Motus R Book

Tara L. Crewe, Zoe Crysler, and Philip Taylor

Contents

A walk through the use of R for Motus automated radio-telemetry data

Our goal with this online 'handbook' is to show Motus (https://motus.org) users how to use the R statistical programming language (https://www.r-project.org/) to import tag detections data for a project or receiver; clean data and filter false positives; explore detections data through visualizations and summaries; transform the data, e.g., by determining time since sunrise/sunset or magnetic declination; and run various analytical procedures. We hope the contents will be of use, and if you have suggestions for additional examples, please let us know by emailing motus@birdscanada.org.

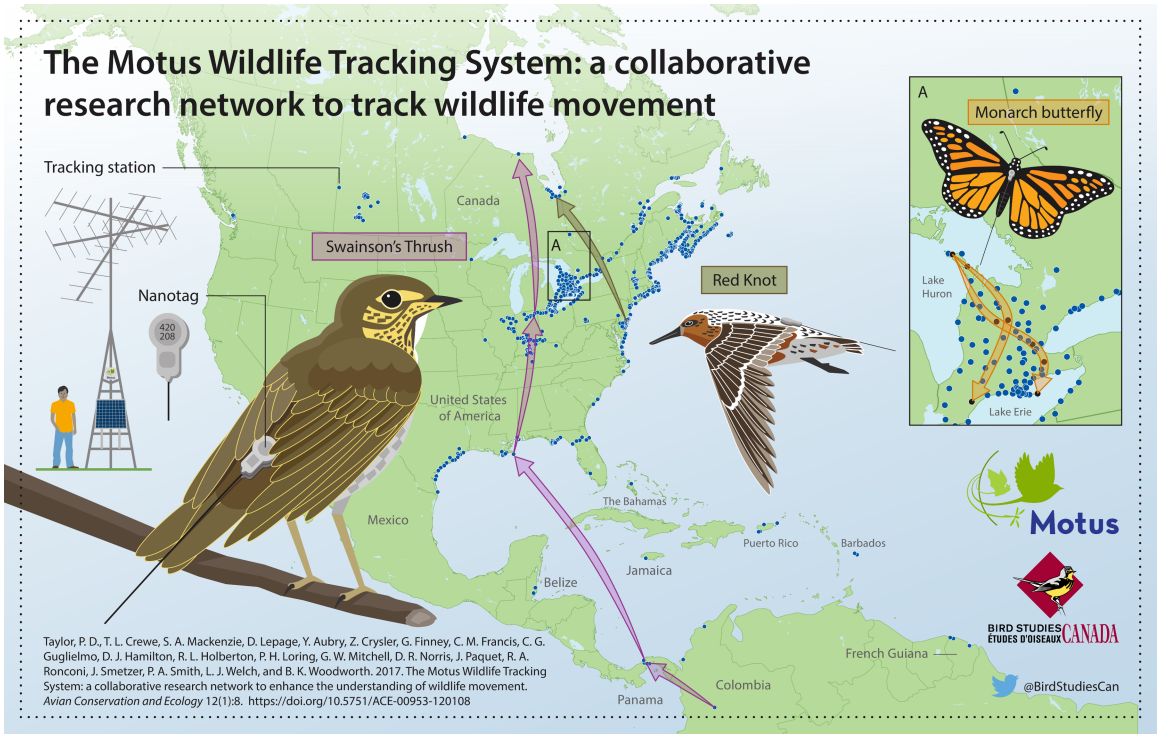
The current version is largely based on the earlier work by Crewe et al. 2018 (available at https://motus.org/motusRBook/archives/MotusRBook2018-01.pdf), and has since been supplemented by various people on the Motus team.

Published July 2020 using `motus` v3.1.0

Chapter 1

Introduction

This chapter was contributed by Tara L. Crewe, Zoe Crysler, and Philip Taylor



The Motus Wildlife Tracking System ('Motus'; Taylor et al. 2017; https://www.motus.org) is an international, collaborative automated radio-telemetry network to track the movement and behaviour of flying organisms affixed with

digitally encoded radio-transmitters. Motus was developed at Acadia University in 2012-2013. In 2014, a major infrastructure expansion was made possible through a Canada Foundation for Innovation grant to Western University, The University of Guelph, and Acadia University. Since then, Motus has grown through the collaboration of independent researchers and organizations (see https://motus.org/about/). It is now managed as a program of Bird Studies Canada (https://www.birdscanada.org) in partnership with Acadia University.

Motus is unique among automated telemetry arrays in that all researchers in a geographic region (e.g., the Americas or Europe) use a shared radio frequency. This allows tagged animals to be detected by any receiving station across the network, greatly broadening the spatial scope of potential research questions. Motus users also use a shared data infrastructure and web portal: all data collected from across the network are centrally stored and archived, which allows users to access detections of their tags by anyone's receiver in the network, and individuals that maintain receivers have access to all detections of anyone's tags on those receivers.

Having a shared data infrastructure also means that users can benefit from R functions written specifically for Motus data by any and all users. The `motus` R package described in this book is in continual development, and the intent of this online 'handbook' is to help users learn the various functionalities of the package, and potentially contribute to it. We also show how additional R packages such as ggplot can be used to explore, visualize, transform, and analyze Motus data.

The content of the handbook will continue to evolve and grow along with the analytical needs of the network. Those interested in contributing code to the Motus R package or this handbook can send proposed additions to motus@birdscanada.org.

Taylor, P. D., T. L. Crewe, S. A. Mackenzie, D. Lepage, Y. Aubry, Z. Crysler, G. Finney, C. M. Francis, C. G. Guglielmo, D. J. Hamilton, R. L. Holberton, P. H. Loring, G. W. Mitchell, D. R. Noriis, J. Paquet, R. A. Ronconi, J. Smetzer, P. A. Smith, L. J. Welch, and B. K. Woodworth. 2017. The Motus Wildlife Tracking System: a collaborative research network to enhance the understanding of wildlife movement. Avian Conservation and Ecology 12(1):8. https://doi.org/10.5751/ACE-00953-120108.

1.1 What this book does not cover

This book does not cover how to register radio tags with Motus, manage tags and station deployments, or upload raw detections data for processing. Information to guide you through those tasks can be found under the 'resources' tab on the Motus website at https://motus.org/resources/. Please remember to register your tags **prior to deployment**, and enter tag and station metadata online

in a timely manner. Please also review the Motus collaboration policy and tag registration and fee schedule at https://motus.org/policy/.

1.2 Prerequisites

This book assumes that you have a basic understanding of R. Regardless of whether you are new to R or not, we highly recommend that you become familiar with 'R for Data Science' by Garrett Grolemund and Hadley Wickham (http://r4ds.had.co.nz/). Their book covers how to import, visualize, and summarize data in R using the tidyverse collection of R packages (https://www.tidyverse.org/). It also provides an invaluable framework for organizing your workflow to create clean, reproducible code (http://r4ds.had.co.nz/workflow-projects.html). We follow their lead by, wherever possible, using the tidyverse framework throughout this book.

1.3 Sample datasets

Throughout this book we use subsets of real datasets to illustrate how to access, manage, explore and analyze Motus data in R. We recommend that you run through the sample code in each chapter with the sample dataset **before** running through with your own data, because you will undoubtedly need to modify the code we provide in order to deal most effectively with your own data (every situation is different).

Chapters 2 through 6 use a subset of data from the James Bay Shorebird Project. The James Bay Shorebird Project conducts monitoring and research on shorebirds staging along the James Bay coast, and is a collaborative effort among the Ontario Ministry of Natural Resources and Forestry, Bird Studies Canada, Trent University, and Environment and Climate Change Canada's Canadian Wildlife Service, in conjunction with a larger conservation initiative involving James Bay First Nations and Nature Canada. The Royal Ontario Museum was a contributing partner until 2016. The goals of the project are to 1) improve the ability to estimate indices of abundance and population trends for shorebird species staging along the western James Bay coast, 2) understand movement patterns and their causes, and 3) identify the relative importance of shorebird staging sites and their habitats. Collectively, this information will aid in the development of conservation measures for Red Knot and other shorebird species through habitat protection like Western Hemisphere Shorebird Reserve Network (WHSRN) designation. More information can be viewed on the James Bay Shorebird Project website at https://www.jamesbayshorebirdproject.com/, on Facebook https://www.facebook.com/jamesbayshorebirdproject/, or by contacting their project lead:

Christian Friis Wildlife Biologist Canadian Wildlife Service Environment and Climate Change Canada / Government of Canada christian.friis@canada.ca / Tel: 416.739.4908

Biologiste de la Faune
Service Canadien de la Faune Environnement et Changement Climatique Canada / Gouvernement du Canada christian.friis@canada.ca / Tél. : 416.739.4908

In Chapter 7, we use a subset of data collected by the Motus project 'Studies of Migratory Birds and Bats, 2014-2017' (Projects #20 and #50) to illustrate the calculation of vanishing bearings of birds departing a stopover site. This project holds Motus data for several Western University projects that took place in southern Ontario, Canada. These projects were led by principal investigators (Chris Guglielmo and Yolanda Morbey) and a number of their graduate students. A variety of species of birds and bats were tracked. For more information contact:

Chris Guglielmo, Professor, Department of Biology, Western University, Canada, cguglie2@uwo.ca / Tel: 519.661.2111 (ext. 81204)

Yolanda Morbey, Associate Professor, Department of Biology, Western University, Canada, ymorbey@uwo.ca / Tel: 519.661.2111 (ext. 80116)

1.4 Acknowledgements

Some of the text included in this book was adapted from John Brzustowski's GitHub repository for the `motus` R package at: https://github.com/jbrzusto/motus.

Motus was conceived as the SensorGnome network by Philip Taylor and John Brzustowski at Acadia University. Initial expansion of the network was supported by a Canada Foundation for Innovation Grant to Western University (Dr. Christopher Guglielmo), The University of Guelph (Dr. Ryan Norris), and Acadia University (Dr. Philip Taylor). The development of the Motus web interface, R package, and accompanying handbook were made possible through a Canarie grant to Bird Studies Canada (https://www.canarie.ca/). Motus continues to grow as a program of Bird Studies Canada, through the collaboration of numerous independent researchers, organizations, and individuals. A non-exhaustive list of Motus partners and collaborators can be found at https://motus.org/data/partners.jsp. If your organization is not listed, please contact motus@birdscanada.org.

Many people have worked together to bring Motus technology, the web interface, and the R-package together. The core 'Motus Team' includes Joey Bernard, John Brzustowski, Tara Crewe, Zoe Crysler, Jeremy Hussell, Catherine Jardine, Steffi LaZerte, Denis Lepage, Stuart Mackenzie, Paul Morrill, and Philip Taylor.

Chapter 2

Loading R Packages

This chapter was contributed by Tara L. Crewe, Zoe Crysler, and Philip Taylor. Revisions by Steffi LaZerte and Denis Lepage

2.1 Installing Motus packages

Two R packages have been developed for Motus users:

1. `motus`: provides functions for downloading and updated detections and deployment data, as well as for creating summary plots, and transforming (add sun rise/sun set times) and analyzing Motus data.
2. `motusData`: provides sample datasets used in some of the chapters of this book.

Motus **users** can install the latest stable versions of the R packages using the following code. As with all R packages, you only need to install the packages once; after installation, you need to load each package (using `library()`) each time you open a new R session.

Please note that some functionalities of the `remotes` package may require updated versions of R and RStudio. To avoid errors, please ensure you are using the most recent releases of R and RStudio, and update your R packages using `update.packages()` in the R console.

To update your existing packages:

```
update.packages()
```

Begin by installing the required packages, if not already installed.

If you have used the older version of `motus` which included use of the `motusClient` package, it is recommended to first uninstall both packages.

```
remove.packages(c("motus", "motusClient"))
```

Then proceed with the installation of the `motus` package

```
install.packages("remotes")
library(remotes)

# install motus
install_github("MotusWTS/motus")

# install motusData package which contains sample datasets, e.g., vanishBearing
# used in Chapter 7
install_github("MotusWTS/motusData")

library(motus)
library(motusData)
```

If you need to update the existing `motus` package, you need to specify `force = TRUE`:

```
# force a re-installation of motus package in case of required updates
install_github("MotusWTS/motus", force = TRUE)

library(motus)
```

If you want to know what version of the motus package you currently have installed:

```
packageVersion("motus")
```

2.1.1 Troubleshooting the installation

Occasionally users run into problems while trying to install or update `motus`. Often this is related to problems with different versions of package dependencies. Here we suggest several solutions.

1. Update all packages during the installation

```
remotes::install_github("MotusWTS/motus", upgrade = "always")
```

2. If the installation of Motus generates errors saying that some of the existing packages cannot be removed, you can try to quit any R session, manually delete the problematic package folder from your R libraries and manually install the package again before trying to install motus. You can also try to set up a custom R library folder with .libPaths() and ensure that you have

full write permissions on that folder, or try to start R in administrator (Windows) or SUDO mode (Linux/Ubuntu) and try installing again.

To set a custom library folder for installing new packages:

```
.libPaths("C:/r-libraries/")
```

3. In some cases, it is easier to upgrade R itself by reinstalling the newest version of R: https://cran.r-project.org/. **Note:** While this results in a nice clean installation with fewer problems, it necessitates the re-installation of R packages which can be time-consuming.

4. If reinstalling R is not an option, you get an error related to packages built under a current version of R, AND updating your packages doesn't help, you can consider overriding the error with the following code. **Note:** This might help you install `motus` but may result in other problems. If possible, it's best to resolve the errors rather than ignoring them.

```
Sys.setenv("R_REMOTES_NO_ERRORS_FROM_WARNINGS"=TRUE)
remotes::install_github("MotusWTS/motus", upgrade = "always")
```

2.2 Installing other packages

Throughout the book, we use `tidyverse`, which is a collection of R packages for data science, including `tidyr`, `dplyr`, `ggplot2`, and `lubridate` for managing and manipulating dates. More information on `tidyverse` can be found at https://www.tidyverse.org/, or by browsing (or better still, thoroughly reading) 'R for Data Science' by Garrett Grolemund and Hadley Wickham (http://r4ds.had.co.nz/). For mapping we also use the `rworldmap`, and `ggmap` packages. These can be installed from CRAN, as follows:

```
install.packages("maps")
library(maps)

install.packages("tidyverse")
library(tidyverse)

install.packages("rworldmap")
library(rworldmap)

install.packages("ggmap")
library(ggmap)
```

We also install but do not load the `plyr` package; we use it directly for the handy `round_any` function (with the code `plyr::round_any()`), but loading it can cause problems with the `dplyr` functions:

```
install.packages("plyr")
```

2.3 Internal data processing

As an animal moves within the detection range of a Motus station, radio transmissions, or 'bursts', are detected by antenna(s) and recorded by a receiver. These raw detection data are either uploaded to the Motus database instantaneously via internet connection, or downloaded from the receiver and uploaded to Motus manually. Behind the scenes, various functions read and process the raw detections data to produce the tag detections file that users access using the R package (see Chapter 3). While most users will not need to call on the internal data processing functions, a complete list of functions within the Motus server R package can be found on GitHub (https://github.com/jbrzusto/motusServer). The code behind each function can be viewed on GitHub, or by typing the following in the R console after loading the R package, replacing `function.name` with the name of the R function of interest:

```
function.name
```

In the next chapter we will examine and load some data.

Chapter 3

Accessing and understanding detections data

This chapter was contributed by Tara L. Crewe, Zoe Crysler, and Philip Taylor. Revisions by Steffi LaZerte and Denis Lepage

Before downloading your detection data, please ensure that you have no pending metadata issues through the online Data Issues page

This chapter will begin with an introduction to the structure of the detections database, followed by instructions on how to download and access the data. At the end, a summary section that includes a script to download, select variables, clean data, and export is provided (see 3.13)

3.1 Data structure

Each tag detection database is stored as an SQLite (`dplyr::src_sqlite`) file with the extension '.motus'. The sqlite format was chosen because:

1. it is **flexible**, allowing for many data formats.
2. it is **accessible** from many software platforms (not just R).
3. it is **appendable**, meaning the database can be created and updated on disk without having to read in and resave the entire contents. This will save time and computer memory when searching to see if any new detections are available for your project or receiver.

The .motus file contains a series of interrelated tables where data are stored in a condensed format to save memory. The following tables are included in your

.motus file;

1. `activity`: data related to radio activity for each hour period (`hourBin`) at each antenna, including a count of the number of short runs used in helping identify false detections.
2. `admInfo`: internal table used to keep track of your the motus package used to create your motus file, and the data version.
3. `antDeps`: metadata related to antenna deployments, e.g., deployment height, angle, antenna type.
4. `batchRuns`: metadata for runIDs and associated batchIDs
5. `batches`: detection data for a given receiver and boot number.
6. `filters`: metadata related to user created filters associated with the specified receiver.

7. `gps`: metadata related to Geographic Positioning System (GPS) position of receiver.
8. `hits`: detection data at the level of individual hits.
9. `meta`: metadata related to the project and datatype (tags vs. receivers) that are included in the .motus file
10. `nodeData`: data related to nodes by `batchID` and time (`ts`)
11. `nodeDeps`; metadata related to nodes
12. `projAmbig`: metadata related to what projects have ambiguous tag detections
13. `projs`: metadata related to projects, e.g., project name, principal investigator.
14. `pulseCounts`: number of radio pulses measured on each antenna over 1 hour periods (`hourBin`).
15. `recvDeps`: metadata related to receiver deployments, e.g., deployment date, location, receiver characteristics.
16. `recvs`: metadata related to receiver serial number and associated Motus deviceID
17. `runs`: detection data associated with a run (continuous detections of a unique tag on a given receiver).
18. `runsFilters`: a list of runIDs associated with user created filters and assigned probabilities.

19. `species`: metadata related to species, e.g., unique identifier, scientific name, common name.
20. `tagAmbig`: metadata related to ambiguous tags, e.g., ambigID and associated motusTagID
21. `tagDeps`: metadata related to tag deployments, e.g., deployment date, location, and species.
22. `tagProp`: metadata related to custom deployment properties entered by the principal investigator (e.g. body weight).
23. `tags`: metadata related to tags, e.g., unique identifier, tag characteristics (e.g., burst interval).

In addition to these tables, there are also 'virtual' tables or 'views', which have been created through queries that merge data from the various tables into a single convenient 'view' that contains all of the fields you are likely to need. The following views are currently included in each .motus file:

1. `allambigs`: lists in long-data format each motusTagID (up to 6) associated with each negative ambigID.
2. `alltags`: provides the full detection data for all tags, and all ambiguous (duplicate) tags, associated with your project. Ambiguous detections are repeated for each motusTagID represented by each ambigID.

3. `alltagsGPS`: same as `alltags` but includes GPS latitude, longitude and altitude (much slower to load on large databases).

Because the file is a `dplyr::src_sqlite` file, all of the `dplyr` functions can be used to filter and summarize the .motus database, without needing to first save the data as a *flat* file (a typical two-dimensional dataframe). The SQL format is very advantageous when you have a large file – the queries using SQL will be substantially faster than those done on a flat dataframe.

3.2 Database types

There are two types of tag detection databases available for download:

1. **receiver database**: includes all detections of any registered tags from a single receiver. A receiver database has a name like SG-1234BBBK5678.motus, where the name is the serial number of the receiver.

2. **project database**: includes all detections of your registered tags from across the Motus network. A tag project database has a name like project-123.motus, where the number is the Motus project ID.

These two databases correspond to the basic model of data sharing:

1. you get all detections of *anyone's* tags by *your* receivers (i.e., one receiver tag database for each receiver you deploy).

