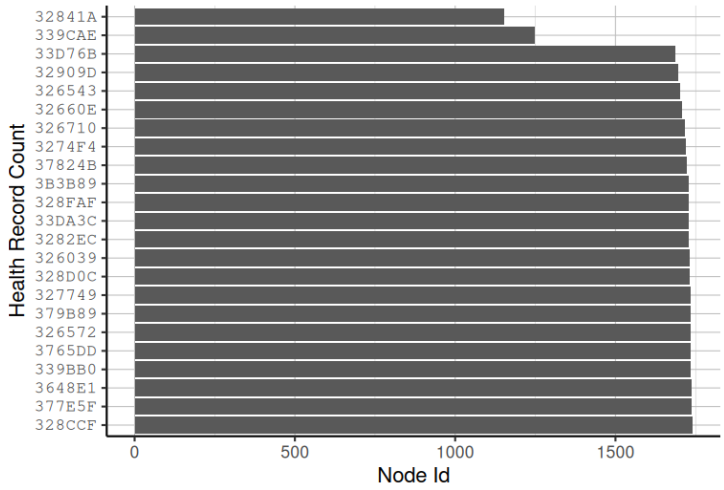


Figure 5.1: Plot of node health record counts

We can see after running the above script that Nodes 339CAE and 32841A have relatively low number of records, so we may want to exclude them from further analyses.

5.1.2 Battery and Solar Levels

Check if the battery and solar voltages are adequate. If they are too low, they can affect the GPS synchronization.

```
# Plot the Battery voltage vs. time for all nodes
ggplot(node_health_df) +
  geom_point(aes(x = time,
                 y = battery,
                 colour = node_id)) +
  classic_plot_theme()
```

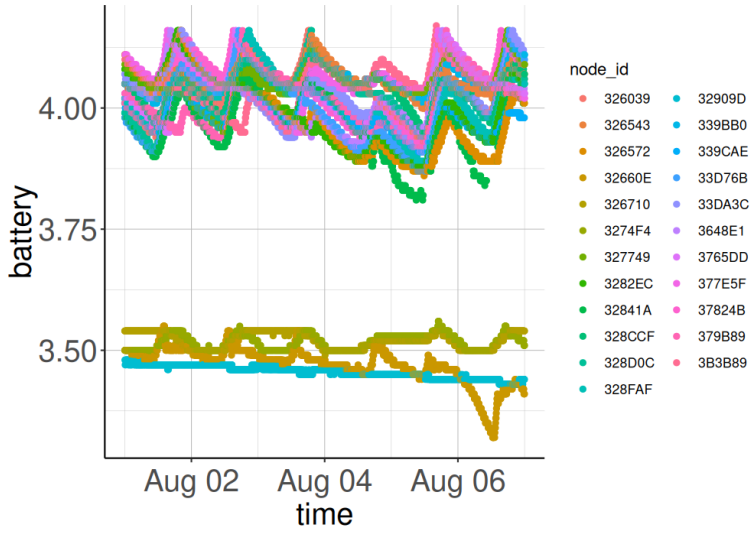


Figure 5.2: Plot of node battery voltage (V) vs. time

```
# Plot the Battery & Solar voltage vs. time for a specific node
# Node 326710 is a normal working Node
selected_node_id <- "326710"
batt_solar_plot <- plot_battery_solar(node_health_df = node_health_df,
                                      selected_node_id = selected_node_id)
batt_solar_plot
```

```
# based on the Battery voltage vs. time plot, Node 32909D has a low battery voltage
selected_node_id <- '32909D'
batt_solar_plot <- plot_battery_solar(node_health_df = node_health_df,
                                      selected_node_id = selected_node_id)
batt_solar_plot
```

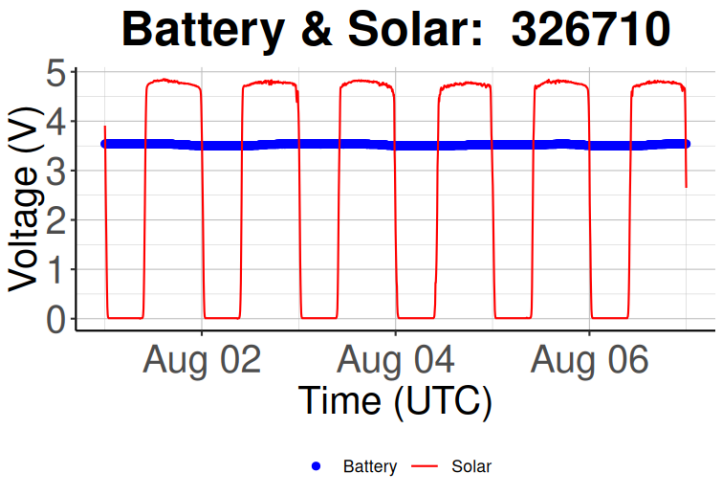
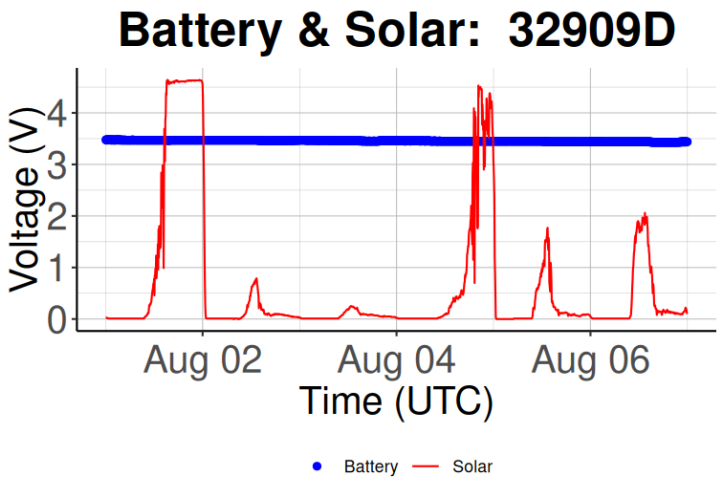


Figure 5.3: Plot of Battery and Solar voltage (V) vs. time for a specific node



Node 326710 is working properly and has consistent solar voltages during the day (~ 5 V) and a consistent battery voltage over time (i.e. 3.5-3.6 V), while node 32909D has inconsistent solar voltages, hinting at either dirty solar panels or covered solar panels, and a much lower battery voltage (~ 2.8 V).

5.1.3 Check GPS

If the clock is out of sync, the GPS coordinates may not be calculated correctly. Use the below script to check the coordinate deviations.

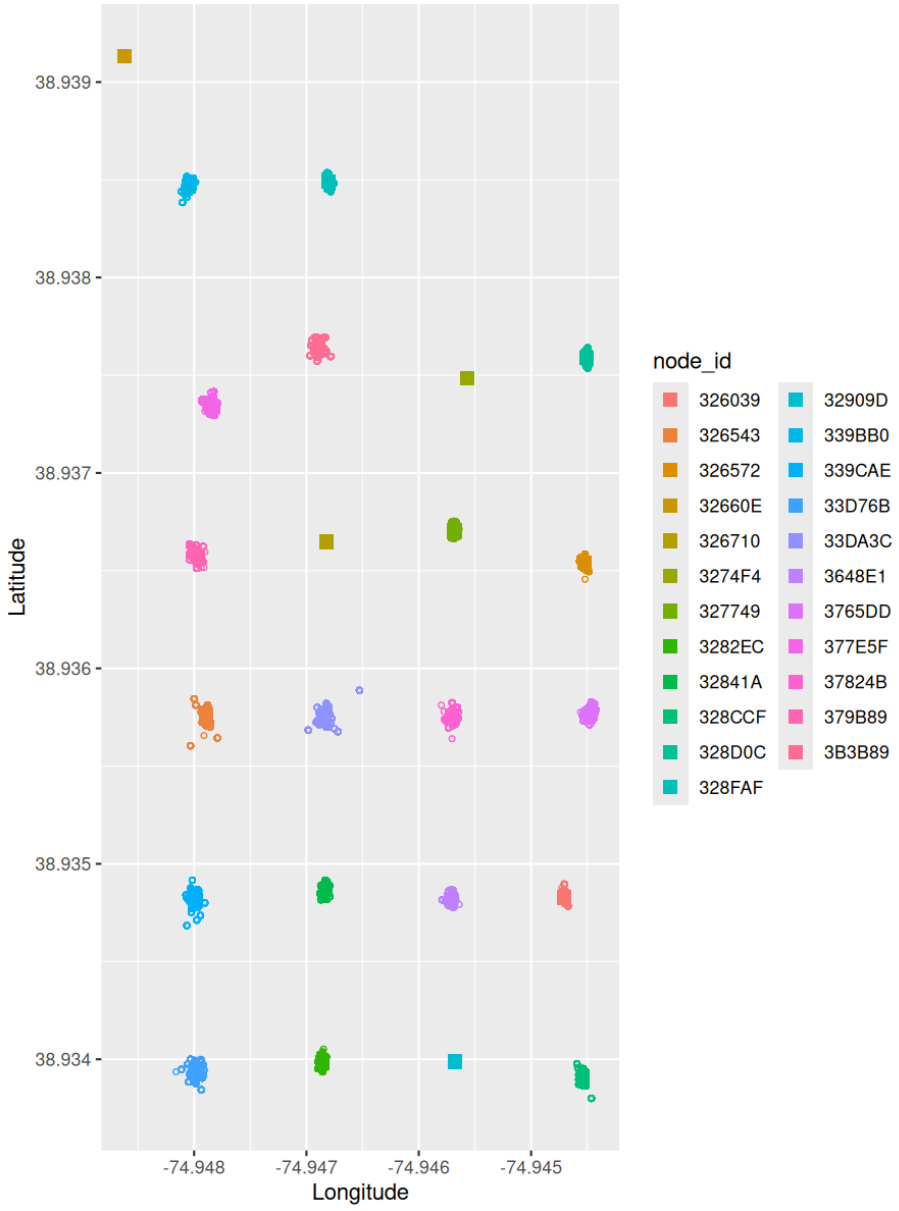
```
# calculate the node locations based on the latitude and longitude from node health re
node_locations <- calculate_node_locations(node_health_df)

# plot reported node locations and calculated node locations
plot_node_locations(node_health_df, node_locations)

# looks like we have an outlier in our nodes list, let's filter that out
# filter meadows nodes due to outlier
nodes_meadows = node_health_df %>%
  filter(!node_id %in% '3B3B8F') # get data from all nodes EXCEPT node 3B3B8F

# calculate the node locations based on the latitude and longitude from node health re
node_locations <- calculate_node_locations(nodes_meadows)
```

```
# plot reported node locations and calculated node locations
plot_node_locations(nodes_meadows, node_locations)
```



Check Time Offset

Plot time offset for each node.

```
# subtract time from recorded_at, then calculate the average time offset for each node
node_summary = node_health_df %>%
  mutate(time_offset = time-recorded_at) %>%
  group_by(node_id) %>%
  summarize(mean_time_offset = mean(time_offset), n = n())

# save node time offsets to csv
write_csv(node_summary,
                file = './data/Meadows V2/node_time_offset_20230802.csv')

# plot the time offset
ggplot(node_summary,
       aes(x = node_id,
           # y = scale(mean_time_offset),
           y = as.numeric(mean_time_offset),
           color=factor(node_id))) +
  geom_point() +
  ggtitle("Time Offset") +
  ylab("Time (s)") +
  geom_hline(yintercept=0,
             linetype="dotted",
             color = "red",
             linewidth=1) +
  labs(color = 'Node ID') +
  classic_plot_theme() +
  theme(axis.title.x = element_blank(),
        axis.text.x=element_blank(),
        axis.ticks.x=element_blank()
        )
```

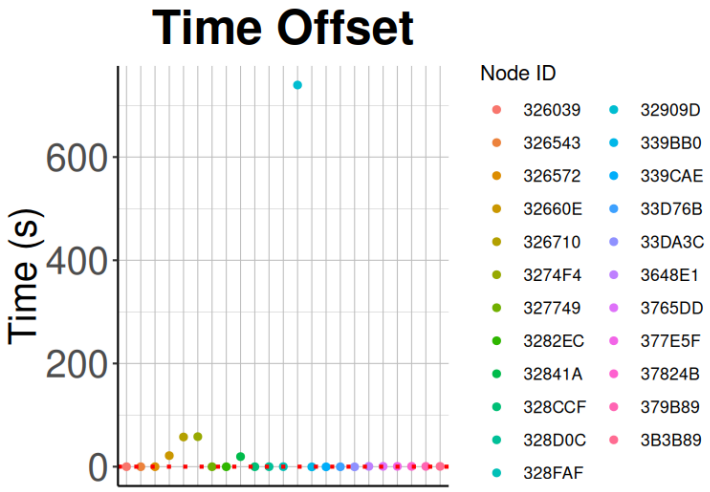


Figure 5.4: Node time offset

Chapter 6

Node Calibration

In the previous chapter we saw that various ways that the node gps can differ, which can lead to inaccurate received signal strength indicators (RSSI) from the tag.

To account for any small changes in the gps values, we need to calibrate the node grid. ## Load library

```
library(celltracktech)
```

6.1 Load sidekick data file

Place the .csv file generated by the CTT Sidekick into the sidekick folder:

```
# Specify the path to the sidekick data file you recorded for calibration
# it is best if you create a 'sidekick' folder to store your calibration file(s).

create_outpath('./data/Meadows V2/sidekick/')
```

The `celltracktech` package comes with the sidekick calibration file from the Meadows V2 site. For the purposes of this tutorial, we will save it as a .csv file and place it in the sidekick folder. If you were using your data, you would need to place the sidekick calibration file manuall in the sidekick folder.

```
write_csv(celltracktech::sidekick_cal, './data/Meadows V2/sidekick/calibration.csv')

sidekick_file_path <- './data/Meadows V2/sidekick/calibration.csv'
```

6.1.1 IF YOU DO NOT HAVE A SIDEKICK DATA FILE:

Create your own sidekick file by walking with a tag through node grid, stopping at specific spots you choose, and note the gps coordinates and time. Then create a csv similar to the sidekick calibration file.

Example:

tag_type	tag_id	time_utc	rssi	lat	lon	heading	antenna_angle
radio434mhz	4C34074B	2023-08-03 19:42:44.721001Z		38.935561	74.948195	24.273	
radio434mhz	072A6633	2023-08-03 19:42:45.307456Z		38.935561	74.948195	23.548	
radio434mhz	19332A07	2023-08-03 19:42:48.366123Z		38.935569	74.948195	15.840	

6.2 Setup

6.2.1 Load environmental variables into RStudio

```
# load env file into environment
load_dot_env(file='.env')
```

6.2.2 Load settings

```
# Settings - ------------------------------------------------------------------

# set significant digits (number of digits after decimal)
options(digits = 10)

# These were created in Chapter 2. If you do not have these in your project directory,
my_token <- Sys.getenv('API_KEY') # load env variable into my_token
myproject <- "Meadows V2" # this is your project name on your CTT account, here we are
outpath <- "./data/" # where your downloaded files are to go

# Specify the path to your database file
database_file <- "./data/Meadows V2/meadows.duckdb"

# Specify the tag ID that you used in your calibration
my_tag_id <- "072A6633"
```

```
# my_tag_id <- "614B661E"

# Specify the time range of node data you want to import for this analysis
#   This range should cover a large time window where you nodes were in
#   a constant location.  All node health records in this time window
#   will be used to accurately determine the position of your nodes
start_time <- as.POSIXct("2023-08-01 00:00:00", tz = "GMT")
stop_time <- as.POSIXct("2023-08-07 00:00:00", tz = "GMT")

# Specify a list of node Ids if you only want to include a subset in calibration
# IF you want to use all nodes, ignore this line and SKIP the step below
# where the data frame is trimmed to only nodes in this list
my_nodes <- c("B25AC19E", "44F8E426", "FAB6E12", "1EE02113", "565AA5B9", "EE799439", "1E762CF3",

# You can specify an alternative map tile URL to use here
my_tile_url <- "https://mt2.google.com/vt/lyrs=y&x={x}&y={y}&z={z}"
```