2. you get all detections of *your* tags by *anyone's* receivers (i.e., one project tag database for each of your Motus projects).

3.3 Load relevant R packages

Before we begin working with data, we need to load the required packages for this chapter. If you have not yet *installed* these packages (from github and CRAN) then please return to Chapter 2 and do so.

```
library(motus)
library(lubridate)
library(dplyr)
```

3.4 Set system environment

Set the system environment time zone to Greenwich Mean Time (UTC), to ensure that you are always working in UTC. This is a very important step, and should be part of every working session. If you fail to do this, then two problems can arise. Times are stored in the Motus database in UTC, and if you do not keep your environment in UTC, then they can be inadvertently changed during import. Second, if tags have been detected across multiple time zones, then they can also inadvertently be changed.

```
Sys.setenv(TZ = "UTC")
```

3.5 Downloading tag detections

To import tag detections for your project or receiver, you need a numerical project id or character scalar receiver serial number.

The success of the Motus network is dependent on the timely upload of detection data from receivers, and on the maintenance of accurate and up to date tag and receiver metadata by collaborators. After downloading your data from the Motus server, users are encouraged to check for updated detection data (see sections 3.7 and 3.8) and metadata (see section 3.9) each time they run an analysis, because collaborators can add detection data and metadata at any time, and these could influence the completeness of your own detections data.

Be warned that large datasets can take some time (sometimes a few hours) to download from the Motus server when downloading for the first time. After the initial download, loading a .motus file into R and updating for any new data will be near instantaneous.

3.5.1 Download data for a project for the *first time*

All data downloads are completed using the `tagme()` function in the `motus` R package. This function will save an SQLite database to your computer with the extension ".motus"; further details on data structure are in section 3.1. The following parameters are available for the `tagme()` function:

- **`projRecv`**: integer project number OR a character vector receiver serial number.

- **new**: if set to TRUE, it will create a new empty .motus file in your local directory. Do not use this parameter or set it to FALSE if you already have a .motus file.
- **update**: if set to TRUE, will download all available data to your existing .motus file. Must be set to TRUE on your first data download and any subsequent downloads if you wish to check for new data. Set to FALSE if you do not wish to check for new data (e.g., if working offline).
- **dir**: Your .motus data is automatically saved to your working directory, unless you specify a different location using this parameter.
- **forceMeta**: if set to TRUE, it will force an update of metadata to an existing .motus file.

Throughout this book we use sample data (see section 1.3) which has been assigned to project 176.

Let's get started by downloading data by project - this will include all detections of your tags on any receiver.

Note that when downloading data from the Motus server for the **first time**, you must specify `new = TRUE` and `update = TRUE`. You will also be prompted to login (see 3.5.2 below).

Unless the directory that you want your data saved in is stated explicitly within the function call, data will be downloaded to the current working directory.

Lets start by determining what our working directory is so we know where our file will be saved.

```
getwd()
```

Specify the project number you wish to download detections for, in this case the sample data project.

```
proj.num <- 176
```

As this is the first time you are downloading data for project 176, set `new = TRUE` and `update = TRUE`. This will create a .motus file in your current working directory, which was shown above using `getwd()` This will also create an SQL object in your R environment called "sql.motus"

```
sql.motus <- tagme(projRecv = proj.num, new = TRUE, update = TRUE)
```

Alternatively you can specify a different location to save the data by entering your preferred filepath. In this example we save to our data folder using the dir parameter. Note that "./" simply means 'relative to the current folder' (shown by `getwd()`).

```
sql.motus <- tagme(projRecv = proj.num, new = TRUE, update = TRUE,
                   dir = "./data/")
```

> **Note:** You'll need to use the **username** `motus.sample` and the **password** `motus.sample` to access this data (see below for more details)!

Using `tagme()` as shown above will download a file to your working or specified directory called "project-176.motus" for the sample data (the number in the file name corresponds to the project number). The progress of the download process should print on the console; if you are not seeing it, try scrolling down your screen while `tagme()` is running.

In the event that your connection to the Motus server fails prior to a complete download (e.g., due to a poor internet connection), use `tagme(proj.num, update = TRUE)` to continue the download from where it left off, ensuring to specify a directory if it is saved outside the working directory.

3.5.2 User Authentication

3.5.2.1 Login

The first time you call a function using the Motus R package, you will be asked to enter your motus.org username and password in the R console to authenticate your access to project data. This will only happen once per R session. If you do not have a Motus username and password, you can sign up at https://motus.org/data/user/new. Permission to access project data will then be granted by Motus staff or the project principal investigator.

Throughout this book we use sample data (see section 1.3) which has been assigned to project 176. When accessing this data you will need to login using username and password 'motus.sample' in the R console when prompted by the `tagme()` function (see section 3.5.1). It will look like this:

```
> tagme(176, update = TRUE, dir = "./data/")
Please enter a value for login name at motus.org
==>
motus.sample
Please enter a value for password at motus.org
==>
motus.sample
Checking for new data in project 176
src:  sqlite 3.19.3 [F:\Motus\Motus_RPackage\MotusRBoo
tbls: admInfo, allambigs, alltags, antDeps, batches, b
, recvs,
  runs, runsFilters, species, tagAmbig, tagDeps, tags
>
```

To download data for one of your own projects, you simply need to change the project number to that of your own project in the `tagme()` call, and enter your own Motus login/password in the R console when prompted. If you are already logged in as the sample data user, you will need to first logout to download your own data (see 3.5.2.2).

3.5.2.2 Logging out

Once you are logged in under one user account, you will not be able to access data from another account. If you need to logout of the current account to access other data, you can run the code below.

```
motusLogout()
```

3.5.3 Download data for a receiver for the *first time*

We could also download data by receiver through the same process as described above. This will provide you with all detections of any tags on the specified receiver. As there are no receivers registered to sample project 176, **this call will not work**. If you have a receiver registered to your own project, replace the receiver serial number in the tagme call below with the serial number for your own receiver, ensuring that you are logged in using your own credentials (see section 3.5.2.2).

```
proj.num <- "SG-123BBBK1234"
sql.motus <- tagme(projRecv = proj.num, new = TRUE, update = TRUE)
```

This will download a file to your working directory named 'SG-123BBBK1234.motus'.

Some users may wish to work directly with the .motus SQLite file. However, since many users are more familiar with a 'flat' dataframe format, instructions to view the the data as a flat dataframe within R, and on how to export the flat file to .csv or .rds format, are included below. Throughout the majority of this book, we use a flat dataframe format.

3.5.4 Downloading multiple receivers at the same time

If you have a large number of receivers in your project, and wish to get receiver specific data for each one, rather than downloading them one by one as above, we can download them with a simple loop. Note that since the sample project doesn't have any receivers associated with it, this script will not result in a download but you can try it with your own project if you have receivers.

```
# specify the project whose receivers you wish to download:
proj.num <- 176

# get a copy of the metadata only
sql.motus <- tagme(proj.num, new = TRUE, update = FALSE, dir = "./data/")
metadata(sql.motus, proj.num)
tbl.recvDeps <- tbl(sql.motus, "recvDeps")

df.serno <- tbl.recvDeps %>%
  filter(projectID == proj.num) %>%
  select(serno) %>%
  distinct() %>%
  collect() %>% as.data.frame()

# loop through each receiver (may take a while!)
for (row in 1:nrow(df.serno)) {
  tagme(df.serno[row, "serno"], dir = "./data/", new = TRUE, update = TRUE)
}

# Note you can remove the dir argument if you want to save it to your working
# directory, just make sure that you use the same directory in both calls
```

You can also create a list of receivers you'd like to download if you don't want to download project-wide receivers:

```
# create list of receivers you'd like to download
df.serno <- c("SG-AB12RPI3CD34", "SG-1234BBBK4321"))

# loop through each receiver (may take a while!), and save to the working directory
for (k in 1:length(df.serno)) {
  tagme(df.serno[k], new = TRUE, update = TRUE)
}
```

```
# loop through each receiver (may take a while!), and save to a specified directory
for (k in 1:length(df.serno)) {
  tagme(df.serno[k], dir = "/Users/zoecrysler/Downloads/",
        new = TRUE, update = TRUE)
}
```

3.5.5 Updating all .motus files within a directory

Once you have .motus files, you can also update them all by simply calling the `tagme()` function but leaving all arguments blank, apart from the directory:

```
# If you have them saved your working directory:
tagme()

# If you have them saved in a different directory:
tagme(dir = "./data/")
```

3.5.6 Accessing downloaded detection data

Now that we've downloaded our data as an SQLite database and loaded it into an R object called `sql.motus`, we want to access the tables stored within. Detailed descriptions of all the tables stored in the .motus file can be found in section 3.1.

You can also view the list of tables, and variables contained within those tables, using the `DBI` and `RSQLite` packages (these are automatically installed when you install `motus`).

```
library(DBI)
library(RSQLite)

# specify the filepath where your .motus file is saved, and the file name.
file.name <- dbConnect(SQLite(), "./data/project-176.motus")

# get a list of tables in the .motus file specified above.
dbListTables(file.name)
```

```
##  [1] "activity"    "admInfo"     "allambigs"   "alltags"     "alltagsGPS"  "antDeps"
## [12] "hits"        "meta"        "nodeData"    "nodeDeps"    "projAmbig"   "projBat
## [23] "species"     "tagAmbig"    "tagDeps"     "tagProps"    "tags"
```

```
# get a list of variables in the "species" table in the .motus file.
dbListFields(file.name, "species")
```

```
## [1] "id"         "english"    "french"     "scientific" "group"      "sort"
```

The *virtual* table `alltags` contains the detection data, along with most metadata variables that users need from the various underlying .motus tables. We access the tables using the `tbl()` function from the `dplyr` package which we installed in Chapter 2:

```
# this retrieves the "alltags" table from the "sql.motus" SQLite file we read in earlier
tbl.alltags <- tbl(sql.motus, "alltags") # virtual table
```

We now have a new `tbl.alltags` object in R. The underlying structure of these tables is a list of length 2:

```
str(tbl.alltags)
```

```
## List of 2
##  $ src:List of 2
##   ..$ con  :Formal class 'SQLiteConnection' [package "RSQLite"] with 7 slots
##   .. .. ..@ ptr               :<externalptr>
##   .. .. ..@ dbname            : chr "/home/steffi/Projects/Business/BSC/MOTUS/MotusRBook/data
##   .. .. ..@ loadable.extensions: logi TRUE
##   .. .. ..@ flags             : int 70
##   .. .. ..@ vfs               : chr ""
##   .. .. ..@ ref               :<environment: 0x5653b5879a98>
##   .. .. ..@ bigint            : chr "integer64"
##   ..$ disco: NULL
##   ..- attr(*, "class")= chr [1:4] "src_SQLiteConnection" "src_dbi" "src_sql" "src"
##  $ ops:List of 2
##   ..$ x   : 'ident' chr "alltags"
##   ..$ vars: chr [1:59] "hitID" "runID" "batchID" "ts" ...
##   ..- attr(*, "class")= chr [1:3] "op_base_remote" "op_base" "op"
##  - attr(*, "class")= chr [1:5] "tbl_SQLiteConnection" "tbl_dbi" "tbl_sql" "tbl_lazy" ...
```

The first part of the list, `src`, is a list that provides details of the SQLiteConnection, including the directory where the database is stored. The second part is a list that includes the underlying table. Thus, the R object `alltags` is a *virtual* table that stores the database structure and information required to connect to the underlying data in the .motus file. As stated above, the advantage of storing the data in this way is that it saves memory when accessing very large databases, and functions within the dplyr package can be used to manipulate and summarize the tables before collecting the results into a typical 'flat' format dataframe.

If you want to use familiar functions to get access to components of the underlying data frame, then use the `collect()` function. For example, to look at the names of the variables in the alltags table:

```
tbl.alltags %>%
  collect() %>%
  names() # list the variable names in the table
```

```
##  [1] "hitID"          "runID"          "batchID"        "ts"             "tsCorrecte
## [10] "freqsd"         "slop"           "burstSlop"      "done"           "motusTagID
## [19] "tagProjID"      "mfgID"          "tagType"        "codeSet"        "mfg"
## [28] "pulseLen"       "tagDeployID"    "speciesID"      "markerNumber"   "markerType
## [37] "tagDepAlt"      "tagDepComments" "fullID"         "deviceID"       "recvDeploy
## [46] "recvDeployName" "recvSiteName"   "isRecvMobile"   "recvProjID"     "recvUtcOff
## [55] "speciesFR"      "speciesSci"     "speciesGroup"   "tagProjName"    "recvProjNa
```

3.5.7 Adding GPS Data

If GPS data is required, users will either have to use the `alltagsGPS` view, or will have to add GPS values to a data subset.

To use the `alltagsGPS` view (note that this can be slow, which is why GPS data is excluded by default):

```
tbl.alltagsGPS <- tbl(sql.motus, "alltagsGPS")
```

Alternatively, to speed things up you can first filter your data, then use the `getGPS()` function to retrieve GPS values.

```
# Filter the data
tbl.Dunlin <- tbl(sql.motus, "alltags") %>%
  filter(speciesEN == "Dunlin") %>%
  collect()

# Get the daily median location of GPS points for these data
Dunlin.GPS <- getGPS(src = sql.motus, data = tbl.Dunlin)
```

We match GPS locations to `hitID`s according to one of several different time values, specified by the `by` argument.

This can be one of three options:

1. the median location within **`by = X`** minutes of a `hitID`
   - here, `X` can be any number greater than zero and represents the size of the time block in minutes over which to calculate a median location
   - be aware that you should ideally not chose a period smaller than the frequency at which GPS fixes are recorded, or some hits will not be associated with GPS
2. **`by = "daily"`** median location (**default**, implied in the above example)
   - similar to `by = X` except the duration is 24hr (same as `by = 1440`)
   - this method is most suitable for receiver deployments at fixed location.
3. or the **`by = "closest"`** location in time
   - individual GPS lat/lons are returned, matching the closest `hitID` timestamp
   - this method is most accurate for mobile deployments, but is potentially slower than `by = X`.

- you can also specify a `cutoff` which will only match GPS records which are within `cutoff = X` minutes of the hit. This way you can avoid having situations where the 'closest' GPS record is actually days away.

```
Dunlin.GPS <- getGPS(src = sql.motus, data = tbl.Dunlin, by = 60)
Dunlin.GPS <- getGPS(src = sql.motus, data = tbl.Dunlin, by = "closest", cutoff = 120)
```

To keep all `hitID`s, regardless of whether they GPS data or not, use the argument `keepAll = TRUE`.

```
Dunlin.GPS <- getGPS(src = sql.motus, data = tbl.Dunlin, keepAll = TRUE)
```

Finally, use the `left_join()` function to add GPS to your data subset. Note that this sample data doesn't actually have GPS values so these examples are for illustration only.

```
tbl.Dunlin.GPS <- left_join(tbl.Dunlin, Dunlin.GPS, by = "hitID")
```

As there is no GPS data in this example data set, we'll continue with our original `tbl.alltags` file.

3.5.8 Converting to flat file

To convert the `alltags` view or other table in the .motus file into a typical 'flat' format, i.e., with every record for each field filled in, use the `collect()` and `as.data.frame()` functions. The output can then be further manipulated, or used to generate a RDS file of your data for archiving or export.

```
df.alltags <- tbl.alltags %>%
  collect() %>%
  as.data.frame()
```

Now we have a flat dataframe of the alltags table called `df.alltags`. We can look at some metrics of the file:

```
names(df.alltags)      # field names
str(df.alltags)        # structure of your data fields
head(df.alltags)       # prints the first 6 rows of your df to the console
summary(df.alltags)    # summary of each column in your df
```

Note that the format of the time stamp (ts) field is numeric and represents seconds since January 1 1970. We recommend that when you transform your tables into flat dataframes, that you format the time stamp using the lubridate package at that time, e.g.:

```
df.alltags <- tbl.alltags %>%
  collect() %>%
  as.data.frame() %>%     # for all fields in the df (data frame)
```

```
  mutate(ts = as_datetime(ts, tz = "UTC", origin = "1970-01-01"))

# the tz = "UTC" is not necessary here, provided you have set your system time to UTC
# ... but it serves as a useful reminder!
```

Note that time stamps can only be manipulated in this way *after* collecting the data into a flat dataframe.

If you want to load only part of your entire virtual table (e.g. certain fields, certain tags, or all tags from a specified project or species), you can use `dplyr` functions to filter the data before collecting into a dataframe. Some examples are below:

1. To select certain variables:

```
# to grab a subset of variables, in this case a unique list of Motus tag IDs at
# each receiver and antenna.
df.alltagsSub <- tbl.alltags %>%
  select(recv, port, motusTagID) %>%
  distinct() %>%
  collect() %>%
  as.data.frame()
```

2. To select certain tag IDs:

```
# filter to include only motusTagIDs 16011, 23316
df.alltagsSub <- tbl.alltags %>%
  filter(motusTagID %in% c(16011, 23316)) %>%
  collect() %>%
  as.data.frame() %>%
  mutate(ts = as_datetime(ts, tz = "UTC", origin = "1970-01-01"))
```

3. To select a specific species:

```
# filter to only Red Knot (using speciesID)
df.4670 <- tbl.alltags %>%
  filter(speciesID == 4670) %>%
  collect() %>%
  as.data.frame() %>%
  mutate(ts = as_datetime(ts, tz = "UTC", origin = "1970-01-01"))

# filter to only Red Knot (using English name)
df.redKnot <- tbl.alltags %>%
  filter(speciesEN == "Red Knot") %>%
  collect() %>%
  as.data.frame() %>%
  mutate(ts = as_datetime(ts, tz = "UTC", origin = "1970-01-01"))
```

Using `dplyr`, your virtual table can also be summarized before converting to a

flat file. For example, to find the number of different detections for each tag at each receiver:

```
df.detectSum <- tbl.alltags %>%
  count(motusTagID, recv) %>%
  collect() %>%
  as.data.frame()
```

In later chapter(s) we will show you additional ways of summarizing and working with your data.

3.6 Export your 'flat' dataframe to CSV or RDS file

A good workflow is to create a script that deals with all your data issues (as described in later chapters), and then saves the resulting dataframe for re-use. If you do this, you can quickly start an analysis or visualization session from a known (and consistent) starting point. We use an .rds file, which preserves all of the associated R data structures (such as time stamps).

```
saveRDS(df.alltags, "./data/df_alltags.rds")
```

Some users may also want to export the flat dataframe into a .csv file for analysis in other programs. This can easily be done with the following code. Note that it **does not** preserve time stamps:

```
write.csv(df.alltags, "./data/df_alltags.csv")
```

3.7 Update and/or open an existing database

As you or other users upload data to our server, you may have additional tag detections that weren't present in your initial data download. Since the .motus file is a SQLite database, you can update your existing file with any newly available data, rather than doing a complete new download of the entire database. To open and update a detections database that already exists (has been downloaded previously), we use the `tagme()` function but set `new = FALSE`:

```
sql.motus <- tagme(projRecv = proj.num, new = FALSE, update = TRUE, dir = "./data/")
```

If you are working offline, and simply want to open an already downloaded database without connecting to the server to update, use `new = FALSE` and `update = FALSE`:

```
# use dir = to specify a directory
sql.motus <- tagme(projRecv = proj.num, new = FALSE, update = FALSE)
```

3.8 Check if new data are available

To check if new data are available for your project or receiver without downloading the data, you can use the `tellme()` function, which returns a list with:

- **numHits**: number of new tag detections.
- **numBytes**: approximate uncompressed size of data transfer required, in megabytes.
- **numRuns**: number of runs of new tag detections, where a run is a series of continuous detections for a tag on a given antenna.
- **numBatches**: number of batches of new data.
- **numGPS**: number of GPS records of new data.

The following assumes that a local copy of the database already exists:

```
tellme(projRecv = proj.num)                    # If db is in the working directory
tellme(projRecv = proj.num, dir = "./data/")   # To specify a different directory
```

To check how much data is available for a project but you *do not* have a database for it, use the 'new' parameter:

```
tellme(projRecv = proj.num, new = TRUE)
```

3.9 Force an update/re-import of tag and receiver deployment metadata

Tag and receiver metadata are automatically merged with tag detections when data are downloaded. However, if metadata have been updated since your initial download, you can force re-import of the metadata when updating a database by running:

```
sql.motus <- tagme(projRecv = proj.num, forceMeta = TRUE)
```

3.10 Import full tag and receiver metadata

When you use `tagme()` to download or update your .motus file, you are provided with the metadata for:

1. any tags registered to your project which have detections;
2. tags from other projects which are associated with ambiguous detections (see Chapter 5) in your data;
3. receivers that your tags and any ambiguous tags were detected on.

In many instances, you will want access to the full metadata for all tags and receivers across the network, e.g., to determine how many of your deployed tags

were not detected, or to plot the location of stations with and without detections. The `metadata()` function can be used to add the complete Motus metadata to your .motus file. The metadata function only needs to be run once, but we suggest that you re-import the metadata occasionally to ensure that you have the most recent and up-to-date information.

Running the metadata function as follows will add the appropriate metadata from across the network (all tags and all receivers) to the `recvDeps` and `tagDeps` tables in your .motus file:

```
# access all tag and receiver metadata for all projects in the network.
metadata(sql.motus)
```

Alternatively, you can load metadata for a specific project(s) using:

```
# access tag and receiver metadata associated with project 176
metadata(sql.motus, projectIDs = 176)

# access tag and receiver metadata associated with projects 176 and 1
metadata(sql.motus, projectIDs = c(176, 1))
```

3.11 Ensure that you have the correct database version

When you call the tagme function to load the sqlite database, the version of the R package used to download the data is stored in an admInfo table. Over time, changes will be made to the functionality of the R package that may require adding new tables, views or fields to the database. If your version of the database does not match the version of the R package, some of the examples contained in this book may not work. The following call will check that your database has been updated to the version matching the current version of the `motus` R package. If your database does not match the most current version of the R package, use `tagme()` with `update = TRUE` to update your database to the correct format. Refer to Appendix B if the `checkVersion()` call returns a warning.

```
checkVersion(sql.motus)
```

3.12 R object naming convention

Throughout this chapter and the rest of the book, we name R objects according to their structure and the source of the data contained in the object. So, SQLite objects will be prefixed with `sql.`, virtual table objects will be prefixed with `tbl.`, and dataframe objects will be prefixed with `df.`; the rest of the name will

include the name of the .motus table that the data originates from. Throughout the rest of the book we will be relying on and referencing the naming formats below; please ensure that you are familiar with these before continuing to the next chapter. The following code assumes you have already downloaded the sample data and do not need to update it; if you have not downloaded the data, see section 3.5.1 for instructions on initial download:

```
# SQLite R object, which links to the .motus file:
sql.motus <- tagme(176, update = TRUE, dir = "./data")

# virtual table object of the alltags table in the sample.motus file:
tbl.alltags <- tbl(sql.motus, "alltags")
df.alltags <- tbl.alltags %>%
                collect() %>%
                as.data.frame() %>% # dataframe ("flat") object of alltags table
                mutate(ts = as_datetime(ts, tz = "UTC", origin = "1970-01-01"))
```

3.13 Summary: download, access, and export data

Throughout the book we will predominantly be working with the `alltags` table in flat format. Although described in detail above, here we include a quick summary of how you download, access, and convert the sample data for use in the rest of the book. In later sections of the book, we include additional recommended modifications and filtering of unnecessary variables, more information is available in section 5.1 in Chapter 5.

```
# set proj.num to 176 for sample data
proj.num <- 176

# - download and load detection data from the sample project and save in data folder
# - login and password for sample data is "motus.sample"
# - if you are accessing already-downloaded data, use new = FALSE; if you don't
#   want to update your data, also set update = FALSE
sql.motus <- tagme(proj.num, new = TRUE, update = TRUE, dir = "./data/")

# access the "alltags" table within the SQLite file
tbl.alltags <- tbl(sql.motus, "alltags")

# convert "tbl.alltags" to a flat dataframe and change numeric time to a datetime obje
df.alltags <- tbl.alltags %>%
              collect() %>%
              as.data.frame() %>%
              mutate(ts = as_datetime(ts, tz = "UTC", origin = "1970-01-01"))
```

For your own data we suggest creating a script with the following workflow:

1. download and/or update your data
2. select variables of interest for the table you are working with (typically `alltags`)
3. include any initial cleaning
4. save the resulting data as an .rds file as described in section 3.6. We suggest using RDS instead of CSV, because the RDS format preserves the underlying structure of the data (e.g. POSIX times stay as POSIX times). If you want to export your data to another program, then a CSV format might be preferred.

We caution that producing a flat file using the full suite of fields can use a lot of memory, and can slow R down considerably when dealing with large datasets. For some combinations of data sets and computers, it may be impossible to directly use data frames in R. If that is the case, then this is the point in your workflow where you should carefully consider the information you need from within your data set (for example, how it is aggregated) and simplify it. You can always return to this script and creating a new RDS file with different variables, or aggregated at a different scale.

To read in a saved RDS file you can run:

```
# reads in your file "df.alltags.rds" saved in the data folder
df.alltags.saved <- readRDS("./data/df_alltags.rds")
```

In the next chapter we will check for missing metadata.

Chapter 4

Tag and Receiver Deployments

This chapter was contributed by Tara L. Crewe, Zoe Crysler, and Philip Taylor.

Before working with your detection data, a first step is to summarize and visualize the metadata for tag and receiver deployments registered to your project. Summarizing and plotting your deployments can be an effective way to find any errors in tag or receiver deployment metadata, which can in turn influence the completeness of the detections data for your project and the projects of others with detections of their own tags on your receivers.

This chapter is a complement to the online Data Issues page, which provides each project with a list of metadata issues (missing or outlying values) to be accepted or ignored. As such, please address any and all errors associated with your project on the Data Issues page **before** importing your data through R. This chapter does not provide a full check of your deployment metadata, but will help uncover errors that have been missed by the automatic queries on the Data Issues page.

We use the James Bay Shorebird Project sample dataset throughout this chapter (see Section 1.3). As you run through the code to look at your own deployments, **please fix any errors or omissions in your metadata by signing in to https://motus.org/**, and under the 'Manage Data' tab, select either 'Manage Tags' to fix tag deployment metadata or 'Manage Receivers' to fix receiver deployment metadata. It is important to fix metadata errors online, so that errors are fixed at the source and archived on the Motus Server, ensuring all users have access to the correct tag and receiver metadata. Metadata corrected online will automatically be corrected in your detection files. If you have already downloaded your detection data, you can update the existing file to include new metadata and detections (see sections 3.9, 3.7).

4.1 Load relevant R packages and set working environment

Before we begin working with data, we need to load the required packages for this chapter. If you have not yet installed these packages (from GitHub or CRAN) then please return to Chapter 2 and do so.

```
library(tidyverse)
library(motus)
library(lubridate)

# Set the system environment time zone to UTC (to ensure that you are always working in UTC)
Sys.setenv(TZ = "UTC")
```

4.2 Load .motus file

This chapter assumes that the .motus file has already been downloaded, if you have not done so please return to Chapter 3 for instructions on how to do so. To update and load the existing file into R, use `tagme()`, you may have to login as described in the previous chapter with username **and** password 'motus.sample'

```
proj.num <- 176

sql.motus <- tagme(proj.num, update = TRUE, dir = "./data")
```

4.3 Tag Deployments

In your .motus file, when using the tagme function, you are only provided with the metadata for any tags from your project with detections along with metadata for associated ambiguous tags from other projects, and receiver metadata for stations where your tags were detected. Here we will:

1. download full tag metadata for our project only
2. determine how many tags are registered to your project
3. determine how many of those registered tags were deployed
4. determine location of tag deployments
5. determine completeness and accuracy of tag deployment metadata

We will run through each of these in sequence.

4.3.1 Download full project tag metadata

Incomplete metadata or missing tag registrations can result in missing detection data. We therefore want to assess the completeness of all tags registered to our projects - not just tags for which we have detections. In order to to this we will use the `metadata()` function for project 176, described in more detail in section 3.10.

```
metadata(sql.motus, projectIDs = proj.num)
```

4.3.2 Number of registered tags

Now that we have complete tag metadata for our project, we can check the number of tags registered by loading the `tags` table in the .motus file. The `tags` table contains the metadata of each registered tag, including a unique tagID and information on manufacturer, model, nominal and offset frequency, burst interval, and pulse length. The `tags` table does not include deployment information. We select the metadata specific to the James Bay Shorebird Project, and ignore tag metadata associated with any duplicate tags belonging to other projects:

```
tbl.tags <- tbl(sql.motus, "tags")
df.tags <- tbl.tags %>%
  filter(projectID == proj.num) %>%
  collect() %>%
  as.data.frame()
```

The number of rows in the `df.tags` database is equivalent to the number of tags registered to the James Bay Shorebird Project in the sample dataset (i.e., 18 tags):

```
nrow(df.tags) # number of registered tags in the database
```

```
## [1] 18
```

You can view the motusTagIDs:

```
unique(df.tags$tagID)
```

```
##  [1] 16011 16035 16036 16037 16038 16039 16044 16047 16048 16052 17357 19129 22867 1
```

If you are missing registered tags, please follow the instructions at https://motus.org/tag-registration/.

4.3.3 Number of registered tags that were deployed

The tag deployment metadata table (`tagDeps`) in the .motus file is required to check which registered tags have deployments. This file includes the date, time,

species, and location of tag deployment. The database is subset to project '176', and we use the `anti_join()` function to determine which registered tags have (or do not have) corresponding deployment information.

```
tbl.tagDeps <- tbl(sql.motus, "tagDeps")

df.tagDeps <- tbl.tagDeps %>%
  filter(projectID == proj.num) %>%
  collect() %>%
  as.data.frame() %>% # once in df format, can format dates with lubridate
  mutate(tsStart = as_datetime(tsStart, tz = "UTC", origin = "1970-01-01"),
         tsEnd = as_datetime(tsEnd, tz = "UTC", origin = "1970-01-01"))

anti_join(df.tags, df.tagDeps, by = "tagID")
```

```
##  [1] tagID        projectID    mfgID        type         codeSet      manufacturer model
## <0 rows> (or 0-length row.names)
```

In the sample data, there are no registered tags without deployment metadata, which suggests that all tags were deployed. If you have undeployed tags in your own files, please check your records to ensure this is the case; without deployment metadata, detections for registered but 'undeployed' tags will be missing from your detections database.

4.3.4 Number of deployments per tag

A tag might be deployed more than once, for example, if a previously deployed tag was recovered, and then later re-deployed on another individual. When tags are deployed more than once, the detections data should be considered independently for each deployment.

Throughout this book we use the `motusTagID` as a unique identifier for a deployment. However, when a tag is deployed more than once, the `motusTagID` will remain consistent between deployments, and we instead need to use the `tagDeployID`, or combination of `motusTagID` and `tagDeployID` to distinguish which detections belong to which deployment.

Here, we check whether there are any tags with more than one deployment in the sample data (there are none), and then show you how to make a combined tagID/deployID variable to use in place of the `motusTagID` if you have multiple deployments of a tag in your own data:

```
df.alltags %>%
  select(motusTagID, tagDeployID) %>%
  filter(!(is.na(tagDeployID))) %>% # remove NA tagDeployIDs
  distinct() %>%
  group_by(motusTagID) %>%
```

```
  mutate(n = n()) %>%
  filter(n > 1)
```

```
## # A tibble: 0 x 3
## # Groups:   motusTagID [0]
## # ... with 3 variables: motusTagID <int>, tagDeployID <int>, n <int>
```

If you do have multiple deployments for a tag, we recommend creating a `motusTagDepID` to use in place of the `motusTagID` to define unique deployments of a tag. Moving forward, you would use `motusTagDepID` in place of `motusTagID` as you work through the rest of the book:

```
df.alltags <- df.alltags %>%
  mutate(motusTagDepID = paste(motusTagID, tagDeployID, sep = "."))

# and do the same for the tag metadata

df.tagDeps <- df.tagDeps %>%
  mutate(motusTagDepID = paste(tagID, deployID, sep = "."))
```

4.3.5 Location of tag deployments

Creating a map of your tag deployments can point out any obvious errors in the tag deployment latitude or longitude that weren't captured by the online metadata message centre queries.

a. Load base map files

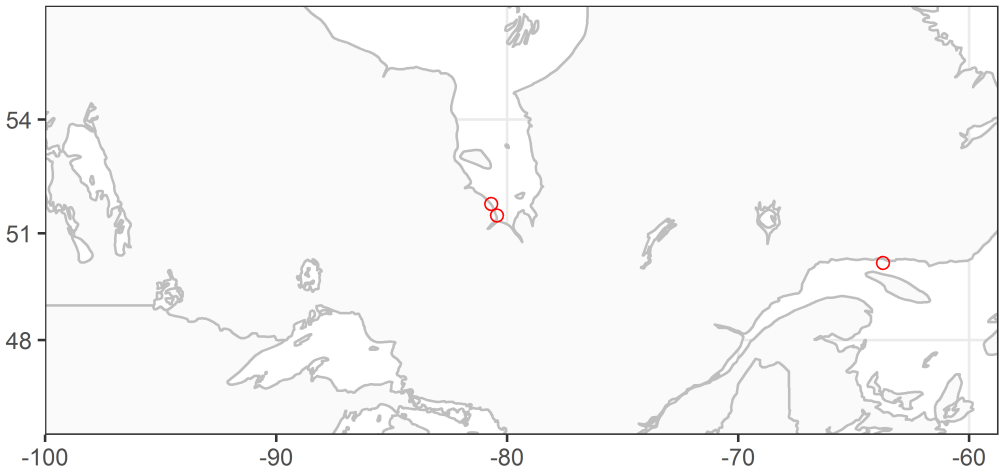
Load base map files from the `rworldmap` package:

```
na.lakes <- map_data(map = "lakes")
na.lakes <- mutate(na.lakes, long = long - 360)

# Include all of the Americas to begin
na.map <- map_data(map = "world2")
na.map <- filter(na.map, region %in% c("Canada", "USA"))

na.map <- mutate(na.map, long = long- 360)

# Others countries in the Americas that you may want to plot, depending on your
# location: "Mexico", "lakes","Belize", "Costa Rica", "Panama", "Guatemala",
# "Honduras", "Nicaragua", "El Salvador", "Colombia", "Venezuela", "Ecuador",
# "Peru", "Brazil", "Guyana","Suriname", "Bolivia", "French Guiana", "Jamaica",
# "Cuba", "Haiti", "Dominican Republic", "The Bahamas", "Turks and Caicos
# Islands", "Puerto Rico", "British Virgin Islands", "Montserrat", "Dominica",
# "Saint Lucia", "Barbados", "Grenada", "Trinidad and Tobago", "Chile",
# "Argentina", "Uruguay"
```

b. Map the locations of tag deployments

Map the location of tag deployments for the sample data:

```
# set limits to map based on locations of detections, ensuring they include the
# deployment locations
xmin <- -100 #min(df.tagDeps$longitude, na.rm = TRUE) - 5
xmax <- max(df.tagDeps$longitude, na.rm = TRUE) + 5
ymin <- min(df.tagDeps$latitude, na.rm = TRUE) - 5
ymax <- max(df.tagDeps$latitude, na.rm = TRUE) + 5

# map using ggplot
ggplot(data = na.lakes, aes(x = long, y = lat)) +
  geom_polygon(data = na.map, aes(long, lat, group = group),
               colour = "grey", fill="grey98") +
  geom_polygon(aes(group = group), colour = "grey", fill = "white") +
  coord_map(projection = "mercator",
            xlim = c(xmin, xmax),
            ylim = c(ymin, ymax)) +
  labs(x = "", y = "") +
  theme_bw() +
  geom_point(data = filter(df.tagDeps, projectID == 176),
             aes(longitude, latitude), size = 2, shape = 1, colour = "red")
```



If there are any errors in tag deployment location, please correct these online at https://motus.org/data/.

4.3.6 Check completeness and accuracy of tag deployment metadata

Required tag metadata includes deployment start date/time, end date/time (if applicable), deployment latitude, deployment longitude, and species. Lack of information on deployment date, time, and location in particular can influence the estimated lifespan of your tag, and therefore whether the tagFinder will 'look' for your tag at the appropriate time(s). It can also increase the potential for ambiguities with duplicate tags in the system.

a. Look at range of metadata values

As a first step, use `summary(df.tagDeps)` to get an idea of the range of each variable, and whether any variables have missing (`NA`) or odd values. The following summarizes a subset of the variables in the `df.tagDeps` database. There are several things to consider: are the range of start and end dates reasonable for your deployments, or are there obvious errors in the timing of deployments? Is the range in deployment latitude and longitude reasonable? Are the values for species IDs correct?

```
df.tagDeps %>%
  select(tagID, projectID, tsStart, tsEnd, speciesID, latitude, longitude) %>%
  summary()
```

```
##      tagID         projectID      tsStart                       tsEnd
##  Min.   :16011   Min.   :176   Min.   :2015-08-02 11:40:00   Min.   :2015-12-17 11:4
##  1st Qu.:16038   1st Qu.:176   1st Qu.:2015-08-13 15:25:00   1st Qu.:2015-12-28 15:1
##  Median :16050   Median :176   Median :2015-09-10 17:50:30   Median :2016-03-10 17:5
##  Mean   :18616   Mean   :176   Mean   :2016-01-24 12:49:36   Mean   :2016-07-28 18:0
##  3rd Qu.:22890   3rd Qu.:176   3rd Qu.:2016-09-25 15:34:15   3rd Qu.:2017-06-06 09:5
##  Max.   :23319   Max.   :176   Max.   :2016-10-15 16:00:00   Max.   :2017-06-26 16:0
```

There are no missing start dates (`tsStart`), and deployment start dates range from 2015 to 2016, which is reasonable for this project.