6.2.3 Load Node Health Data from Files

```
# connect to the database
con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = database_file,
                      read_only = TRUE)

# load node_health table in to RStudio and subset it based on your start and stop times
node_health_df <- tbl(con, "node_health") |>
  filter(time >= start_time & time <= stop_time) |>
  collect()

# disconnect from the database
DBI::dbDisconnect(con)

# remove any duplicate records
node_health_df <- node_health_df %>%
  distinct(node_id,
           time,
           recorded_at,
           .keep_all = TRUE)
```

6.2.4 Get Node Locations

```
# Calculate the average node locations
node_locs <- calculate_node_locations(node_health_df)
```

```
# Plot the average node locations
node_loc_plot <- plot_node_locations(node_health_df,
                                     node_locs,
                                     theme = classic_plot_theme())
node_loc_plot
```

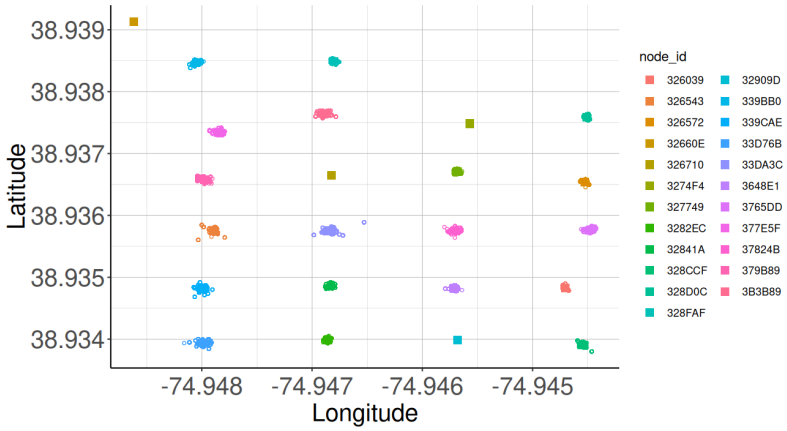


Figure 6.1: Node location plot

```
# Write the node locations to a file
create_outpath('results')

export_node_locations("results/node_locations.csv",
                      node_locs)

# Draw a map with the node locations
node_map <- map_node_locations(node_locs,
                               tile_url = my_tile_url)
node_map
```

6.2.5 Load Station Detection from Files

```
# connect to database
con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = database_file,
                      read_only = TRUE)
```

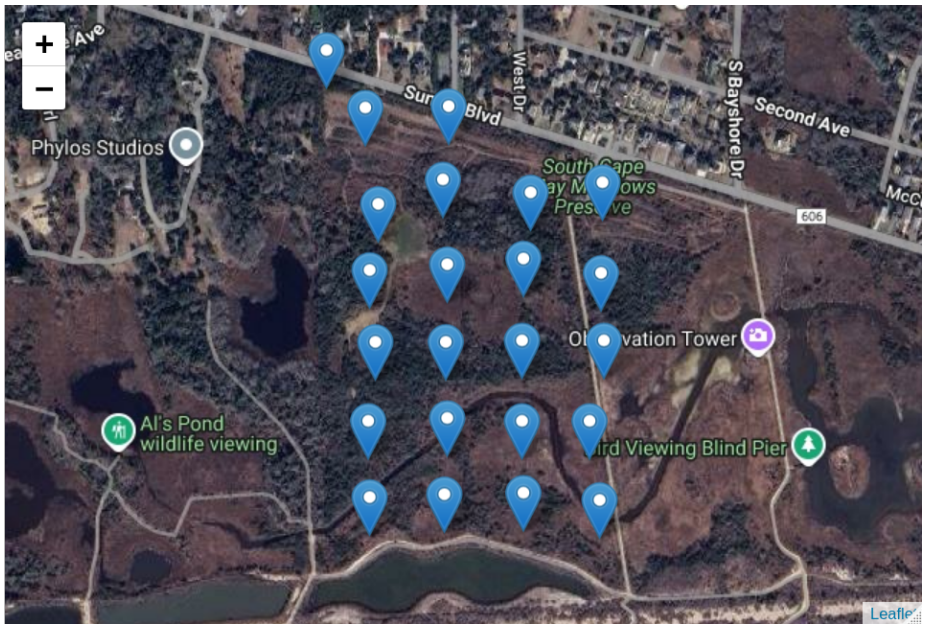


Figure 6.2: Node location map

```
# load raw data table and filter from start_time to stop_time
detection_df <- tbl(con, "raw") |>
  filter(time >= start_time && time <= stop_time) |>
  collect()

# if you are working with blu data, uncomment the lines below and load data from the blu table
# detection_blu <- tbl(con, "blu") |>
#   filter(time >= start_time && time <= stop_time) |>
#   collect

# disconnect from database
DBI::dbDisconnect(con)

# Get beeps from test tag only
detection_df <- subset.data.frame(detection_df,
                                  tag_id == my_tag_id)
```

6.2.6 Load Sidekick Calibration Data

```
# Get Sidekick data from CSV
sidekick_all_df <- load_sidekick_data(sidekick_file_path)
```

```
# Get beeps from test tag only
sidekick_tag_df <- subset.data.frame(sidekick_all_df,
                                     tag_id == my_tag_id)

# Show location of all beeps in relation to node locations
calibration_map <- map_calibration_track(node_locs,
                                         sidekick_tag_df,
                                         tile_url = my_tile_url)

calibration_map
```

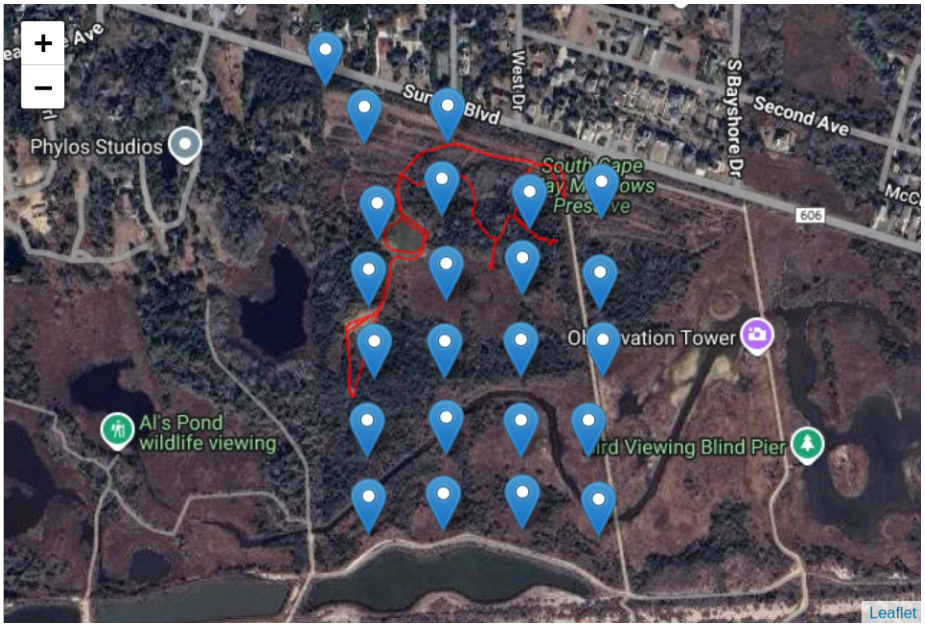


Figure 6.3: Node location plot

6.3 Calculate the RSSI vs. Distance Relationship

This function will match sidekick detections to detections recorded by nodes and sent to the station. Then using the sidekick location, the node locations calculated above, and the rssi measured in the node, a list of rssi and distance pairs is generated and returned.

For Blu Series tags use_sync=TRUE, for 434 MHz tags use_sync=FALSE.

```
rssi_v_dist <- calc_rssi_v_dist(node_locs = node_locs,
                                sidekick_tag_df = sidekick_tag_df,
                                detection_df = detection_df,
                                use_sync = FALSE)

# Plot the resulting RSSI and distance data
ggplot() +
  geom_point(data = rssi_v_dist,
             aes(x = distance,
                 y = rssi,
                 colour = node_id)) +
  labs(title="RSSI vs. Distance",
       x="Distance (m)",
       y="RSSI (dBm)",
       colour="Node ID") +
  classic_plot_theme()
```

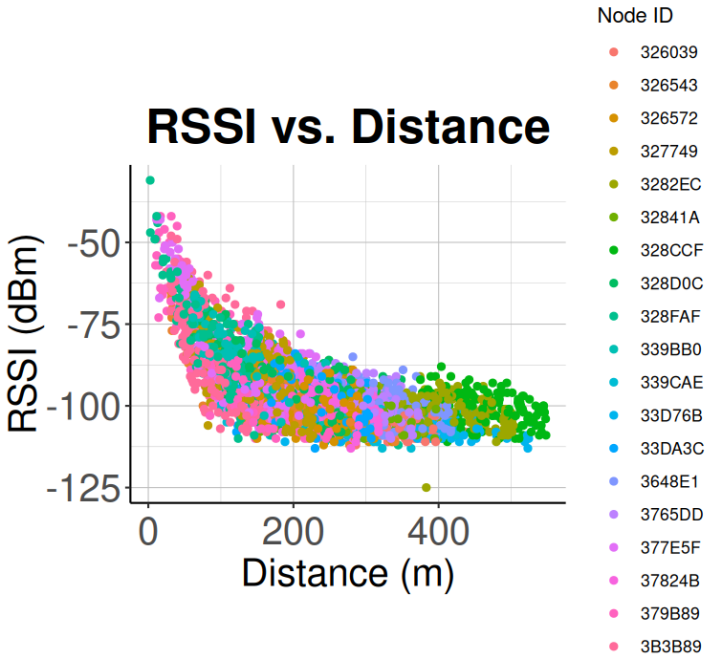


Figure 6.4: RSSI vs. Distance

```
# Fit the RSSI vs distance data with exponential relationship
nlsfit <- nls(
  rssi ~ a - b * exp(-c * distance),
```

```
  rssi_v_dist,
  start = list(a = -105, b = -60, c = 0.17)
)

summary(nlsfit)

# Get the coefficients from the fit result
co <- coef(summary(nlsfit))
rssi_coefs <- c(co[1, 1], co[2, 1], co[3, 1])

# Add a predicted column to the RSSI vs distance data
rssi_v_dist$pred <- predict(nlsfit)

# Plot the RSSI vs distance data with the fit curve
calibration_plot <- plot_calibration_result(rssi_v_dist, classic_plot_theme())
calibration_plot

# Print the coefficients from the fit. You'll need these coefficients later
# for localization.
print(rssi_coefs)
```

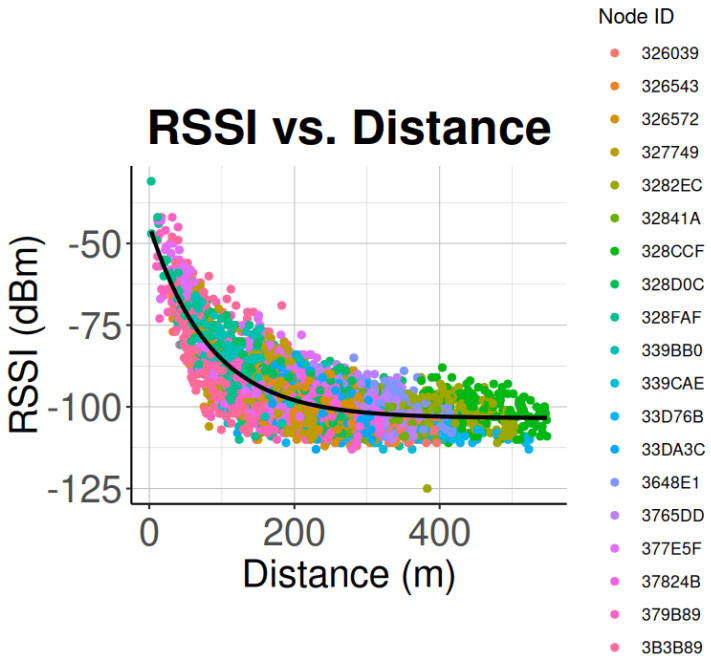


Figure 6.5: RSSI vs. Distance

Your node grid is now calibrated!

Chapter 7

Presence/Absence Analysis

This type of analysis can be used to answer questions on stopover duration or for threat monitoring at airports, windmills, etc.

You will need to manually create a tag deployment csv file. This file will contain: * the tag id (all uppercase) * the tag deployment date * the standardized 4-letter Alpha code (if your study involves birds) * the tag type (Power, Life, Blu) * the antenna type (1/8 wave) * any other characteristics of the individual wearing the tag (sex, weight, etc.).

Ex.

TagId	DeployDate	Species	TagType	AntennaType
332D074B	09/03/23	NOWA	Power	1/8 wave
664C5219	08/23/23	NOWA	Power	1/8 wave
4C2A0707	08/20/23	NOWA	Power	1/8 wave

Then move that tag deployment file into the 'outpath' folder.

The `celltracktech` package includes the deployments data for the Meadows V2 project. We will import it into the RStudio environment and save it as a .csv file in the `data/Meadows V2/` folder for this tutorial.

```
write_csv(celltracktech::deployments, './data/Meadows V2/meadows_deployments_2023.csv')
```

If you are using your own data, move your project-specific deployments .csv file into your `data/<project name>/` folder.

We are now ready to start calculating presence/absence.

7.1 Set parameters

```
options(digits = 10)

# Specify the path to your database file
database_file <- "./data/Meadows V2/meadows.duckdb"

# Specify the path to the deployment info file
deployment_info_file <- "data/Meadows V2/meadows_deployments_2023.csv"

# Specify the time range of node data you want to import for this analysis
start_time <- as.POSIXct("2023-08-01 00:00:00", tz = "GMT")
stop_time <- as.POSIXct("2023-12-31 00:00:00", tz = "GMT")
```

7.2 Load data from database

```
# Load from DB
con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = database_file,
                      read_only = TRUE)

# load the raw data table in to RStudio and subset it based on your start and stop tim
detection_df <- tbl(con, "raw") |>
  filter(time >= start_time & time <= stop_time) |>
  collect()

# if you study uses blu tags, load the blu data table into RStudio
detection_blu <- tbl(con, "raw") |>
  filter(time >= start_time & time <= stop_time) |>
  collect()

# disconnect from database
DBI::dbDisconnect(con)
```

7.3 Load Tag Deployment Info

7.3.1 Load Tag Deployment File

Again, you will need to create this file yourself. You might want to include other info about the individuals in this file for later analysis (species, weight, sex, etc.).

```
deployment_df <- read_csv(deployment_info_file)
```