The species IDs are numeric, and somewhat meaningless without an ability to assign an actual species name to the numeric ID, which we do next, however there are no missing values.

b. Check that species IDs are appropriate for your data

The `species` table in the .motus file associates each numeric species ID with an English, French, and scientific name. We load that table, and subset to the suite of numeric `speciesID`s in the tag metadata:

```
# generate list of species IDs in project 176 metadata
sp.list <- unique(df.tagDeps$speciesID)

# Species metadata
tbl.species <- tbl(sql.motus, "species")
```

```
tbl.species %>%
  filter(id %in% sp.list) %>%
  collect() %>%
  as.data.frame()
```

```
##     id                english                       french             scientific group sort
## 1 4180    Semipalmated Plover           Pluvier semipalmé Charadrius semipalmatus BIRDS   NA
## 2 4670              Red Knot          Bécasseau maubèche        Calidris canutus BIRDS   NA
## 3 4680             Sanderling         Bécasseau sanderling           Calidris alba BIRDS   NA
## 4 4690 Semipalmated Sandpiper          Bécasseau semipalmé        Calidris pusilla BIRDS   NA
## 5 4760 White-rumped Sandpiper   Bécasseau à croupion blanc    Calidris fuscicollis BIRDS   NA
## 6 4780     Pectoral Sandpiper Bécasseau à poitrine cendrée      Calidris melanotos BIRDS   NA
## 7 4820                Dunlin           Bécasseau variable         Calidris alpina BIRDS   NA
```

This lists all species that are included in the tag deployment metadata for the project. If there are species that do not make sense, this is likely due to a data entry error when assigning a deployment to a species. You can look for records in your tag metadata that are associated with a particular `speciesID` using the following code; you would then use the `deployID` associated with the entry/entries to find and update the deployment record in your project metadata online:

```
filter(df.tagDeps, speciesID == 4780)
```

```
##   deployID tagID projectID status             tsStart               tsEnd deferSec speciesID b
## 1    10517 22867       176   <NA> 2016-09-06 15:35:00 2017-05-18 15:35:00       NA      4780
##
## 1 Sex:F, Age:HY, Bill28, Tarsus:26.2, Wing Chord:123, Wing Flat:129, Mass:57.7, Flag: (FEW)7P6
##   id bi tsStartCode tsEndCode                                fullID motusTagDepID
## 1 NA NA          1L        3L SampleData#272.1:5.3@166.38(M.22867)   22867.10517
```

Please remember, any metadata corrections need to be made online

4.4 Check Receiver Metadata

There are two sources of receiver metadata in Motus detection data: receivers registered to your own project, and receivers registered to the projects of others. You can access metadata for all receivers in the network, because negative data (i.e., my tag was *not* detected at station X even though station X was active) is often as important as positive data. It also allows you to map where your tags were detected relative to the distribution of receivers throughout the Motus network.

Receiver metadata errors or omissions that you find in your .motus file can only be fixed for receivers registered to your own project.

All users are encouraged to enter complete and accurate receiver metadata for the benefit of the entire network. If you anticipate needing specific information on receiver or antenna deployments for stations deployed by others, please consider using the Motus discussion group (https://motus.org/discussion/) to request that other registered users record the receiver deployment details you will need; be specific about the exact receiver deployment details you are interested in, and when and where in the network your tags will be deployed and potentially detected.

In the following steps we will:

1. download full receiver metadata across the network
2. determine number of project receiver deployments
3. determine timing of project receiver deployments
4. determine location of network-wide and project receiver deployments
5. determine completeness and accuracy of receiver metadata

4.4.1 Download full receiver metadata

Later on in this chapter we will want to map all receivers in the network, so we will now load metadata from all projects, as opposed to simply project 176 as we did above. The `metadata()` function is described in more detail in section 3.10.

```
metadata(sql.motus)
```

4.4.2 Number of project receiver deployments

To see which (if any) receiver deployments are registered to your project, import, subset and summarize the receiver deployment data:

```
tbl.recvDeps <- tbl(sql.motus, "recvDeps")

df.projRecvs <- tbl.recvDeps %>%
  filter(projectID == proj.num) %>%
  collect() %>%
  as.data.frame() %>%
  mutate(tsStart = as_datetime(tsStart, tz = "UTC", origin = "1970-01-01"),
         tsEnd = as_datetime(tsEnd, tz = "UTC", origin = "1970-01-01"))

summary(df.projRecvs)
```

```
##     deployID       serno           receiverType          deviceID       macAddress
##  Min.   :1134   Length:18          Length:18          Min.   : 74.00   Length:18
##  1st Qu.:2287   Class :character   Class :character   1st Qu.: 75.75   Class :chara
##  Median :3101   Mode  :character   Mode  :character   Median :280.00   Mode  :chara
##  Mean   :2952                                         Mean   :250.11
```

```
##  3rd Qu.:4002                                        3rd Qu.:333.00
##  Max.   :4221                                        Max.   :528.00
##
##    longitude         isMobile         tsStart                       tsEnd
##  Min.   :-80.80   Min.   :0.0000   Min.   :2014-07-12 00:00:00   Min.   :2014-11-06 00:00:00
##  1st Qu.:-80.63   1st Qu.:0.0000   1st Qu.:2015-05-24 06:00:00   1st Qu.:2015-10-20 00:00:00
##  Median :-80.57   Median :0.0000   Median :2016-05-17 12:00:00   Median :2016-03-16 21:05:00
##  Mean   :-80.47   Mean   :0.1667   Mean   :2016-01-23 14:03:16   Mean   :2016-03-25 17:06:15
##  3rd Qu.:-80.45   3rd Qu.:0.0000   3rd Qu.:2016-08-04 01:26:15   3rd Qu.:2016-12-01 00:00:00
##  Max.   :-79.81   Max.   :1.0000   Max.   :2017-08-20 23:30:00   Max.   :2017-08-20 23:30:00
##  NA's   :3                                                      NA's   :2
```

There are 18 receiver deployments registered to the sample project. Four deployments are missing latitude and longitude, and six deployments are missing end dates, which suggests that those receivers are still deployed.

The following code keeps only variables of interest (by removing those we do not need), and arranges the remaining records by receiver ID, latitude, and start date:

```
df.projRecvs %>%
  mutate(dateStart = date(tsStart)) %>%
  select(-serno,-fixtureType, -macAddress, -tsStart, -tsEnd, -elevation,
         -projectID, -status, -receiverType, -siteName) %>%
  arrange(deviceID, latitude, dateStart)
```

```
##    deployID deviceID             name latitude longitude isMobile utcOffset  dateStart
## 1      3100       74       Washkaugou  51.1540  -79.8144        0        NA 2016-05-18
## 2      2291       75      North Bluff  51.4839  -80.4500        0        NA 2015-05-25
## 3      3102       75      North Bluff  51.4839  -80.4501        0        NA 2016-05-18
## 4      4051       75      North Bluff  51.4839  -80.4501        0        NA 2017-05-17
## 5      4221       75      North Bluff  51.4839  -80.4501        0        NA 2017-08-20
## 6      3103       78       Piskwamish  51.6579  -80.5678        0        NA 2016-05-18
## 7      4050       78       Piskwamish  51.6580  -80.5679        0        NA 2017-05-17
## 8      1134      280        Longridge  51.8230  -80.6911        0        NA 2014-07-16
## 9      2285      280        Longridge  51.8231  -80.6912        0        NA 2015-05-24
## 10     3097      280        Longridge  51.8244  -80.6909        0        NA 2016-05-17
## 11     4048      280    Halfway Point  51.8753  -80.7973        0        NA 2017-05-16
## 12     1135      285        Netitishi  51.2913  -80.1167        0        NA 2014-07-12
## 13     2289      285        Netitishi  51.2913  -80.1168        0        NA 2015-05-25
## 14     2286      349       Piskwamish  51.6578  -80.5676        0        NA 2015-05-24
## 15     1137      349       Piskwamish  51.6582  -80.5669        0        NA 2014-07-16
## 16     3813      528        NP mobile       NA        NA        1        NA 2015-07-06
## 17     4001      528 BurntPointAerial       NA        NA        1        NA 2016-07-19
## 18     4002      528   JamesBayAerial       NA        NA        1        NA 2016-08-09
```

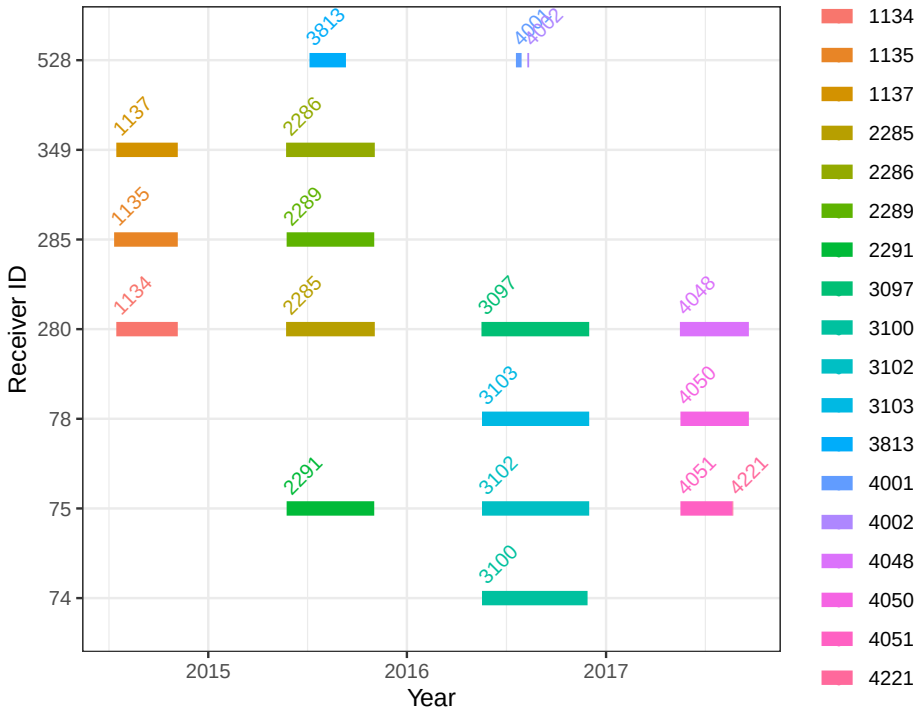
The number of receiver deployments in the metadata should correspond with the number of field deployments.

Looking at the `isMobile` column for the four receiver deployments that are missing latitude and longitude information, it is evident that these are mobile receivers that do not have a fixed position (ie. they have a value of 1). Because they are mobile, coordinates of the deployment aren't expected, and in this case will remain `NA`. Receiver deployment coordinates for mobile receivers, when present, are meant to represent the starting point for the deployment.

4.4.3 Timing of project receiver deployments

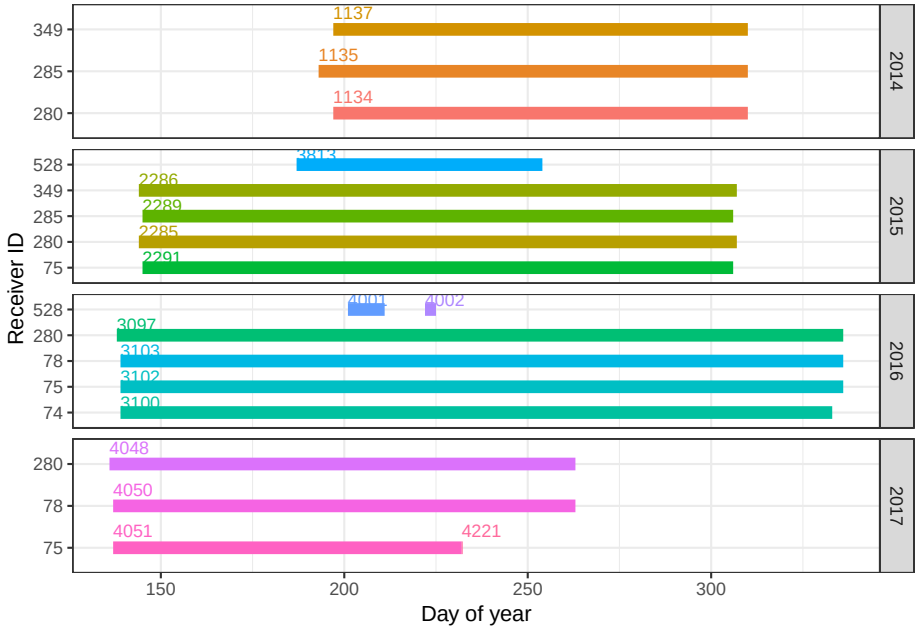
The timing of deployments can be displayed graphically; horizontal line(s) in the following plot show the time span for each receiver (`deviceID`) deployment registered to the James Bay Shorebird Project. Note that for the two receivers without deployment end dates, the code assigns an arbitrary end date based on the maximum end date of the other receivers plus one month - without this fix, deployments without end dates do not get displayed. Different deployments of the same receiver should not overlap in time:

```
# put data in long format to simplify plotting (or use geom_segment)

df.projRecvs.long <- df.projRecvs %>%
  select(deviceID, deployID, tsStart, tsEnd) %>%
  gather(when, ts, c(tsStart, tsEnd)) %>%
  # fake end date:
  mutate(ts = if_else(is.na(ts), max(ts, na.rm = TRUE) + duration(1, "month"), ts))

ggplot(data = df.projRecvs.long,
       aes(x = ts, y = as.factor(deviceID), colour = as.factor(deployID))) +
  theme(legend.position = "none") +
  geom_line(lwd = 3) +
  # instead, centre to the right
  geom_text(data = filter(df.projRecvs.long, when == "tsStart"),
            aes(label = deployID), hjust = "left", nudge_y = 0.2, size = 3, angle = 45)
  theme_bw() +
  labs(x = "Year", y = "Receiver ID")
```



If you want more detail for a given year (or all years) you can either subset and re-plot, or use the day of year on the x-axis, and `facet_wrap()` by year.

```
ggplot(data = df.projRecvs.long,
       aes(x = yday(ts), y = as.factor(deviceID), colour = as.factor(deployID))) +
  theme_bw() +
  theme(legend.position = "none") +
  geom_line(lwd = 3) +
  # centre labels to the left
  geom_text(data = filter(df.projRecvs.long, when == "tsStart"),
            aes(label = deployID), hjust = "left", nudge_y = 0.4, size = 3) +
  labs(x = "Day of year", y = "Receiver ID") +
  facet_grid(year(ts) ~ ., scales = "free")
```



4.4.4 Location of receiver deployments

Maps provide better spatial context than simple plots; the following steps plot the location of Motus receivers on a map of North America, with receivers deployed by the sample project displayed in red.

a. Load all receiver metadata

```
df.recvDeps <- tbl.recvDeps %>%
  collect() %>%
  as.data.frame() %>%
  mutate(tsStart = as_datetime(tsStart, tz = "UTC", origin = "1970-01-01"),
         tsEnd = as_datetime(tsEnd, tz = "UTC", origin = "1970-01-01"))
```

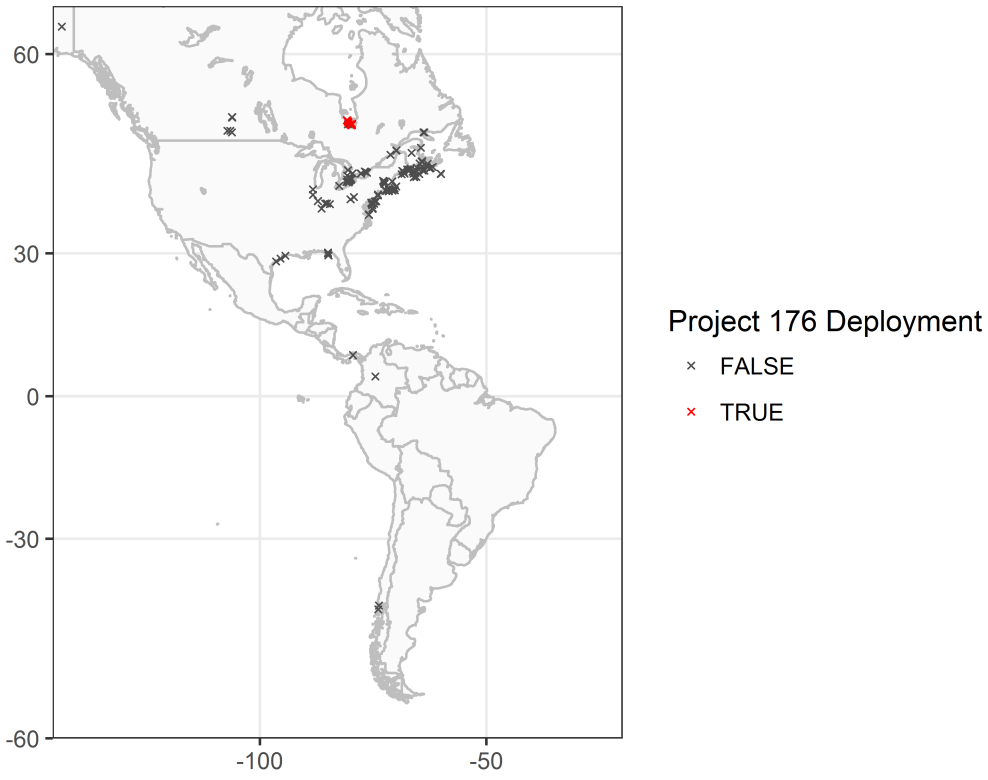
b. Load base map files

```
na.lakes <- map_data(map = "lakes")
na.lakes <- mutate(na.lakes, long = long - 360)

# Include all of the Americas to begin
na.map <- map_data(map = "world2")
na.map <- filter(na.map,
                 region %in% c("Canada", "USA", "Mexico", "lakes", "Belize",
                               "Costa Rica", "Panama", "Guatemala", "Honduras",
                               "Nicaragua", "El Salvador", "Colombia", "Venezuela",
                               "Ecuador", "Peru", "Brazil", "Guyana","Suriname",
```

```
                                "Bolivia", "French Guiana", "Jamaica", "Cuba",
                                "Haiti", "Dominican Republic", "The Bahamas",
                                "Turks and Caicos Islands", "Puerto Rico",
                                "British Virgin Islands", "Montserrat", "Dominica",
                                "Saint Lucia", "Barbados", "Grenada",
                                "Trinidad and Tobago", "Chile", "Argentina",
                                "Uruguay", "Paraguay")) %>%
  mutate(long = long - 360)
```

c. Map the location of receivers in the Americas Map showing the location of network-wide receivers (dark grey 'x') and receivers deployed by the James Bay Shorebird Project (project 176; red 'x').

```
# set map limits using detection locations;
# ensure they include the deployment locations
xmin <- min(df.recvDeps$longitude, na.rm = TRUE) - 2
xmax <- -20 # restrict to the Americas (excluding a few points in Europe)
ymin <- -60 #min(df.recvDeps$longitude, na.rm = TRUE) - 2
ymax <- max(df.recvDeps$latitude, na.rm = TRUE) + 2

# map
ggplot(data = na.lakes, aes(x = long, y = lat)) +
  theme_bw() +
  geom_polygon(data = na.map, aes(long, lat, group = group),
               colour = "grey", fill = "grey98") +
  geom_polygon(aes(group = group), colour = "grey", fill = "white") +
  coord_map(projection = "mercator", xlim = c(xmin, xmax), ylim = c(ymin, ymax)) +
  labs(x = "", y = "") +
  geom_point(data = df.recvDeps,
             aes(longitude, latitude, colour = as.logical(projectID == 176)),
             size = 0.8, shape = 4) +
  scale_colour_manual(values = c("grey30", "red"), name = "Project 176 Deployment")
```

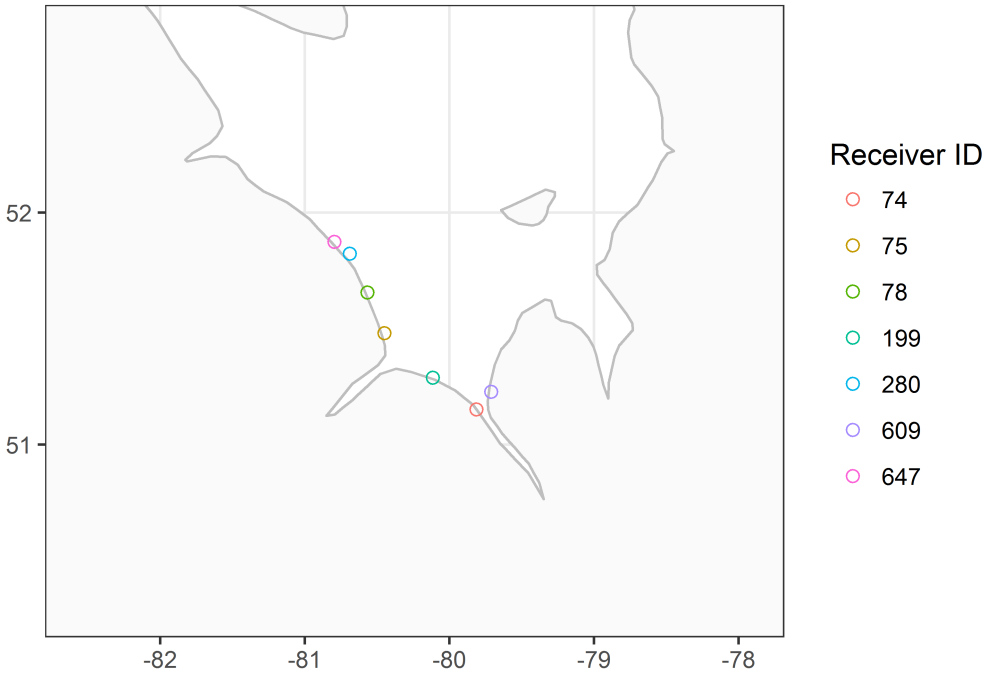


d. Map the location of project specific receivers only

Map of project-specific receivers, created by setting the x-axis (longitude) and y-axis (latitude) map limits using the 'df.projRecvs' dataframe created above. Deployments are restricted to those that were active at in 2016.

```
# set map limits using detection locations;
# ensure they include the deployment locations
xmin <- min(df.projRecvs$longitude, na.rm = TRUE) - 2
xmax <- max(df.projRecvs$longitude, na.rm = TRUE) + 2
ymin <- min(df.projRecvs$latitude, na.rm = TRUE) - 1
ymax <- max(df.projRecvs$latitude, na.rm = TRUE) + 1

# map
ggplot(data = na.lakes, aes(x = long, y = lat))+
  theme_bw() +
```

```
  geom_polygon(data = na.map, aes(long, lat, group = group),
               colour = "grey", fill = "grey98") +
  geom_polygon(aes(group = group), colour = "grey", fill = "white") +
  coord_map(projection = "mercator", xlim = c(xmin, xmax), ylim = c(ymin, ymax)) +
  labs(x = "", y = "") +
  geom_point(data = filter(df.projRecvs,
                           year(tsStart) == 2016,
                           !is.na(latitude)),  # remove mobile receivers
             aes(longitude, latitude, colour = as.factor(deviceID)), size = 2, shape = 1)+
  scale_colour_discrete(name = "Receiver ID")
```



4.4.5 Completeness and accuracy of receiver metadata

Motus users will be concerned primarily with the completeness of metadata for receiver deployments with detection(s) of their tags, because these can directly influence the interpretation of those detections. For example, missing deployment latitude or longitude will result in an unknown location for the tag detection, and missing information on antenna type and/or orientation can impede the

estimation of flight or departure orientation.