7.3.2 Get List of Detected Tags

```
# count the number of detections for each tag. the minimum number of detections to be included in
tag_det_count <- get_tag_detection_count(detection_df, min_det_count = 250000)

# plot the number of detections for each individual tag
ggplot(tag_det_count,
       aes(x = factor(x = tag_id,
                      levels = tag_id),
           y = n)) +
  geom_bar(stat = "identity") +
  coord_flip() +
  labs(x = "Tag ID",
       y = "Detection Count") +
  tag_hist_plot_theme()
```

7.4 Generate Detection Summary

While it is good to know how many detections an individual tag has, it is better to know how many detections there were across time. The code chunk below will subset your `detection_df` based on the tags in your tag deployment file, and create a plot of detections across time.

```
# Discard detections that aren't from deployed tags
detection_df <- subset.data.frame(detection_df,
                                  tag_id %in% deployment_df$TagId) |>
                               filter(is.na(tag_id) == FALSE)

# OPTIONAL: You can select individual species here
# subsetting the deployment dataframe to only include NOWA or northern waterthrush
deployment_df <- subset.data.frame(deployment_df,
                                   Species == "NOWA")
# detection_df <- subset.data.frame(detection_df,TagId %in% deployment_df$TagId)

det_summary_df <- detection_summary(
  detection_df = detection_df,
  tag_list = deployment_df$TagId
)

# create detection summary data frame and sort it based on last detection date
det_summary_df <- det_summary_df[order(det_summary_df$last_det, decreasing = TRUE), ]

# create a heat-bin plot, with the bins reperesenting time periods, and the color representing th
ggplot(data = detection_df,
       aes(x = time,
```

```
          y = factor(tag_id,
                     det_summary_df$tag_id))) +
  geom_bin2d(binwidth = c(3600, 1)) + # Hour time bins
  scale_fill_continuous(type = "viridis") +
  labs(x = "Time (UTC)",
       y = "Tag Id") +
  tag_hist_plot_theme()
```

7.5 Show Detection History for a Specific Tag

```
# selected tag is a power tag on a swamp sparrow
selected_tag_id <- "1978341E"

# set your start and stop times
plot_start_time <- as.POSIXct("2023-10-09 10:00:00", tz = "GMT")
plot_stop_time <- as.POSIXct("2023-10-11 14:00:00", tz = "GMT")

# subset the detection_df to only include the selected tag id
tag_dets <- subset.data.frame(detection_df, tag_id == selected_tag_id)

# plot the number of detections for this specific tag at each node across time
ggplot(tag_dets) +
  geom_point(aes(x = time,
                 y = tag_rssi,
                 colour = node_id),
             shape = 1) +
  xlim(as.POSIXct(plot_start_time),
       as.POSIXct(plot_stop_time)) +
  ggtitle(paste("Detections",
                selected_tag_id)) +
  xlab("Time (UTC)") +
  ylab("RSSI (dBm)") +
  classic_plot_theme()
```

Chapter 8

Activity Budget

When is the animal active and/or inactive? Use the following script to answer research questions like:

- Nesting/Incubation behavior
- Roosting times
- Foraging times
- Mortality detection

8.1 Load settings

```
# activate renv
renv::activate()
```

```
# load celltracktech library
library(celltracktech)

# set significant digits
options(digits = 10)

# Specify the path to your database file
database_file <- "./data/Meadows V2/meadows.duckdb"

start_time <- as.POSIXct("2023-09-01 00:00:00",tz = "GMT")
stop_time <- as.POSIXct("2023-09-02 00:00:00",tz = "GMT")
```

8.2 Load Tag Node Detection Data

```
# Connect to database
con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = database_file,
                      read_only = TRUE)

# load raw data table into RStudio
detection_df <- tbl(con, "raw") |>
  filter(time >= start_time && time <= stop_time) |>
  collect()

# disconnect from the database
DBI::dbDisconnect(con)
```

8.3 Tag Activity

```
# selected_tag_id <- "2D4B782D", # SWSP (swamp sparrow) - Power Tag

# select a specific tag, below is a Power Tag on a Northern Waterthrush (NOWA)
selected_tag_id <- '614B661E'

# subset detection_df dataframe to only included the selected_tag_id
tag_dets <- subset.data.frame(detection_df, tag_id == selected_tag_id)

# sort the rows by time, ascending
tag_dets <- tag_dets[order(tag_dets$time, decreasing = FALSE), ]

tag_beep_interval <- 13 # seconds, will need to know of tag type beep intervals

# calculate tag activity - this turns the number of detections into a single value for
tag_activity <- calculate_tag_activity(tag_dets, tag_beep_interval)

# calculate average tag activity
avg_tag_act <- calc_avg_activity(tag_activity, start_time, stop_time)

# set start and stop times for the plots (next section)
plot_start_time <- as.POSIXct("2023-09-01 06:00:00", tz = "GMT")
plot_stop_time <- as.POSIXct("2023-09-02 06:00:00", tz = "GMT")
```

8.3.1 Scatter Plot of RSSI vs time by Node

```
ggplot(data = tag_dets) +
  geom_point(aes(x = time,
                 y = tag_rssi,
                 colour = node_id),
             shape=1) +
  xlim(plot_start_time, plot_stop_time) +
  ggtitle(paste("Detections", selected_tag_id)) +
  xlab("Time (UTC)") +
  ylab("RSSI (dBm)") +
  classic_plot_theme()
```

Here we see that this bird was spending most of the time in the vicinity of nodes 3282EC, 339CAE, and 37824B, but the strongest detections are near node 32841A.

8.3.2 Scatter Plot of activity vs time by Node

Next we will quantify the activity level. Activity is an arbitrary variable; it can be movement, roosting, a specific behavior, etc. You will need to define what activity is based on your study.

```
ggplot(data = tag_activity) +
  geom_point(aes(x = time,
                 y = abs_act,
                 colour = node_id),
             shape=1) +
  ggtitle(paste("Detections", selected_tag_id)) +
  xlab("Time (UTC)") +
  ylab("Activity (Arb. Units)") +
  xlim(plot_start_time, plot_stop_time) +
  classic_plot_theme()
```

Here we see the activity level (y-axis) for every 5 min from September 1 to September 2. Each color represents a unique Node ID. From a brief glance, this tag had low to mid activity levels near nodes 3282EC and 37824B, but it is still difficult to discern what is going on in this plot.

8.3.3 2D Histogram of Activity over Time

We can summarize the above plot even further by creating a heatmap. The code chunk below creates a heatmap bin plot, which displays the activity for an individual over a period of time, with the color of each bin representing the count of activity (on a log scale), while the bins tell us the activity level.

```
my_breaks <- c(1, 10, 100, 1000, 10000)

ggplot(data = tag_activity,
```

```
      aes(x = time,
          y = abs_act)) +
  geom_bin2d(binwidth = c(3600, 1)) +
  xlim(plot_start_time,
       plot_stop_time) +
  scale_fill_viridis_c(name = "Counts",
                       trans = "log",
                       breaks = my_breaks,
                       labels = my_breaks) +
  xlab("Time (UTC)") +
  ylab("Activity / Hour (Arb. Units)") +
  classic_plot_theme()
```

From this plot, we can see that there are multiple instances of low activity (demonstrated by the yellow bins at activity/hour 1), and fewer instances of high activity (demonstrated by the blue bins at activity/hour 4 and above).

8.3.4 2D histogram of activity vs time WITH avg activity

This code chunk makes the same plot as above, but also displays the average activity level as a red line.

```
ggplot(data = tag_activity,
       aes(x = time,
           y = abs_act)) +
 geom_bin2d(binwidth = c(3600, 1)) +
  geom_line(data = avg_tag_act,
            aes(x = time,
                y = avg_activity),
            colour = "Red") +
  geom_point(data = avg_tag_act,
             aes(x = time,
                 y = avg_activity),
             colour = "Red") +
  xlim(plot_start_time,
       plot_stop_time) +
  xlab("Time (UTC)") +
  ylab("Activity / Hour (Arb. Units)") +
  scale_fill_viridis_c(name = "Counts",
                       trans = "log",
                       breaks = my_breaks,
                       labels = my_breaks) +
  classic_plot_theme()
```

Here we see the average activity level for this tag is roughly 1 per hour from September 1 to September 2.

8.3.5 Avg activity / hour Vs time.

Let's zoom in and isolate the average activity per hour:

```
ggplot(data = tag_activity) +
  geom_line(data = avg_tag_act,
            aes(x = time,
                y = avg_activity),
            colour = "Red") +
  geom_point(data = avg_tag_act,
             aes(x = time,
                 y = avg_activity),
             colour = "Red") +
  xlim(plot_start_time, plot_stop_time) +
  xlab("Time (UTC)") +
  ylab("Activity / Hour (Arb. Units)") +
  scale_fill_viridis_c(name = "Counts",
                       trans = "log",
                       breaks = my_breaks,
                       labels = my_breaks) +
  classic_plot_theme()
```

We see that average activity is highest between 6am-12pm but decreases suddenly after that, which is what we would expect for bird activity level in September.

Chapter 9

Habitat Use

Calculate where an animal spends its time.

9.1 Load libraries and settings

9.1.1 Activate renv

```
renv::activate()
```

9.1.2 Load libraries and settings

```
library(celltracktech)
```

9.1.3 Load Functions

```
# Specify the path to your database file
database_file <- "./data/Meadows V2/meadows.duckdb"

# Specify the path to the deployment info file
deployment_info_file <- "./data/Meadows V2/meadows_deployments_2023.csv"
```

9.1.4 Load RSSI Coefficients

Remember Chapter 5 in which we calibrated the node grid? We will want to use those RSSI coefficients to accurately calculate the tag tracks from the detection data.

We will set the coefficients in the code block below, as well as our start and stop time for the Nodes, and for the tag detections.

```
# Specify the RSSI vs Distance fit coefficients from calibration
a <- -103.46373779280
b <- -59.03199894670
c <- 0.01188255653

# create list of rssi coefficients
rssi_coefs <- c(a, b, c)
```

9.1.5 Load Settings

```
options(digits = 10)

# Specify the time range of node data you want to import for this analysis
#   This range should cover a large time window where your nodes were in
#   a constant location.  All node health records in this time window
#   will be used to accurately determine the position of your nodes
node_start_time <- as.POSIXct("2023-08-01 00:00:00", tz = "GMT")
node_stop_time <- as.POSIXct("2023-08-07 00:00:00", tz = "GMT")

# Selected Tag Id - Hybrid tag on a Gray Catbird (GRCA)
selected_tag_id <- '2A33611E' # tag with most detections, 1/4 wave

# Analysis Time Range
det_start_time <- as.POSIXct("2023-10-01 12:00:00", tz = "GMT")
det_stop_time <- as.POSIXct("2023-10-06 12:00:00", tz = "GMT")

# You can specify an alternative map tile URL to use here
my_tile_url <- "https://mt2.google.com/vt/lyrs=y&x={x}&y={y}&z={z}"
```

9.2 Load Node Health Data from Files

```
# Load from DB
con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = database_file,
                      read_only = TRUE)

# load node_health data table into RStudio and filter it based on the start and stop t
node_health_df <- tbl(con, "node_health") |>
  filter(time >= node_start_time && time <= node_stop_time) |>
  collect()

# disconnect from the database
DBI::dbDisconnect(con)
```

```
# Remove duplicates
node_health_df <- node_health_df %>%
  distinct(node_id,
           time,
           recorded_at,
           .keep_all = TRUE)
```

9.3 Get Node Locations

Due to variations in GPS coordinates, it is a good idea to plot the Node locations and overlay the plot over a satellite image of your study site.

```
# Calculate the average node locations
node_locs <- calculate_node_locations(node_health_df)

# Plot the average node locations
node_loc_plot <- plot_node_locations(node_health_df,
                                     node_locs,
                                     theme = classic_plot_theme())
node_loc_plot

# Draw a map with the node locations
node_map <- map_node_locations(node_locs, tile_url = my_tile_url)
node_map
```

9.4 Load Station Detection Data

These are your tag detections in the 'raw' or 'blu' data tables in your database.

```
# Load from DB
con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = database_file,
                      read_only = TRUE)

# load raw data table into RStudio and filter it based on start and stop time
detection_df <- tbl(con, "raw") |>
  filter(time >= det_start_time && time <= det_stop_time) |>
  filter(tag_id == selected_tag_id) |>
  collect()

# disconnect from the databse
DBI::dbDisconnect(con)

# create time_value variable
```

```
detection_df <- detection_df %>%
  mutate(time_value = as.integer(time))
```

9.5 Build a Grid

To actually quantify the amount of habitat an animal uses, we will create a 500x800 m grid and overlay it over the map. Each grid bin will be 5x5m.

```
# set the grid coordinates
grid_center_lat <- 38.93664800
grid_center_lon <- -74.9462
grid_size_x <- 500 # meters
grid_size_y <- 800 # meters
grid_bin_size <- 5 # meters

# Create a data frame with the details about the grid
grid_df <- build_grid(
  node_locs = node_locs,
  center_lat = grid_center_lat,
  center_lon = grid_center_lon,
  x_size_meters = grid_size_x,
  y_size_meters = grid_size_y,
  bin_size = grid_bin_size
)

# Draw all of the grid bins on a map
grid_map <- draw_grid(node_locs, grid_df)
grid_map
```

9.6 Calculate Locations

This will display the tag tracks and the tag location on the node grid map.

```
# create a locations dataframe with the calculate_track() function
locations_df <- calculate_track(
  start_time = "2023-10-04 23:00:00",
  # start_time = "2023-08-01 00:00:00",
  length_seconds = 6 * 3600,
  step_size_seconds = 15,
  det_time_window = 30, # Must have detection within this window to be included in pos
  filter_alpha = 0.7,
  filter_time_range = 60, # Time range to include detections in filtered value
  grid_df = grid_df,
  detection_df = detection_df,
```

```
  node_locs = node_locs,
  rssi_coefs = rssi_coefs,
  track_frame_output_path = NULL # If NULL no individual frames will be saved
)
print(locations_df)

# overlay the tag tracks on the node grid and satelitte picture
track_map <- map_track(node_locs,
                       locations_df,
                       my_tile_url)
track_map

# calculate and display the location density on the node grid map
# source("R/functions/grid_search/grid_search_functions.R")
loc_density <- calc_location_density(grid_df = grid_df,
                                     locations_df = locations_df)

loc_density_map <- map_location_density(loc_density_df = loc_density, my_tile_url)
loc_density_map
```

If you zoom in on the map, you can see the areas where the animal spent most of its time.

For tags with 1/8 wavelengths, you may need to use a lower RSSI coefficient to accurately map tracks and habitat use. Try using -115 for RSSI coefficient 'a'.

Below is an example of a Power Tag with a 1/8 wave antenna, and will need a lower RSSI coefficient:

```
# Specify the RSSI vs Distance fit coefficients from calibration
a <- -115.0 # for 1/8 wave tags
b <- -59.03199894670
c <- 0.01188255653

rssi_coefs <- c(a, b, c)

# Selected Tag Id - Power Tag on a Swamp Sparrow (SWSP), 1/8 wave
selected_tag_id <- "2D4B782D"
```

Run the rest of the code blocks after setting these new RSSI coefficients to plot the habitat use of an individual with a 1/8 wave antenna.

Chapter 10

Grid Search Analysis

If you are interested in the extent of an animal's movement, use the following grid search analysis.

Like in Chapter 8 - Habitat Use, the grid search analysis divides an area into a grid, and calculates the received signal strength at each node. Then this process is repeated for each time step in a series of detections recorded by the node network.