In many cases, however, metadata for receiver deployments *without* tag detections can also be useful, for example to estimate probability of detecting an animal that passes within range of a station.

In this section, the focus is on metadata for receivers registered to a particular project. Depending on your interests, these summaries can be applied to a larger group of receivers, e.g., all receivers with detections or all receivers within certain geographic limits (with or without detections).

a. Load receiver and antenna metadata

```
# antenna metadata for ALL Motus antenna deployments;
# to simplify, keep only the variables of interest.
tbl.antDeps <- tbl(sql.motus, "antDeps")

df.antDeps <- tbl.antDeps %>%
  select(deployID, port, antennaType, bearing, heightMeters) %>%
  collect() %>%
  as.data.frame()

# receiver deployments; select variables of interest
df.recvDeps <- df.recvDeps %>%
  select(deployID, receiverType, deviceID, name, latitude, longitude,
         isMobile, tsStart, tsEnd, projectID, elevation)

df.stationDeps <- left_join(df.recvDeps, df.antDeps, by = "deployID")
```

Subset these to receivers registered to a project:

```
df.stationDeps <- filter(df.stationDeps, projectID == proj.num)
```

b. Look at range of metadata values

Use `summary()` to get a general idea of the distribution of the variables in the data.

```
summary(df.stationDeps)
```

```
##     deployID    receiverType          deviceID         name              latitude
##  Min.   :1134   Length:55          Min.   : 74.0   Length:55          Min.   :51.15
##  1st Qu.:2286   Class :character   1st Qu.: 75.0   Class :character   1st Qu.:51.48
##  Median :3100   Mode  :character   Median :280.0   Mode  :character   Median :51.66
##  Mean   :2919                      Mean   :201.3                      Mean   :51.57
##  3rd Qu.:4048                      3rd Qu.:285.0                      3rd Qu.:51.70
##  Max.   :4221                      Max.   :528.0                      Max.   :51.88
##                                                                       NA's   :3
##      tsEnd                         projectID     elevation          port           ant
##  Min.   :2014-11-06 00:00:00   Min.   :176   Min.   :-7.000   Length:55          Len
```

```
##  1st Qu.:2015-11-02 00:00:00   1st Qu.:176   1st Qu.:-7.000   Class :character   Class :charac
##  Median :2016-10-05 08:15:00   Median :176   Median :-7.000   Mode  :character   Mode  :charac
##  Mean   :2016-05-04 00:08:20   Mean   :176   Mean   :-5.714
##  3rd Qu.:2016-12-01 00:00:00   3rd Qu.:176   3rd Qu.:-4.000
##  Max.   :2017-08-20 23:30:00   Max.   :176   Max.   :-4.000
##  NA's   :7                                   NA's   :48
```

There are the 4 deployments with missing latitude and longitude associated with the four deployments of mobile receivers that we saw earlier.

Elevation is missing from 74 of 91 records, but elevation is not a required field, and can be estimated from other sources, or directly in R (for example, see https://stackoverflow.com/questions/8973695/conversion-for-latitude-longitude-to-altitude-in-r).

Antenna bearing is missing from 18 of 91 records, and height of the antenna(s) is missing for 4 of 91 records. Subset the records with missing antenna bearing to see if these can be fixed:

```
df.stationDeps %>%
  filter(is.na(bearing)) %>%
  select(-elevation, -deviceID, -tsEnd)
```

```
##    deployID receiverType             name latitude longitude isMobile             tsStart proj
## 1      3097  SENSORGNOME        Longridge  51.8244  -80.6909        0 2016-05-17 00:00:00
## 2      3100  SENSORGNOME       Washkaugou  51.1540  -79.8144        0 2016-05-18 00:00:00
## 3      3102  SENSORGNOME      North Bluff  51.4839  -80.4501        0 2016-05-18 00:00:00
## 4      3103  SENSORGNOME       Piskwamish  51.6579  -80.5678        0 2016-05-18 00:00:00
## 5      3813  LOTEKSRX800        NP mobile       NA        NA        1 2015-07-06 00:00:00
## 6      4001  LOTEKSRX800 BurntPointAerial       NA        NA        1 2016-07-19 08:00:00
## 7      4002  LOTEKSRX800   JamesBayAerial       NA        NA        1 2016-08-09 07:15:00
## 8      4048  SENSORGNOME    Halfway Point  51.8753  -80.7973        0 2017-05-16 15:55:00
## 9      4050  SENSORGNOME       Piskwamish  51.6580  -80.5679        0 2017-05-17 15:19:00
## 10     4051  SENSORGNOME      North Bluff  51.4839  -80.4501        0 2017-05-17 15:00:00
## 11     4221  SENSORGNOME      North Bluff  51.4839  -80.4501        0 2017-08-20 23:30:00
```

Receiver deployments with missing antenna bearing(s) are restricted to deployments of omni-directional antennas or mobile receivers, and so the missing values make sense. These records also show that the four records with missing antenna height are also associated with the four mobile receivers, and so again the missing values make sense and do not need to be fixed.

Remember that any missing metadata needs to be corrected online. Metadata corrected online will automatically be corrected in your detection files. If you have already downloaded your detection data, you can update the existing file to include new metadata and detections (see sections 3.9, 3.7).

In the next chapter we will examine our data for false positives, and remove detections of ambiguous tags.

Chapter 5

Data Cleaning

This chapter was contributed by Tara L. Crewe, Zoe Crysler, and Philip Taylor. Revisions by Catherine Jardine, Steffi LaZerte and Denis Lepage

There are three sources of 'error' that can result in tag detections appearing in your database that are incorrect.

First, random radio noise ('static') can be detected and interpreted to be the transmission of a tag. These are called 'false positives'.

Second, despite our best efforts to avoid it, duplicate tags are sometimes transmitting in the network at the same time. When two tags are deployed at the same time that have the same ID code, burst interval, and nominal transmit frequency, it results in situations where the detections may belong to either tag. If that happens, we must rely on contextual information to separate them (if we can). We term these 'Ambiguous tags'.

Third, a tag can appear to be present when two tags are transmitting at the same time that by chance produce a signal that looks like a third tag that is not in fact present. Such tags are most common at roosting sites or breeding colonies, where many tags are transmitting simultaneously. We term these 'Aliased tags'. We do not deal explicitly with Aliased tags in this chapter; we are working on a way to globally identify them and eliminate them from the data. We mention them here because you may encounter situations with what appear to be highly plausible detections that don't make biological sense. Please contact us if you think you have some of these Aliased tag detections in your database.

The *goal of this chapter* is to provide you with the tools you need to check your data for false detections, and remove them from your data. We do so by providing example workflows that deal with 'false positives' and 'ambiguous tags' in the following steps:

1) **Run a preliminary filter to remove all detections with `runLen` of**

3 or less, and detections with intermediate `runLens` made during periods of high noise/activity.
A run is a group of consecutive detections of a tag detected on a single antenna at a single receiver. In general, a detection with a run length of 2 or 3 (i.e., 2 or 3 bursts) has a high probability of being a false positive detection. With the exception of a few 'quiet' stations with little noise, we generally recommend that you filter out all detections with a run length of 3 or less. However, because you will likely lose some true detections in the process, we also recommend that after a full analysis of your data, you return to these detections and examine them individually, to determine (usually contextually) if they can be considered real. Stations that are particularly noisy may also have false detections with longer `runLens`.

2) **Determine how many of your tag detections may be ambiguous detections**

3) **Provide a workflow for examining individual tags, and determine if runs in those tags are errors**

4) **Filter errors from your data**

5.1 Load required packages

Follow the instructions in Chapter 2 to install the following packages before loading, if they are not already installed.

```
Sys.setenv(tz = "UTC")

library(motus)
library(tidyverse)
library(lubridate)
```

5.2 Load detections data

Recall from Chapter 3 that when accessing the sample database, you will need to input `motus.sample` in the R console as both username and password when prompted by the `tagme()` user authentication process. This section assumes you have already completed the initial sample data download.

```
proj.num <- 176
sql.motus <- tagme(proj.num, update = TRUE, dir = "./data/")
```

5.3 Assess tag detections

First, determine which project tags have detections. There are several reasons why deployed tags might not be detected, including:

1) The tag was not properly activated on deployment. To avoid this, always check that a tag is active using a hand-held receiver before attaching the tag to your study animal and releasing it.

2) An animal with a properly activated tag might not have passed within range of a receiving station. Study designs that incorporate strategic placement of receivers to meet project goals can improve the probability of a tag being detected.

3) Missing or incorrect tag deployment metadata in the Motus database can result in the data processing algorithm not 'looking' for your tag at the time the tag was deployed, or at all. Please ensure your tag metadata are entered correctly.

Before going further, **please check whether any of your tags were deployed more than once**, as described in section 4.3.4. If so, you will need to use `tagDeployID` or a combination of `motusTagID` and `tagDeployID` to uniquely define detections associated with a tag deployment (either will do, but combining the two fields will let you know which tagID is associated with each deployment).

In the sample data, all tags were deployed only once, and so we use the `motusTagID` as a unique identifier for a tag deployment in all R code throughout the book.

Using the `count()` function we can see that there are detections for 18 tags deployed by the sample project.

```
tbl(sql.motus, "alltags") %>%
  filter(tagProjID == proj.num) %>% # subset to include only tags registered to projec
  count(motusTagID) %>%
  as.data.frame()
```

```
##    motusTagID     n
## 1       16011   127
## 2       16035   456
## 3       16036   118
## 4       16037  1353
## 5       16038   162
## 6       16039  1126
## 7       16044   305
## 8       16047   839
## 9       16048    98
## 10      16052   159
## 11      17357   289
```

```
## 12      19129  1288
## 13      22867  5767
## 14      22897 34796
## 15      22902  2923
## 16      22905 26010
## 17      23316  5734
## 18      23319 22759
```

If we break down these counts by run length using the following code we further see that many have run lengths of 3 (the `run 3` column).

```
tbl(sql.motus, "alltags") %>%
  filter(tagProjID == proj.num) %>% # subset to include only tags registered to project
  mutate(rl.gt.3 = if_else(runLen == 3, "run 3", "run > 3")) %>%
  count(motusTagID, rl.gt.3) %>%
  collect() %>%
  spread(key = rl.gt.3, value = n)
```

```
## # A tibble: 18 x 3
## # Groups:   motusTagID [18]
##    motusTagID `run > 3` `run 3`
##         <int>     <int>   <int>
##  1      16011       118       9
##  2      16035       441      15
##  3      16036       115       3
##  4      16037      1335      18
##  5      16038       156       6
##  6      16039      1117       9
##  7      16044       287      18
##  8      16047       758      81
##  9      16048        83      15
## 10      16052       144      15
## 11      17357       286       3
## 12      19129      1189      99
## 13      22867      5572     195
## 14      22897     34373     423
## 15      22902      2854      69
## 16      22905     25782     228
## 17      23316      5551     183
## 18      23319     22567     192
```

Although some of these may be valid detections, we have found it simpler to just remove them from our analysis, and possibly revisit them at a later stage. In the next few sections we will explore a series of filters that do just this.

5.4 Preliminary filtering

We can perform some preliminary filtering based on run length (`runLen`). As runs are composed of sequences of hits, the longer the run the more confident we can be that it represents a true detection. However, local conditions at an individual receiver may vary in their exposure to background radio noise. Sites with relatively more background noise may be more prone to reporting false detections. Therefore the standard motus filter relies both on the length of the run, and the amount of radio activity at a given site. This value is stored as a field called `motusFilter` in the `runs` table. An additional table (`activity`) allows users to calculate a filter based on different criteria if they want.

5.4.1 Understanding the `motusFilter`

Starting in July 2019, data downloaded from motus with the `tagme()` function will include a standard filter value that can be used to identify detections that we believe have a higher probability of being false hits. The various outputs on the motus web site are pre-filtered, but the R package provides access to all detections, allowing users more control over which detections to keep or omit. The `motusFilter` field in the `runs` table represents the probability that the run is a true detection. Currently the `motusFilter` contains just two values `0` or `1`. Runs with a `motusFilter` of `0` have a low probability of being true detections.

5.4.2 How the `motusFilter` is generated

The `motusFilter` is based the number of detections of different run lengths at a given site, across all motus projects. Periods with lots of radio interference will typically generate a high number of very short runs that are in reality spurious data. In the presence of a high ratio of runs with length = 2 at a given time, we consider that site as 'noisy' and increase the minimum threshold for run lengths that we consider to be valid detections.

In general, a detection with a run length of 2 or 3 (i.e., 2 or 3 bursts) has a relatively high probability of being a false positive detection. Therefore, runs with a length 3 or less are conservatively assigned a `motusFilter` of 0. For runs greater than length 3, data in the `activity` table provides a method of assessing the relative amount of background noise at a site which can be used to assess the probability of these runs being true detections. The `activity` table provides the number of runs of various lengths for each hour (called `hourBin`) within each batch and for each antenna. A high number of detections within an hour (e.g., >= 100) with a high proportion of short runs (e.g., length 2 >= 85%) are indicative of a noisy environment, more likely to generate false positives. For those periods, we treat the intermediate run lengths ($3 <$ `runLen` < 5) as invalid

(`motusFilter` = 0). Intermediate run lengths are otherwise considered valid (`motusFilter` = 1).

The `motusFilter` values in the `runs` table have been calculated on the basis of the above criteria, which were determined through some empirical examination of data. If you are working with a dataset downloaded through `tagme()` prior to July 2019 it will not include those values. In those cases, you will either need to download a new copy of the entire dataset for your project or receiver, or to use the `filterByActivity()` function described below to calculate the missing values.

To omit runs identified as dubious by `motusFilter` we can use an anti-join.

```
# Number of rows with runs 3 hits or less
filter(tbl(sql.motus, "alltags"), runLen <= 3) %>%
  collect() %>%
  nrow()
```

```
## [1] 4511
```

```
# Identify runs to remove
to_remove <- tbl(sql.motus, "runs") %>%
  select(runID, motusFilter) %>%
  filter(motusFilter == 0)

# Use anti-join to remove those runs from the alltags
tbl_filtered <- anti_join(tbl(sql.motus, "alltags"), to_remove, by = "runID")

# Number of rows with runs 3 hits or less after being filtered
filter(tbl_filtered, runLen <= 3) %>%
  collect() %>%
  nrow()
```

```
## [1] 0
```

5.4.3 Custom filters with the `filterByActivity()` function

The `motusFilter` is one method of determining false detections, but motus users are encouraged to explore alternative filter parameters. The `motus` R-package includes a `filterByActivity()` function that allows users to specify custom parameters used to identify false positives based on the `activity` table. Users can return either just the "true" positives (`return = "good"`), just the "false" positives (`return = "bad"`) or all hits (`return = "all"`) but with a new column, "probability", which reflects either 0 (expected false positive) or 1 (expected true positive).

For example, the following code, adds a `probability` column to the sample project data, which is identical to the `motusFilter` column (i.e., by default

`filterByActivity()` uses the same conditions).

Note that this function requires the SQLite database connection (not a flat data frame), but returns a data.frame of the `alltags` view (not a SQLite database connection).

```
tbl_motusFilter <- filterByActivity(sql.motus, return = "all")
```

Users can adjust these parameters to be less strict (i.e., exclude fewer detections). This example excludes all runs of length 2 or less and will exclude any runs less than length 4 from `hourBin`s which have more than 500 runs and where at least 95% of those runs have a run length of 2.

```
tbl_relaxed <- filterByActivity(sql.motus, minLen = 2, maxLen = 4,
                                maxRuns = 500, ratio = 0.95, return = "all")
```

These parameters can also be more strict (i.e., exclude more detections). This next example excludes all runs of length 4 or less and will exclude any runs less than length 10 from `hourBin`s which have more than 50 runs and where at least 75% of those runs have a run length of 2.

```
tbl_strict <- filterByActivity(sql.motus, minLen = 4, maxLen = 10,
                               maxRuns = 50, ratio = 0.75, return = "all")
```

Note that the filters may exclude some true detections in the process. Therefore, we recommend that after a full analysis of your data, you return to these detections and examine them individually, to determine (usually contextually) if they can be considered real.

5.5 Preparing the data

When accessing the `alltags` view, we filter the data and remove some unnecessary variables to reduce the overall size of the data set and make it easier to work with. **This is particularly important for large, unwieldy projects**; details on how to view the variables in a `tbl`, and how to filter and subset prior to collecting data into a dataframe can be found in Chapter 3.5.8.