10.1 Loading Settings

10.1.1 Activate renv

```
renv::activate()
```

10.1.2 Load libraries

```
library(celltracktech)
```

10.1.3 Load settings

```
# Specify the path to your database file
database_file <- "./data/Meadows V2/meadows.duckdb"

# (OPTIONAL) Specify Node time offsets if necessary
node_time_offset_file <- "./data/Meadows V2/node_time_offset_20230802.csv"
node_toff_df <- read.csv(node_time_offset_file)

# Specify the tag ID that you want to locate
```

```
my_tag_id <- "072A6633"

# Specify the RSSI vs Distance fit coefficients from calibration
a <- -103.0610446987
b <- -60.6023833206
c <- 0.0120558164
rssi_coefs <- c(a, b, c)

# Specify the time range of node data you want to import for this analysis
#   This range should cover a large time window where your nodes were in
#   a constant location.  All node health records in this time window
#   will be used to accurately determine the position of your nodes

# IMPORTANT! If you have included a node time offset file,
# make sure it was calculated using the same start and stop times as below.
node_start_time <- as.POSIXct("2023-08-01 00:00:00", tz = "GMT")
node_stop_time <- as.POSIXct("2023-08-07 00:00:00", tz = "GMT")

# Specify time range of detection data you want to pull from the DB
det_start_time <- as.POSIXct("2023-08-03 00:00:00", tz = "GMT")
det_stop_time <- as.POSIXct("2023-08-04 00:00:00", tz = "GMT")

# Specify a list of node Ids if you only want to include a subset in calibration
# IF you want to use all nodes, ignore this line and SKIP the step below
# where the data frame is trimmed to only nodes in this list
# my_nodes <- c("B25AC19E", "44F8E426", "FAB6E12", "1EE02113", "565AA5B9", "EE799439",

# You can specify an alternative map tile URL to use here
my_tile_url <- "https://mt2.google.com/vt/lyrs=y&x={x}&y={y}&z={z}"
```

10.2 Load Node Health data from Database

```
# Load from DB
con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = database_file,
                      read_only = TRUE)

node_health_df <- tbl(con, "node_health") |>
  filter(time >= node_start_time & time <= node_stop_time) |>
  collect()

DBI::dbDisconnect(con)
```

10.3 Get Node Locations

```
# Calculate the average node locations
node_locs <- calculate_node_locations(node_health_df)

# Plot the average node locations
node_loc_plot <- plot_node_locations(node_health_df,
                                     node_locs,
                                     theme = classic_plot_theme())
node_loc_plot

# Write the node locations to a file
export_node_locations("./results/node_locations.csv", node_locs)

# Draw a map with the node locations
node_map <- map_node_locations(node_locs, tile_url = my_tile_url)
node_map
```

10.4 Load Station Detection Data

```
# Load from DB
con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = database_file,
                      read_only = TRUE)

detection_df <- tbl(con, "raw") |>
  filter(time >= det_start_time & time <= det_stop_time) |>
  filter(tag_id == my_tag_id) |>
  collect()

DBI::dbDisconnect(con)

detection_df <- detection_df %>% mutate(time_value = as.integer(time))
```

10.5 Build a Node Grid

```
grid_center_lat <- 38.93664800
grid_center_lon <- -74.9462
grid_size_x <- 500 # meters
grid_size_y <- 800 # meters
grid_bin_size <- 5 # meters
# Create a data frame with the details about the grid
grid_df <- build_grid(
```

```
  node_locs = node_locs,
  center_lat = grid_center_lat,
  center_lon = grid_center_lon,
  x_size_meters = grid_size_x,
  y_size_meters = grid_size_y,
  bin_size = grid_bin_size
)
# Draw all of the grid bins on a map
grid_map <- draw_grid(node_locs, grid_df)
grid_map
```

10.6 (Optional) Calculate Test Solution

```
test_time <- as.POSIXct("2023-08-03 19:57:50", tz = "GMT")
test_rec_df <- calc_receiver_values(
  current_time = test_time,
  det_window = 60,
  station_tag_df = detection_df,
  node_locs = node_locs,
  node_t_offset = node_toff_df,
  rssi_coefs = rssi_coefs,
  filter_alpha = 0.7,
  filter_time_range = 120
)
print(test_rec_df)

# Find the GridSearch Solution
test_grid_values <- calc_grid_values(grid_df, test_rec_df, rssi_coefs)
solution <- subset(test_grid_values, test_grid_values$value == max(test_grid_values$val
print(solution)

# Multilateration calculation
reduced_rec_df <- subset.data.frame(test_rec_df, test_rec_df$filtered_rssi >= a)
node_w_max <- reduced_rec_df[reduced_rec_df$filtered_rssi == max(reduced_rec_df$filtere
multilat_fit <- nls(reduced_rec_df$exp_dist ~ haversine(reduced_rec_df$lat,reduced_rec_
                    reduced_rec_df,
                    start= list(ml_lat = node_w_max$lat, ml_lon = node_w_max$lon),
                    control=nls.control(warnOnly = T, minFactor=1/65536, maxiter = 1
                   )
print(multilat_fit)

co <- coef(summary(multilat_fit))

print(paste(co[1,1],co[2,1]))
```

```
multilat_result <- c(co[1,1],co[2,1])
test_map <- draw_single_solution(test_rec_df,
                                 test_grid_values,
                                 solution,
                                 multilat_result,
                                 my_tile_url)
test_map
```

10.7 Calculate Track

```
track_df <- calculate_track(
  start_time = "2023-08-03 19:50:45",
  length_seconds = 1050,
  step_size_seconds = 10,
  det_time_window = 60, # Must have detection within this window to be included in position calcu
  filter_alpha = 0.7,
  filter_time_range = 120, # Time range to include detections in filtered value
  grid_df = grid_df,
  detection_df = detection_df,
  node_locs = node_locs,
  node_t_offset = node_toff_df,
  rssi_coefs = rssi_coefs,
  track_frame_output_path = NULL # If NULL no individual frames will be saved
)
print(track_df)
track_map <- map_track(node_locs, track_df, my_tile_url)
track_map
```

10.8 (Optional) Compare with Known Track

```
# If you've recorded a test track with the sidekick and want to see how well you
# are able to recreate it you can use the commands below.

# load test track from celltracktech package; if you recorded a test track, you
# would need to load it from the .csv file
# sidekick_df <- read_csv('./path/to/csv_file')
sidekick_df <- celltracktech::sidekick_cal_test2

# Correct sidekick time formatting
sidekick_df <- sidekick_df %>% mutate(time_utc = substring(c(sidekick_df$time_utc), 1, 19))

# Add numerical time value column
```

```
sidekick_df <- sidekick_df %>% mutate(time_value = get_time_value(sidekick_df$time_utc)

# Trim Sidekick data to the time of the calculated track
sidekick_df <- subset.data.frame(sidekick_df, time_value <= max(track_df$time))

track_error_df <- calc_track_error(sidekick_df, track_df)

print(track_error_df)
print(min(track_error_df$error))
print(max(track_error_df$error))

print(paste("Grid Search Solution Error = ",
            mean(track_error_df$error),
            " +/- ",
            sd(track_error_df$error)))

print(paste("Multilateration Solution Error = ",
            mean(track_error_df$ml_error),
            " +/- ",
            sd(track_error_df$ml_error)))

compare_map <- map_track_error(node_locs,
                               track_error_df,
                               sidekick_df,
                               my_tile_url)
compare_map
```

10.9 Plot Uncertainty Analysis

```
# plot grid search analysis error and multilateration error across track point
ggplot() +
  geom_point(data = track_error_df,
             aes(x = i,
                 y = ml_error,
                 color = 'Multilateration Error')) +
  geom_point(data = track_error_df,
             aes(x = i,
                 y = error,
                 color = 'Grid Search Error')) +
  labs(color = 'Error Type') +
  scale_color_manual(values=c('Grid Search Error' = 'black',
                              'Multilateration Error' = 'orange'))+
  xlab("Track Point #") +
  ylab("Solution Error (m)") +
```

```
  classic_plot_theme()

# plotting grid search analysis error across max rssi (dBm)
ggplot() +
  geom_point(data = track_error_df,
             aes(x = max_rssi,
                 y = ml_error,
                 color = 'Multilateration Error')) +
  geom_point(data = track_error_df,
             aes(x = max_rssi,
                 y = error,
                 color = 'Grid Search Error')) +
  labs(color = 'Error Type') +
  scale_color_manual(values=c('Grid Search Error' = 'black',
                              'Multilateration Error' = 'orange'))+
  xlab("Max RSSI (dBm)") +
  ylab("Position Error (m)") +
  classic_plot_theme()
```

Chapter 11

Multilateration

This process is just like habitat use and grid search analysis, but instead of using the RSSI, it uses time difference of arrival instead to calculate the location of an animal.

You should use this method when your node grid is evenly spaced.

Multilateration paper

11.1 Load settings

11.1.1 Reset R's brain, remove all previous objects

```
rm(list=ls())
tagid = c("072A6633","2D4B782D") #"0C5F5CED"
timezone="UTC"
options(digits=9) # sets number after decimal
```

11.1.2 Load libraries

```
library(celltracktech)
```

11.1.3 Re-create sample data from node calibration (Chapter 5)

```
# create time window by reducing location precision or can input data with TestId column (user-de

# load node calibration file from Chapter 5
# mytest <- read.csv("./calibration_2023_8_3_all.csv")
```

```
mytest <- read.csv("./data/Meadows V2/sidekick/calibration_2023_8_3_all.csv")
mytest$Time <- as.POSIXct(mytest$time_utc, tz="UTC")

# connect to database
con <- DBI::dbConnect(duckdb::duckdb(),
                      dbdir = "./data/Meadows V2/meadows.duckdb",
                      read_only = TRUE)

# filter for specific dates and for the specified tags in tag id
testdata <- tbl(con, "raw") |>
  filter(time >= as.Date("2023-07-31") && time <= as.Date("2023-10-31")) |>
  filter(tag_id %in% tagid) |>
  collect()

# set node start and stop dates
start_buff = as.Date("2023-08-01", tz="UTC")
end_buff = as.Date("2023-08-07", tz="UTC")

nodehealth <- tbl(con, "node_health") |>
  filter(time >= start_buff  && time <= end_buff) |>
  collect()

# disconnect from database
DBI::dbDisconnect(con)

# create dataframe of nodes and their locations (lat, lon)
nodes <- node_file(nodehealth)
```

11.2 Isolate raw Received Signal Strength (RSS) data from Node network associated with Test Data

```
combined_data <- data.setup(mytest,
                            testdata,
                            nodes,
                            tag_col = "tag_id",
                            tagid = "072A6633",
                            time_col = "Time",
                            timezone = "UTC",
                            x = "lon",
                            y = "lat",
                            loc_precision = 6,
                            fileloc = "./data/Meadows V2/meadows.duckdb",
```

```
                                filetype = "raw")
```

11.3 Exponential Decay Function - Relationship between Distance and Tag RSS Values

```
# Plot of the relationship between RSS and distance
ggplot(data = combined_data,
       aes(x = distance,
           y = avgRSS,
           color = node_id)) +
  geom_point(size = 2)
```

As distance increases, we see average RSS decreasing exponentially.

11.3.1 Preliminary Exponential Decay Model - Determine starting values for the final model

- SSasvmp - self start for exponential model to find the data starting values
- Asvm - horizontal asymptote (when large values) - y values decay to this value
- R0 - numeric value when avgRSS (i.e., response variable) = 0
- lrc - natural logarithm of the rate constant (rate of decay)

```
# preliminary model - non-linear sampling
exp.mod <- nls(avgRSS ~ SSasymp(distance,
                                Asym,
                                R0,
                                lrc),
               data = combined_data)

  # Summary of Model
summary(exp.mod)
  # rate of decay
exp(coef(exp.mod)[["lrc"]])
```

11.3.2 Final Exponential Decay Model

User provides self-starting values based on visualization of the data and values in the Preiliminary Model Output

exponential model formula: avgRSS ~ a * exp(-S * distance) + K

- a = intercept
- S = decay factor
- K = horizontal asymptote

```
##  ***** Variables to define for final model below - replace values below with values
a <- coef(exp.mod)[["R0"]]
S <- exp(coef(exp.mod)[["lrc"]])
K <- coef(exp.mod)[["Asym"]]

  # Final Model
nls.mod <- nls(avgRSS ~ a * exp(-S * distance) + K,
               start = list(a = a,
                            S = S,
                            K= K),
               data = combined_data)

  # Model Summary
summary(nls.mod)

  # Model Coefficients
coef(nls.mod)

## Check the fit of the model and get predicted values
  # Get residuals and fit of model and add variables to main table
combined_data$E <- residuals(nls.mod)
combined_data$fit <- fitted(nls.mod)

  # Plot residuals by fit or distance
#ggplot(combined_data, aes(x = distance, y = E, color = node_id)) +
#        geom_point(size = 2)

#ggplot(combined_data, aes(x = fit, y = E, color = node_id)) +
#  geom_point(size = 2)

  # Get model predictions
combined_data$pred <- predict(nls.mod)

## Plot with predicted line
ggplot(combined_data, aes(x = distance,
                          y = avgRSS,
                          color=node_id)) +
  geom_point() +
  geom_line(aes(y = pred), color="black", lwd = 1.25) +
  scale_y_continuous(name = "RSS (dB)") +
  scale_x_continuous(name = "Distance (m)") +
  theme_classic()

a <- unname(coef(nls.mod)[1])
S <- unname(coef(nls.mod)[2])
```

```r
K <- unname(coef(nls.mod)[3])

combined_data <- estimate.distance(combined_data, K, a, S)

tile_url = "https://tile.openstreetmap.org/{z}/{x}/{y}.png"
testout <- combined_data[combined_data$TestId==0,]
leaflet() %>%
    addTiles(
      urlTemplate = tile_url,
      options = tileOptions(maxZoom = 25)
    ) %>%
    addCircleMarkers(
      data = nodes,
      lat = nodes$node_lat,
      lng = nodes$node_lng,
      radius = 5,
      color = "cyan",
      fillColor = "cyan",
      fillOpacity = 0.5,
      label = nodes$node_id
    )  %>%
    addCircles(
      data=testout,
      lat = testout$node_lat,
      lng = testout$node_lng,
      radius = testout$distance,
      color = "red",
      #fillColor = "red",
      fillOpacity = 0)

no.filters <- trilateration.TestData.NoFilter(combined_data)
RSS.FILTER <- c(-80, -85, -90, -95)
RSS.filters <- trilateration.TestData.RSS.Filter(combined_data, RSS.FILTER)
#DIST.FILTER <- c(315,500,750,1000)
# Calculate error of location estimates of each test location when Distance filters are applied p
#Dist.filters <- trilateration.TestData.Distance.Filter(combined_data, DIST.FILTER)

SLIDE.TIME <- 2
GROUP.TIME <- "1 min"

test_data <- testdata %>%
  filter(time >= as.Date("2023-10-05") & time <= as.Date("2023-10-15")) %>%
  filter(tag_id == "2D4B782D") %>%
  collect()
```

```
# Function to prepare beep data for trilateration
# by estimating distance of a signal based on RSS values
beep.grouped <- prep.data(test_data,
                          nodes,
                          SLIDE.TIME,
                          GROUP.TIME,
                          K,
                          a,
                          S)

RSS.filter <- -95
location.estimates <- trilateration(beep.grouped, nodes, RSS.FILTER)

# this will take a while...
mapping(nodes, location.estimates)
```