5.5.1 Label detections by probability

First we will use the `filterByActivity()` function to label dubious detections. This returns all the data in the `alltags` view with a new `probability` column.

```
tbl.alltags <- filterByActivity(sql.motus, return = "all")
```

Alternatively we can change the default, so the `filterByActivity()` function uses the `alltagsGPS` view. **However,** on very large databases this could be slow.

```
tbl.alltags.gps <- filterByActivity(sql.motus, return = "all", view = "alltagsGPS")
```

5.5.2 Clean up

Let's filter to the 'good' detections. We will filter to `1` for detections to keep and `0` for dubious detections. Then we'll use the `collect()` and `as.data.frame()` functions to transform the dataframe into a 'flat' (i.e. non SQLite) file, and then transform all time stamp variables from seconds since January 1 1970 to datetime (POSIXct) format.

```
df.alltags.sub <- tbl.alltags %>%
  filter(probability == 1) %>%
  collect() %>%
  as.data.frame() %>%
  mutate(ts = as_datetime(ts),  # work with dates AFTER transforming to flat file
         tagDeployStart = as_datetime(tagDeployStart),
         tagDeployEnd = as_datetime(tagDeployEnd))
```

Let us also save the excluded detections for later analysis.

```
df.block.0 <- filter(tbl.alltags, probability == 0) %>%
  select(motusTagID, runID) %>%
  distinct() %>%
  collect() %>%
  data.frame()
```

5.5.3 Add GPS data

The `filterByActivity()` function can use the `alltagsGPS` view, but this may be slow. A way around this speed problem is to use the `getGPS()` function to retrieve the GPS values of a data subset (`data`) after the data has been filtered.

Note that in this example, the sample dataset doesn't have GPS data

```
# Retrieve GPS data for each hitID
gps_index <- getGPS(sql.motus, data = df.alltags.sub, by = "closest")

# Merge GPS points in with our data
df.alltags.sub <- left_join(df.alltags.sub, gps_index, by = "hitID")
```

We match GPS locations to `hitID`s according to one of several different time values, specified by the `by` argument. This can be the `closest` location in time, the `daily` median location, or the median location within `X` minutes of a `hitID` (here, `X` can be any number greater than zero and represents the size of the time block in minutes over which to calculate a median location). If using

the `closest` option, you can also specify a `cutoff` which will only match GPS records which are within `cutoff = X` minutes of the hit. This way you can avoid having situations where the 'closest' GPS record is actually days away (see also Chapter 3.5.7).

We then create receiver latitude and longitude variables (`recvLat`, `recvLon`, `recvAlt`) based on the coordinates recorded by the receiver GPS (`gpsLat`, `gpsLon`, `gpsAlt`), and where those are not available, infilled with coordinates from the receiver deployment metadata (`recvDeployLat`, `recvDeployLon`, `recvDeployAlt`). Missing GPS coordinates may appear as `NA` if they are missing, or as `0` or `999` if there was a problem with the unit recording.

Finally, we create 'receiver names' by adding rounded `recvLat` and `recvLon` to the `recvDeployName` for those receivers in the database that do not have these values filled in. As more users explore (and fix!) their metadata, these missing values should begin to disappear. We'll fix this here as sometimes if there is missing metadata (ie. a missing receiver deployment spanning some of your detections), you will get `NA`s which can lead to problems later on.

```
df.alltags.sub <- df.alltags.sub %>%
  mutate(recvLat = if_else((is.na(gpsLat)|gpsLat == 0|gpsLat == 999),
                           recvDeployLat, gpsLat),
         recvLon = if_else((is.na(gpsLon)|gpsLon == 0|gpsLon == 999),
                           recvDeployLon, gpsLon),
         recvAlt = if_else(is.na(gpsAlt), recvDeployAlt, gpsAlt)) %>%
  select(-noise, -slop, -burstSlop, -done, -bootnum, -mfgID,
         -codeSet, -mfg, -nomFreq, -markerNumber, -markerType,
         -tagDepComments, -fullID, -deviceID, -recvDeployLat,
         -recvDeployLon, -recvDeployAlt, -speciesGroup, -gpsLat,
         -gpsLon, -recvAlt, -recvSiteName) %>%
  mutate(recvLat = plyr::round_any(recvLat, 0.05),
         recvLon = plyr::round_any(recvLon, 0.05),
         recvDeployName = if_else(is.na(recvDeployName),
                                  paste(recvLat, recvLon, sep=":"),
                                  recvDeployName))

# Note that in the select statement, you can just select the variables you need
# e.g.: select(runID, ts, sig, freqsd, motusTagID, ambigID, runLen, tagProjID,
#              tagDeployStart, tagDeployEnd, etc.)
# As opposed to those you don't need (-done, -bootnum, etc.)
```

Now we have a nice clean data frame!

5.6 Preliminary data checks

Prior to more specific filtering the data, we will perform a few summaries and plots of the data.

5.6.1 Summarize tag detections

An initial view of the data is best achieved by plotting. We will show you later how to plot detections on a map, but we prefer a simpler approach first; plotting detections through time by both latitude and longitude. First however, we should simplify the data. If we don't, we risk trying to plot thousands or millions of points on a plot (which can take a long time). We'll do this by creating a little function here, since we will use this operation again in future steps.

Note that we will need to remove about 150 detections, because there is no geographic data associated with the receiver metadata, and so no way to determine the location of those detections. Do a simple check to see if these receivers belong to you, and if so, please **fix the metadata online**!

For example, here we can see which receivers are missing (`is.na(recvLat)`).

```
df.alltags.sub %>%
  filter(is.na(recvLat)) %>%
  select(recvLat, recvLon, recvDeployName, recvDeployID, recv,
         recvProjID, recvProjName) %>%
  distinct()
```

```
##   recvLat recvLon recvDeployName recvDeployID           recv recvProjID recvProjName
## 1      NA      NA      NP mobile         3813       Lotek-280        176   SampleData
## 2      NA      NA          NA:NA           NA SG-1415BBBK0382         NA         <NA>
## 3      NA      NA          NA:NA           NA SG-2814BBBK0547         NA         <NA>
```

Simplify the data for plotting

We can simplify the data by summarizing by the `runID`. If you want to summarize at a finer/coarser scale, you can also create other groups. The simplest alternative is a rounded timestamp variable; for example by using `mutate(ts.h = plyr::round_any(ts, 3600)`. Other options are to just use date (e.g `date = as_date(ts)`).

Here is an advanced example creating a function so we can re-use this simplification later.

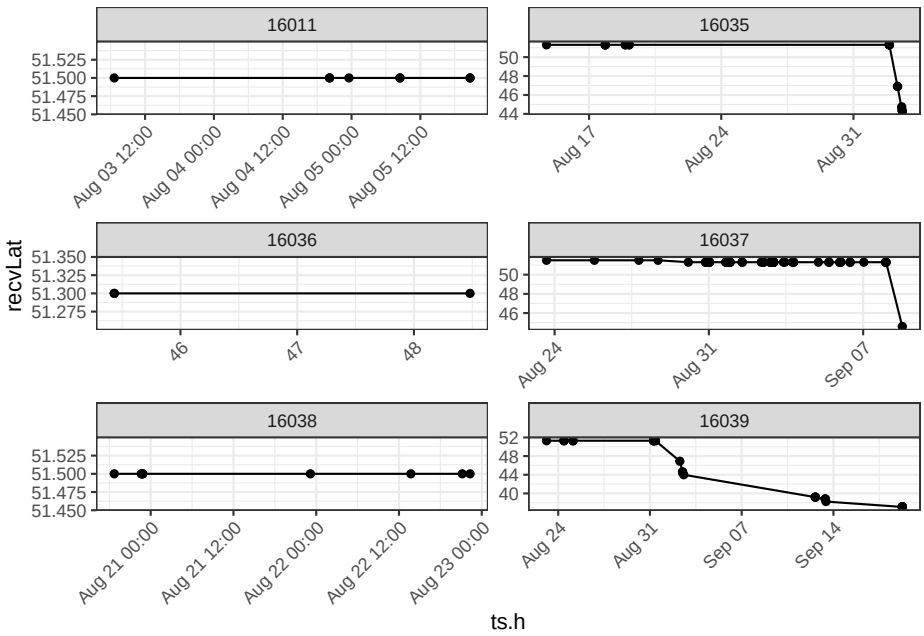
```
fun.getpath <- function(df) {
  df %>%
    filter(tagProjID == proj.num, # keep only tags registered to the sample project
           !is.na(recvLat) | !(recvLat == 0)) %>% # drops data without lon/lat
    group_by(motusTagID, runID, recvDeployName, ambigID,
```

```
            tagDepLon, tagDepLat, recvLat, recvLon) %>%
    #summarizing by runID to get max run length and mean time stamp:
    summarize(max.runLen = max(runLen), ts.h = mean(ts)) %>%
    arrange(motusTagID, ts.h)
} # end of function

df.alltags.path <- fun.getpath(df.alltags.sub)
```

```
## `summarise()` regrouping output by 'motusTagID', 'runID', 'recvDeployName', 'ambigI
```

We would initially plot a subset of tags by either latitude or longitude, to get an overview of where there might be issues. Here, to simplify the example, we plot only six tags. We avoid examining the ambiguous tags for now.

```
ggplot(data = filter(df.alltags.path,
                     motusTagID %in% c(16011, 16035, 16036, 16037, 16038, 16039)),
       aes(x = ts.h, y = recvLat)) +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 45, vjust = 1, hjust = 1)) +
  geom_point() +
  geom_path() +
  facet_wrap(~ motusTagID, scales = "free", ncol = 2)
```



Although there don't seem to be any immediate problems, let's take a look at the tags showing up around 44 degrees during September. Let's examine these tags in more detail by examining the runs in the data frame that are associated

with detections in September.

```
df.alltags.sub %>%
  filter(month(ts) %in% c(9),
         motusTagID %in% c(16035, 16037, 16039),
         recvLat < 44) %>%
  group_by(recvDeployName, month(ts), runLen) %>%
  summarize(n = length(ts),
            n.tags = length(unique(motusTagID))) %>%
  arrange(runLen)
```

```
## `summarise()` regrouping output by 'recvDeployName', 'month(ts)' (override with `.groups` argu
```

```
## # A tibble: 13 x 5
## # Groups:   recvDeployName, month(ts) [5]
##    recvDeployName        `month(ts)` runLen     n n.tags
##    <chr>                       <dbl>  <int> <int>  <int>
##  1 Assateague State Park           9      6     6      1
##  2 FINWR                           9      6     6      1
##  3 FINWR                           9     10    10      1
##  4 FINWR                           9     20    20      1
##  5 Prime Hook                      9     23    23      1
##  6 BULL                            9     24    24      1
##  7 Prime Hook                      9     27    27      1
##  8 Bombay Hook                     9     32    32      1
##  9 Prime Hook                      9     32    32      1
## 10 Bombay Hook                     9     36    36      1
## 11 BULL                            9     38    38      1
## 12 Bombay Hook                     9     53    53      1
## 13 FINWR                           9     73    73      1
```

Since we have already filtered dubious detections, these remaining ones don't seem immediately unreliable (all with runs of at least 6). If you are interested, you can re-run the code above, but on the full data frame (`tbl(sql.motus, "alltags")`) containing run lengths of 2 or 3. You will see that there are likely false positive detections at these sites, that were already eliminated by filtering. These additional detections provide further evidence that these sites experienced some radio noise during these particular months, resulting in some false positive detections.

You may also be interested more generally in exploring which data have only short run lengths. For example, the following code shows the maximum run length at all sites by month (for those runs which haven't been removed by filtering).

```
df.alltags.sub %>%
  mutate(month = month(ts)) %>%
  group_by(recvDeployName, month) %>%
```

```
  summarize(max.rl = max(runLen)) %>%
  spread(key = month, value = max.rl)
```

```
## `summarise()` regrouping output by 'recvDeployName' (override with `.groups` argumen
```

```
## # A tibble: 42 x 6
## # Groups:   recvDeployName [42]
##    recvDeployName          `4`   `5`   `8`   `9`  `10`
##    <chr>                 <int> <int> <int> <int> <int>
##  1 Assateague State Park    NA    NA    NA     6    NA
##  2 BennettMeadow            NA    NA    NA    NA    11
##  3 BISE                     NA    NA    NA    NA     6
##  4 Bombay Hook              NA    NA    NA    53    NA
##  5 Brier2                   NA    NA    NA    29    NA
##  6 BSC HQ                   NA    21    NA    NA    NA
##  7 BULL                     NA    NA    NA    38     5
##  8 Comeau (Marshalltown)    NA    NA    NA     4    NA
##  9 CONY                     NA    NA    NA    NA     7
## 10 D'Estimauville           NA    NA    NA    40    NA
## # ... with 32 more rows
```

Alternatively, you can produce a list of sites where the maximum run length of detections was never greater than (say) 4, which may sometimes (but not always!) indicate they are simply false detections.

```
df.alltags.sub %>%
  mutate(month = month(ts)) %>%
  group_by(recvDeployName, month) %>%
  summarize(max.rl = max(runLen)) %>%
  filter(max.rl < 5) %>%
  spread(key = month, value = max.rl)
```

```
## `summarise()` regrouping output by 'recvDeployName' (override with `.groups` argumen
```

```
## # A tibble: 3 x 4
## # Groups:   recvDeployName [3]
##   recvDeployName          `4`   `9`  `10`
##   <chr>                 <int> <int> <int>
## 1 Comeau (Marshalltown)    NA     4    NA
## 2 Old Cut                   4    NA    NA
## 3 TRUS                     NA    NA     4
```

It is impossible to go through every possible issue that you may encounter here. Users are strongly encouraged to explore their data fully, and make reasoned decisions on which detections are unlikely or indeterminate. Through the rest of this chapter we will show you how to collect these runs, and apply them to your data prior to analysis.

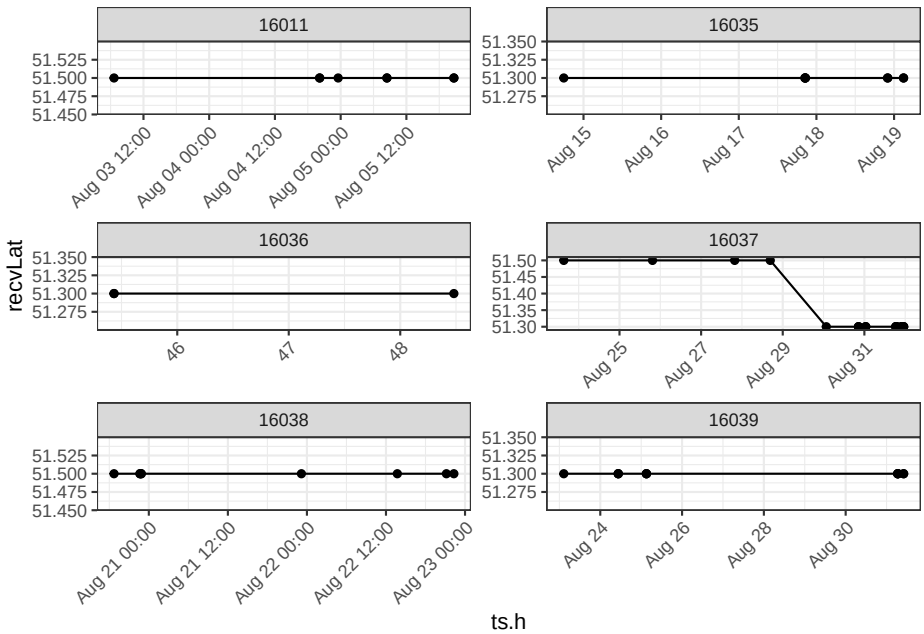
To start, if we decided that those detections in September were false positives, we could create a data frame that contains the `motusTagID`s and `runID`s for them.

We could then re-create the plot with the newly filtered data.

```
# Collect dubious detections
df.block.1 <- df.alltags.sub %>%
  filter(month(ts) %in% 9,
         motusTagID %in% c(16035, 16037, 16039)) %>%
  select(motusTagID, runID) %>%
  distinct()

# use the function we created earlier to make a new 'path' data frame for plotting
df.alltags.path <- fun.getpath(filter(df.alltags.sub,
                                      motusTagID %in% c(16011, 16035, 16036,
                                                        16037, 16038, 16039),
                                      !(runID %in% df.block.1$runID)))
```

```
## `summarise()` regrouping output by 'motusTagID', 'runID', 'recvDeployName', 'ambigID', 'tagDep
```

```
ggplot(data = df.alltags.path, aes(x = ts.h, y = recvLat)) +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 45, vjust = 1, hjust = 1)) +
  geom_point() +
  geom_path() +
  facet_wrap(~ motusTagID, scales = "free", ncol = 2)
```



The reader is encouraged to explore the rest of the tags within this group, to determine if there are additional false positives.

5.7 Examining ambiguous detections

Before we go further, we need to check to see if any tags have ambiguous detections. If there are, we will need to explore them, and create additional filters to remove detections from our database.

Are any of your tags associated with ambiguous detections?

The `clarify()` function in the `motus` R package provides a summary of ambiguities in the detections data. Each `ambigID` refers to a selection of detections that could belong to one or more (up to 6) `motusTagID`s, which are listed in the `id1` to `id6` columns:

```
clarify(sql.motus)
```

```
##   ambigID numHits   id1                                fullID1   id2
## 1     -56    5734 22867 SampleData#272.1:5.3@166.38(M.22867) 23316   SampleData#272
## 2    -106     279 17021        Selva#172:6.1@166.38(M.17021) 17357   SampleData#172
## 3    -114      86 22897 SampleData#303.1:5.3@166.38(M.22897) 24298      NEONICS#303
## 4    -134   22749 22905   SampleData#301:5.3@166.38(M.22905) 23319 SampleData#301.1
## 5    -171    2074 22778   RBrownAMWO#308:5.3@166.38(M.22778) 22902 SampleData#308.1
## 6    -337       4 10811        Niles#152:6.1@166.38(M.10811) 16011   SampleData#152
##   id6 fullID6 motusTagID tsStart tsEnd
## 1  NA    <NA>         NA      NA    NA
## 2  NA    <NA>         NA      NA    NA
## 3  NA    <NA>         NA      NA    NA
## 4  NA    <NA>         NA      NA    NA
## 5  NA    <NA>         NA      NA    NA
## 6  NA    <NA>         NA      NA    NA
```

We can see that there are six tags with ambiguous detections within this data set. Detections associated with five of the six `ambigID`s could belong to one of two tags, and detections associated with one `ambigID` (`-171`) could belong to one of three tags. The `fullID` fields list the project names associated with the duplicate tags (e.g., "SampleData", "Selva", "Niles"), along with features of the tags (manufacturer tag ID, burst, and transmit frequency).

Let's get a data frame of these, and do some plots to see where there may be issues.

```
df.ambigTags <- df.alltags.sub %>%
  select(ambigID, motusTagID) %>%
  filter(!is.na(ambigID)) %>%
  distinct()
```

Using our `getpath()` function, we'll create paths and then plot these detections. We'll add some information to the plot, showing where (in time) the tags are actually ambiguous. We can then inspect the overall plots (or portions of them) to determine if we can contextually unambiguously assign a detection of an ambiguous tag to a single deployment.

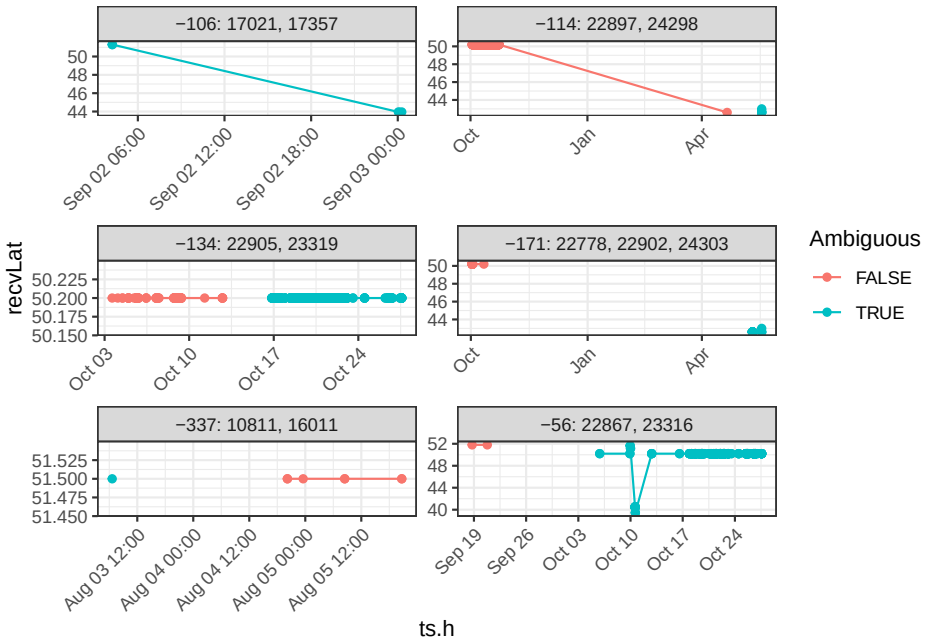
```
df.alltags.path <- fun.getpath(filter(df.alltags.sub,
                                      motusTagID %in% df.ambigTags$motusTagID,
                                      tagProjID == proj.num)) %>%
  # create a boolean variable for ambiguous detections:
  mutate(Ambiguous = !(is.na(ambigID)))
```

```
## `summarise()` regrouping output by 'motusTagID', 'runID', 'recvDeployName', 'ambigID', 'tagDep
```

```
# to put all ambiguous tags from the same project on the same plot together, we
# need to create a new 'ambig tag' variable we call 'newID.

ambigTags.2 <- filter(df.alltags.sub) %>%
  select(ambigID, motusTagID) %>%
  filter(!is.na(ambigID)) %>%
  distinct() %>%
  group_by(ambigID) %>%
  summarize(newID = paste(unique(ambigID), toString(motusTagID), sep = ": ")) %>%
  left_join(df.ambigTags, by = "ambigID")
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
# and merge that with df.alltags.path
df.alltags.path <- left_join(df.alltags.path, ambigTags.2, by = "motusTagID") %>%
  arrange(ts.h)

ggplot(data = df.alltags.path,
       aes(x = ts.h, y = recvLat, group = Ambiguous, colour = Ambiguous)) +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 45, vjust = 1, hjust = 1)) +
  geom_point() +
  geom_path() +
  facet_wrap(~ newID, scales = "free", ncol = 2)
```



Let's deal with the easy ones first.

ambigID -337: `motusTagIDs` 10811 and 16011

```
df.alltags.sub %>%
  filter(ambigID == -337) %>%
  count(motusTagID, tagDeployStart, tagDeployEnd, tagDepLat, tagDepLon)
```

```
##   motusTagID      tagDeployStart        tagDeployEnd tagDepLat tagDepLon n
## 1      10811 2014-10-28 07:00:00 2015-08-03 07:00:00   39.1140  -74.7142 4
## 2      16011 2015-08-02 11:39:59 2015-12-17 11:39:59   51.4839  -80.4500 4
```

We can see from the plot that ambiguous tag -337 is ambiguous only at the beginning of the deployment.

We can see from the summary of the tag deployment data that there were only 4 detections, at the exact latitude of deployment of tag 16011, and just before the non-ambiguous detections of `motusTagID` 16011. So the issue here is simply that the tail end of the deployment of tag 10811 slightly overlaps with the deployment of tag 16011. We can confidently claim these detections as belonging to motusTagID 16011, and remove the ambiguous detections assigned to the other tag.

We'll create another data frame to keep track of these runs.

```
# we want the detections associated with the motusTagID that we want to
# ultimately REMOVE from the data frame
df.block.2 <- df.alltags.sub %>%
```

```
  filter(ambigID == -337,
         motusTagID == 10811) %>%
  select(motusTagID, runID) %>%
  distinct()
```

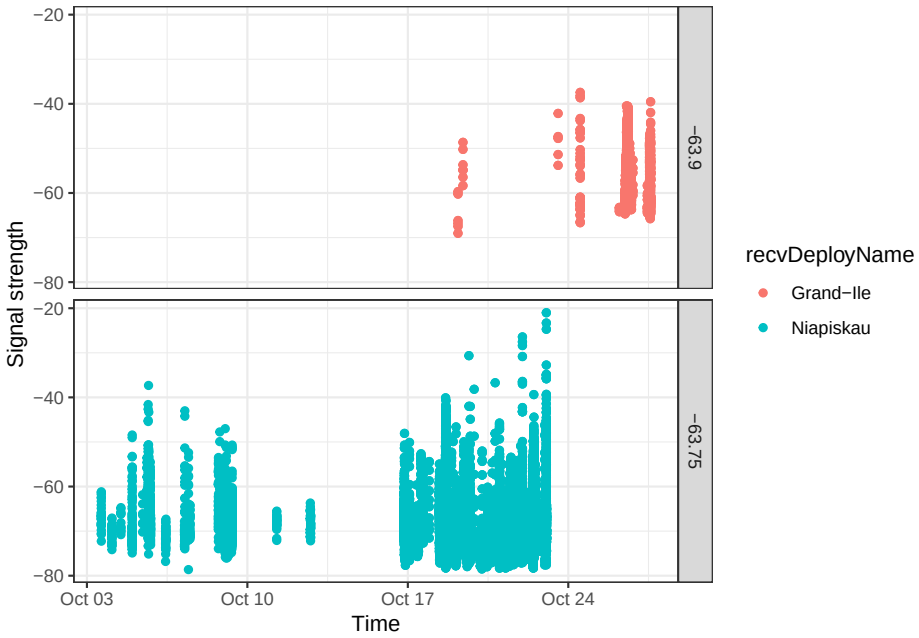
ambigID -134: motusTagIDs 22905 and 23319

```
df.alltags.sub %>%
  filter(ambigID == -134) %>%
  count(motusTagID, tagDeployStart, tagDeployEnd,
           tagDepLat, tagDepLon, month(ts))
```

```
##   motusTagID      tagDeployStart        tagDeployEnd tagDepLat tagDepLon month(ts)     n
## 1      22905 2016-10-01 16:00:00 2017-06-12 16:00:00  50.19278 -63.74528        10 22279
## 2      23319 2016-10-15 16:00:00 2017-06-26 16:00:00  50.19278 -63.74528        10 22279
```

Here we have a similar situation, but one that is a bit more complex. Two identical tags were deployed at the same location, shortly after one another. Let's examine a simple plot.

```
ggplot(data = filter(df.alltags.sub,
                     motusTagID %in% c(22905, 23319)),
       aes(x = ts, y = sig, group = recvDeployName, colour = recvDeployName)) +
  geom_point() +
  theme_bw() +
  labs(x = "Time", y = "Signal strength") +
  facet_grid(recvLon ~ .)
```



It appears that these are overlapping detections, at two sites in proximity to one another. Additional information from the field researchers may enable us to disentangle them, but it is not clear from the data.

We will therefore remove all detections of this ambiguous tag from the database.

```
# we want the detections associated with the motusTagID that we want to
# ultimately REMOVE from the data frame

df.block.3 <- df.alltags.sub %>%
  filter(ambigID == -134) %>%
  select(motusTagID, runID) %>%
  distinct()
```

ambigID -171: motusTagIDs 22778, 22902 and 22403

The ambiguous detections for this tag, which occur in the Great Lakes region, could also belong to `motusTagID` 22778 from the RBrownAMWO project or `motusTagID` 24303 from the Neonics project. Let's take a closer look at these detections.

First, find the deployment dates and locations for each tag.

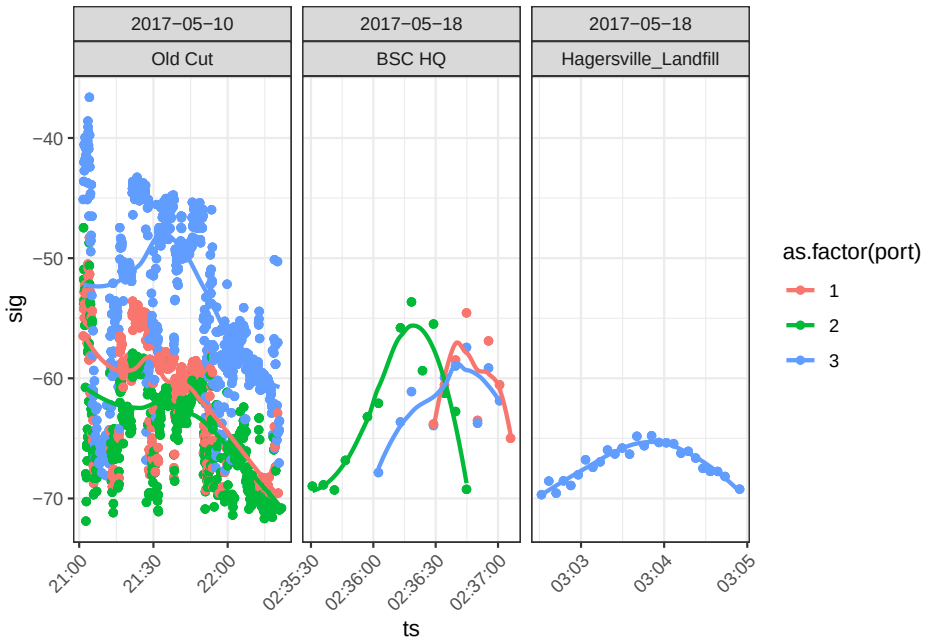
```
df.alltags.sub %>%
  filter(ambigID == -171) %>%
  filter(!is.na(tagDeployStart)) %>%
  select(motusTagID, tagProjID, start = tagDeployStart, end = tagDeployEnd,
         lat = tagDepLat, lon = tagDepLon, species = speciesEN) %>%
  distinct() %>%
  arrange(start) %>%
  collect() %>%
  as.data.frame()
```

```
##   motusTagID tagProjID               start                 end      lat       lon
## 1      22902       176 2016-10-01 16:00:00 2017-06-12 16:00:00 50.19278 -63.74528
## 2      22778        82 2016-10-21 00:00:00 2018-09-09 00:00:00 45.13535 -67.29323
## 3      24303       146 2017-05-10 22:30:59 2017-06-30 22:30:59 42.60600 -80.46900 W
```

And plot the ambiguous detections.

```
df.ambig.171 <- filter(df.alltags.sub, ambigID == -171)

ggplot(data = df.ambig.171, aes(x = ts, y = sig, colour = as.factor(port))) +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 45, vjust = 1, hjust = 1)) +
  geom_point() +
  geom_smooth(method = "loess", se = FALSE) +
  facet_wrap(as_date(ts) ~ recvDeployName, scales = "free_x")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



We see that there are a large number of ambiguous detections on 10 May 2017 at Old Cut (Long Point, Lake Erie, Ontario), consistent with a bird 'hanging around'. These are almost certainly detections of motusTagID '24303' which was deployed at Old Cut on 10 May 2017. Subsequent detections on the 18th of May are near Old Cut (Bird Studies Canada HQ, Port Rowan, Ontario), and then a location to the North of Old Cut (Hagersville, Ontario). These detections are consistent with a bird departing on migration. Note in particular the pattern in the latter two panels of increasing then decreasing signal strength which indicates a bird is flying through the beam of an antenna.

These detections belong to another project, so we simply remove all detections of that ambiguous tag from our database.

```
# we want the detections associated with the motusTagID that we want to
# ultimately REMOVE from the data frame

df.block.4 <- df.alltags.sub %>%
  filter(ambigID == -171) %>%
  select(motusTagID, runID) %>%
  distinct()
```

ambigID -114: motusTagIDs 22897 and 24298

Next we look at the ambiguities for ambiguous tag -114.

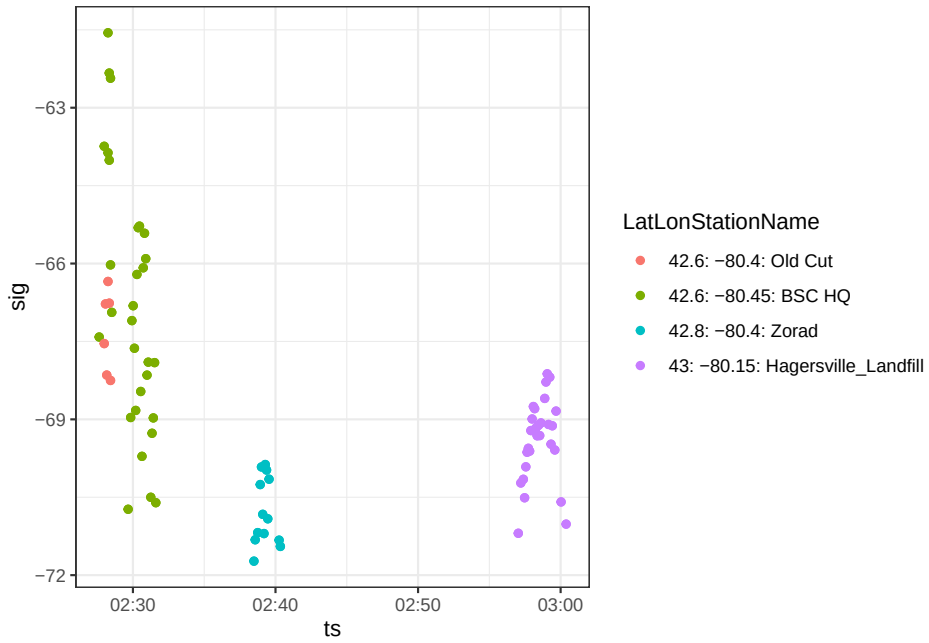
```
df.alltags.sub %>%
  filter(ambigID == -114) %>%
```

```
  filter(!is.na(tagDeployStart)) %>%
  select(motusTagID, tagProjID, start = tagDeployStart, end = tagDeployEnd,
         lat = tagDepLat, lon = tagDepLon, species = speciesEN) %>%
  distinct() %>%
  arrange(start) %>%
  collect() %>%
  as.data.frame()
```

```
##   motusTagID tagProjID               start                 end      lat       lon
## 1      22897       176 2016-10-01 16:00:00 2017-06-12 16:00:00 50.19278 -63.74528
## 2      24298       146 2017-05-10 03:00:00 2017-06-30 03:00:00 42.60690 -80.46900 W
```

We again subset these and plot them. An initial plot suggested that all of the detections are of a migratory flight, so we construct a somewhat different plot from the one above, that emphasizes this behaviour better.

```
df.ambig.114 <- df.alltags.sub %>%
  filter(ambigID == -114) %>%
  mutate(LatLonStationName = paste(recvLat, recvLon, recvDeployName, sep=": "))

ggplot(data = df.ambig.114, aes(x = ts, y = sig, colour = LatLonStationName)) +
  geom_point() +
  theme_bw()
```



Notice that the detections are consistent with a migratory departure from the Long Point area (Old Cut Field Station, Lake Erie, Ontario) about a week after

the ambiguous tag 24298 was deployed at the same location. This again suggests that these ambiguous detections can be removed from our data because they belong to another project.

```
df.block.5 <- df.alltags.sub %>%
  filter(ambigID == -114) %>%
  select(motusTagID, runID) %>%
  distinct()
```

ambigID -106: motusTagIDs 17021 and 17357

These two tags pose an interesting problem. There is only a short period of overlap, between mid August 2015 and mid September. One individual is a Grey-cheeked Thrush, tagged in Colombia, the other a White-rumped Sandpiper, associated with the sample project.

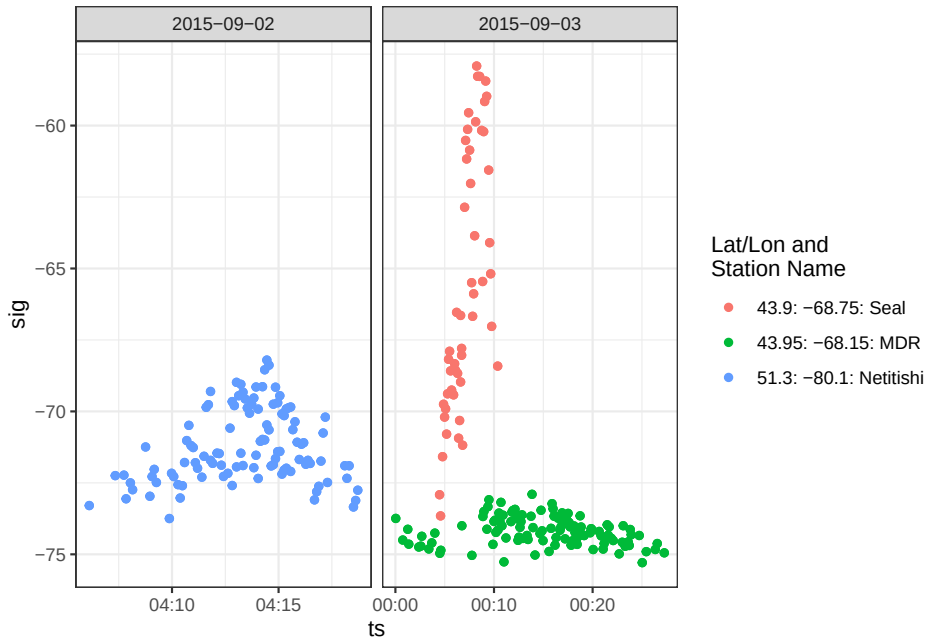
```
df.alltags.sub %>%
  filter(ambigID == -106) %>%
  filter(!is.na(tagDeployStart)) %>%
  select(motusTagID, tagProjID, start = tagDeployStart, end = tagDeployEnd,
         lat = tagDepLat, lon = tagDepLon, species = speciesEN) %>%
  distinct() %>%
  arrange(start) %>%
  collect() %>%
  as.data.frame()
```

```
##   motusTagID tagProjID               start                 end      lat       lon
## 1      17021        57 2015-04-30 05:00:00 2015-09-14 05:00:00 11.12265 -74.08735
## 2      17357       176 2015-08-11 07:20:00 2015-12-26 07:20:00 51.48390 -80.45000
```

```
                                                                    Gray-chee
                                                                White-rumped
```

We plot the ambiguous detections to examine the period of overlap.

```
df.ambig.106 <- filter(df.alltags.sub, ambigID == -106)

ggplot(data = df.ambig.106,
       aes(x = ts, y = sig,
           colour = paste(recvLat, recvLon, recvDeployName, sep = ": "))) +
  theme_bw() +
  geom_point() +
  scale_colour_discrete(name = "Lat/Lon and\nStation Name") +
  facet_wrap(~ as_date(ts), scales = "free_x")
```



Both sets of detections are long run lengths, and look valid (increasing then decreasing signal strength). They are about a day apart, and so it is possible they represent two different birds, or the departure flight of the white-rumped sandpiper from its staging ground. Let's use the `siteTrans()` function (in the `motus` package, see section C.13) to examine the flight from Netitishi to MDR/Seal (in the Gulf of Maine)

```
df.ambig.106 %>%
  filter(motusTagID == 17021) %>% # just pick one of the two ambiguous IDs
  siteTrans(latCoord = "recvLat", lonCoord = "recvLon") %>%
  ungroup() %>%
  filter(rate < 60) %>% # remove the simultaneous detections from Seal and MDR
  mutate(total.time = as.numeric(round(seconds_to_period(tot_ts)))) %>%
  select(start = recvDeployName.x, end = recvDeployName.y,
         date = ts.x, `rate(m/s)` = rate,
         dist, total.time = total.time, bearing)
```

```
## # A tibble: 1 x 7
##   start                end           date                `rate(m/s)`     dist total
##   <chr>                <chr>         <dttm>                    <dbl>    <dbl>
## 1 Netitishi_51.3, -80.1 MDR_44, -68.2 2015-09-02 04:18:42        17.1 1211567.
```

These detections are >1200 km distant from one another, but the flight speed (17 m/s) is consistent with a white-rumped Sandpiper. Given that the Gray-cheeked Thrush tag was near the end of its expected lifetime, we can reasonably claim these detections for our project, and remove the ambiguous detections associated

with motusTagID 17021.

```
df.block.6 <- df.alltags.sub %>%
  filter(ambigID == -106, motusTagID == 17021) %>%
  select(motusTagID, runID) %>%
  distinct()
```

ambigID -56: motusTagIDs 22867 and 23316

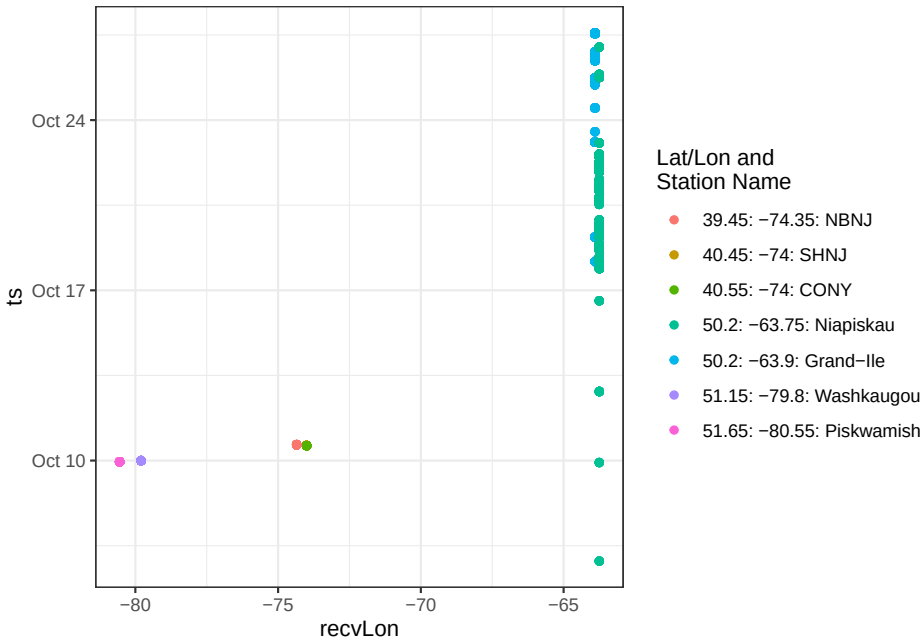
These two tags were also both deployed by the same project.

```
df.alltags.sub %>%
  filter(ambigID == -56) %>%
  filter(!is.na(tagDeployStart)) %>%
  select(motusTagID, tagProjID, start = tagDeployStart, end = tagDeployEnd,
         lat = tagDepLat, lon = tagDepLon, species = speciesEN) %>%
  distinct() %>%
  arrange(start) %>%
  collect() %>%
  as.data.frame()
```

```
##   motusTagID tagProjID               start                 end      lat       lon            s
## 1      22867       176 2016-09-06 15:35:00 2017-05-18 15:35:00 51.79861 -80.69139 Pectoral Sar
## 2      23316       176 2016-10-02 16:00:00 2017-06-13 16:00:00 50.19278 -63.74528           Re
```

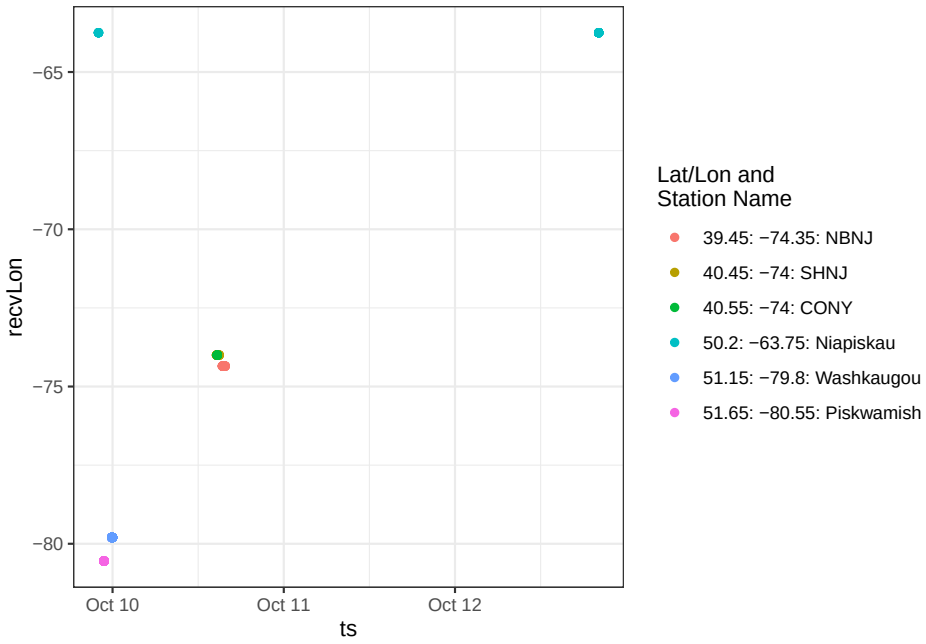
Tag 23316 was deployed by the James Bay Shorebird Project (sample project) about three weeks after tag 22867, which was deployed from a location far to the west.

```
df.ambig.56 <- df.alltags.sub %>%
  filter(ambigID == -56) %>%
  mutate(sig = ifelse(sig > 0, sig * -1, sig))

ggplot(data = df.ambig.56,
       aes(x = recvLon, y = ts,
           colour = paste(recvLat, recvLon, recvDeployName, sep=": "))) +
  theme_bw() +
  geom_point() +
  scale_colour_discrete(name="Lat/Lon and\nStation Name")
```



We can see from the plot that a tag is detected consistently near longitude -65, which is near the deployment location for `motusTagID` 23316 and after it's deployment start date, it was also present at -65 during and after detections far to the west. It's likely all the detections at -65 belong to `motusTagID` 23316, but it is also clear that anything informative about this ambiguity occurs between about 9-11 October, so let's zoom in on that part of the data set.

```
ts.begin <- ymd_hms("2016-10-06 00:00:00")
ts.end <- ymd_hms("2016-10-12 23:00:00")

ggplot(data = filter(df.ambig.56,
                     ts > ts.begin,
                     ts < ts.end),
       aes(x = ts, y = recvLon,
           colour = paste(recvLat, recvLon, recvDeployName, sep = ": "))) +
  theme_bw() +
  geom_point() +
  scale_colour_discrete(name = "Lat/Lon and\nStation Name")
```



We can see that the ambiguous tag was detected consistently at Niapiskau and Grand Ile before and after the period when it was also detected to the north and west (at Washkaugou and Piskwamish) and then to the south (NBNJ, SHNJ, and CONY). We can look at this transition by filtering out the portion of the data not near Niapiskau, and again using the siteTrans function from the motus package.

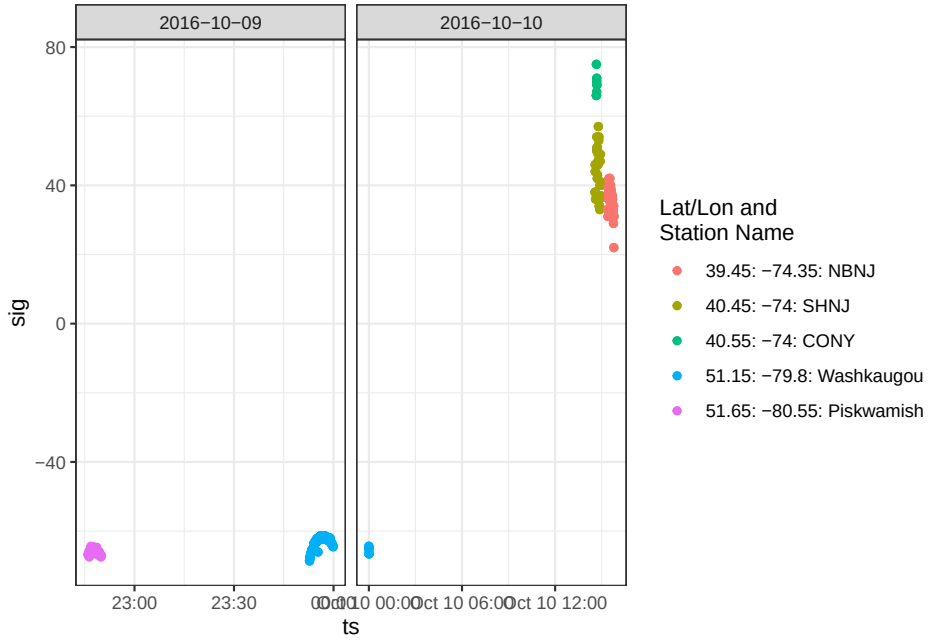
```
# other tag is a duplicate
df.56.tmp <- filter(df.ambig.56, !(recvLat == 50.2), motusTagID == 22867)

siteTrans(df.56.tmp, latCoord = "recvLat", lonCoord = "recvLon") %>%
  ungroup() %>%
  filter(rate < 60) %>% # get rid of simultaneous detections
  mutate(total.time = as.numeric(round(seconds_to_period(tot_ts)))) %>%
  select(start = recvDeployName.x,
         end = recvDeployName.y,
         date = ts.x, `rate(m/s)` = rate,
         dist, total.time = total.time, bearing)
```

```
## # A tibble: 2 x 7
##   start                  end                    date                `rate(m/s)`     dist total
##   <chr>                  <chr>                  <dttm>                    <dbl>    <dbl>
## 1 Piskwamish_51.7, -80.6 Washkaugou_51.2, -79.8 2016-10-09 22:49:59        20.3   76280.
## 2 Washkaugou_51.2, -79.8 SHNJ_40.5, -74         2016-10-10 00:00:42        24.3 1270877.
```

The bird made a 14.5 hour flight between Washkaugou and SHNJ at a rate of

24 m/s, which is plausible. The researchers involved may have other data to support or refute the inference (e.g. an actual sighting of the Red Knot still in Niapiskau after this flight was recorded) but it seems likely that while one tag remained at sites around longitude -65, another tag made the above migratory flights. We can make another more detailed plot of signal strength to examine these potential migratory flights more closely:

```
df.56.tmp <- filter(df.alltags.sub, ambigID == -56, recvLon < -70)

ggplot(data = df.56.tmp,
       aes(x = ts, y = sig,
           colour = paste(recvLat, recvLon, recvDeployName, sep = ": "))) +
  theme_bw() +
  geom_point() +
  scale_colour_discrete(name = "Lat/Lon and\nStation Name") +
  facet_wrap(~ as_date(ts), scales = "free_x")
```



These look like typical fly-by patterns of increasing and then decreasing signal strength. This, coupled with overall detection patterns and knowledge of the species, leads us to believe that the ambiguous detections can be reasonably divided between the two individuals; one detected consistently around longitude -65 (23316), and the other migrating SW during the same period (22867).

To address this problem, we need to create two filters - one that excludes ambiguous detections of tag 22867, and one that excludes some detections of 23316. In this instance, we can do this most easily by filtering on `motusTagID`

and `recvDeployName`.

```
# tag 23316 was only ever at "Grand-Ile", "Niapiskau", and tag 22867 was never
# detected at those sites. So we exclude all detections not at "Grand-Ile",
# "Niapiskau" for motusTag 23316, and do the opposite for tag 22867.

df.block.7 <- df.alltags.sub %>%
  filter(ambigID == -56,
         motusTagID == 23316,
         !(recvDeployName %in% c("Grand-Ile", "Niapiskau"))) %>%
  select(motusTagID, runID) %>%
  distinct()

df.block.8 <- df.alltags.sub %>%
  filter(ambigID == -56,
         motusTagID == 22867,
         recvDeployName %in% c("Grand-Ile", "Niapiskau")) %>%
  select(motusTagID, runID) %>%
  distinct()
```

5.8 Checking validity of run lengths of 2 or 3

At the beginning of this chapter, we dropped all detections with a run length of 2 or 3 or a run length of 4 in noisy conditions, because they are considered to have a high probability of being false positive. Now that we've cleaned the data, and are confident in the detections that remain, you might at this point decide to go back and take a closer look at those omitted detections. You could do this, for example, by re-running the various plots described in this chapter (begin with lat/lon by time plots), to see if any of those detections make sense in the context of where the true detections lie. It is up to the user to decide which detections are reasonable in terms of the biology and behaviour of each tagged individual.

5.9 Filtering the data

5.9.1 Filter and save to RDS

To filter the data, we can omit rows in the `df.block` data frames from the original data using a `anti_join()`, which removes rows from `x` (`df.alltags.sub`) which are present in `y` (`df.block`).

```
# combine our df.block data frames into a single dataframe
df.block.all <- bind_rows(df.block.0, df.block.2, df.block.3,
```

```
                          df.block.4, df.block.5, df.block.6, df.block.7,
                          df.block.8)


df.alltags.sub <- anti_join(df.alltags.sub, df.block.all, by = c("runID", "motusTagID"))
```

Now save the local data frame as an RDS file, for use in the next chapter. Recall from Section 3.6 that the RDS format preserves the R data structure, including time stamps. The other benefit of saving to RDS is that you have the output from a given workflow saved as a flat file, which you can access again with a simple `readRDS` statement.

```
saveRDS(df.alltags.sub, file = "./data/dfAlltagsSub.rds")
```

And to read the data in again:

```
df.alltags.sub <- readRDS("./data/dfAlltagsSub.rds")
```

5.9.2 Save a custom filter in the motus database, and apply it to the data

As an alternative to saving your data as an RDS file, the Motus R package offers functionalities to save your filters directly within your .motus file. Once they are saved in your database, you can do the type of `anti_join()` as above without having to rely on dataframes or an RDS file to store your data. To learn more about the functions available to work with Motus filters, refer to Appendix D for more details.

```
# combine our df.block data frames into a single dataframe
df.block.all <- bind_rows(df.block.0, df.block.2, df.block.3,
                          df.block.4, df.block.5, df.block.6, df.block.7,
                          df.block.8) %>%
  mutate(probability = 0)

# create a new filter with name filtAmbigFalsePos and populate it with df.block.all
tbl.filter <- writeRunsFilter(sql.motus, "filtAmbigFalsePos",
                              df = df.block.all, delete = TRUE)
```

```
## Warning in createRunsFilter(src, filterName, motusProjID, update = FALSE): Warning:
## to update the properties (e.g. name) of the existing filter.
```

```
## Filter records saved
```

```
# obtain a table object where the filtered records from tbl.filter.1 have been removed
tbl.alltags.sub <- anti_join(tbl.alltags, tbl.filter, by = c("runID", "motusTagID"))
```

Chapter 6

Exploring data with the Motus R package

This chapter was contributed by Tara L. Crewe, Zoe Crysler, and Philip Taylor.

Once you have clarified any possible ambiguous tags, and removed false positives, you are ready to start analyzing your clean data set. This chapter will walk you through some simple procedures to start working with and visualizing the clean sample data set; you can modify these scripts to work with your own data. For a more in-depth R tutorial we strongly recommend working through R for Data Science by Garrett Grolemund and Hadley Wickham (http://r4ds.had.co.nz/).

6.1 Load required packages

Follow the instructions in Chapter 2 to install the following packages before loading, if you haven't already done so.

```
library(motus)
library(tidyverse)
library(ggmap)
library(lubridate)
```

```
Sys.setenv(TZ = "UTC")
```

6.2 Load data

If you followed along with the the previous Chapter (Chapter 5) and are working with the cleaned `df.alltags.sub` file, you can skip this step and move to section

6.3.

Otherwise, if you saved your data as an RDS file, you can load it using:

```
df.alltags.sub <- readRDS("./data/dfAlltagsSub.rds") # change dir to local directory
```

Or, if you've applied a custom filter to your .motus file, you can load the previously downloaded sample motus data (see Chapter 3) and clean it now. Currently the main benefit of using the custom filter is that you apply the filter to the .motus file, which allows you more flexibility in applying `dplyr` functions to manage and filter the data (e.g., you can select different variables to include in the data than we included in the RDS file in Chapter 5). This approach also allows you to more readily integrate new data added to your database with the tagme function. Because we are selecting the same variables and filtering the same records, the following gives you the same dataset as the `readRDS` statement above:

```
# load the .motus file (remember 'motus.sample' is both username and password)
proj.num <- 176
sql.motus <- tagme(proj.num, update = TRUE, dir = "./data/")
tbl.alltags <- tbl(sql.motus, "alltagsGPS")

# obtain a table object of the filter
tbl.filter <- getRunsFilters(sql.motus, "filtAmbigFalsePos")

# filter and convert the table into a dataframe, with a few modications
df.alltags.sub <- left_join(tbl.alltags, tbl.filter, by = c("runID", "motusTagID")) %>%
  mutate(probability = ifelse(is.na(probability), 1, probability),
         recvLat = if_else((is.na(gpsLat)|gpsLat == 0|gpsLat == 999),
                           recvDeployLat,
                           gpsLat),
         recvLon = if_else((is.na(gpsLon)|gpsLon == 0|gpsLon == 999),
                           recvDeployLon,
                           gpsLon),
         recvAlt = if_else(is.na(gpsAlt),
                           recvDeployAlt,
                           gpsAlt)) %>%
  filter(probability > 0) %>%
  select(-noise, -slop, -burstSlop, -done, -bootnum, -codeSet,
         -mfg, -nomFreq, -markerNumber, -markerType, -tagDepComments,
         -fullID, -deviceID, -recvDeployLat, -recvDeployLon, -recvDeployAlt,
         -speciesGroup, -gpsLat, -gpsLon, -recvAlt, -recvSiteName) %>%
  collect() %>%
  as.data.frame() %>%
  mutate(ts = as_datetime(ts),  # work with dates AFTER transforming to flat file
         tagDeployStart = as_datetime(tagDeployStart),
         tagDeployEnd = as_datetime(tagDeployEnd),
```

```
        recvDeployName = if_else(is.na(recvDeployName),
                                 paste(recvLat, recvLon, sep=":"),
                                 recvDeployName))
```

Note that if your project is very large, you may want to convert only a portion of it to the dataframe, to avoid memory issues. Details on filtering the tbl prior to collecting as a dataframe are available in section 3.5.8.

Here we do so by adding a filter to the above command, in this case, only creating a dataframe for `motusTagID` 16047, but you can decide how to best subset your data based on your need (e.g. by species or year):

```
# create a subset for a single tag, to keep the dataframe small
df.alltags.16047 <- filter(df.alltags.sub, motusTagID == 16047)
```

6.3 Summarizing your data

Here we will run through some basic commands, starting with the `summary()` function to view a selection of variables in a data frame:

```
sql.motus %>%
  tbl("alltags") %>%
  select(ts, motusTagID, runLen, speciesEN, tagDepLat, tagDepLon,
         recvDeployLat, recvDeployLon) %>%
  collect() %>%
  summary()
```

```
##        ts              motusTagID        runLen        speciesEN           tagDepLat       tag
##  Min.   :1.438e+09   Min.   :10811   Min.   :   2.0   Length:108826      Min.   :11.12   Min.
##  1st Qu.:1.476e+09   1st Qu.:22897   1st Qu.:  30.0   Class :character   1st Qu.:50.19   1st Q
##  Median :1.477e+09   Median :22905   Median : 122.0   Mode  :character   Median :50.19   Media
##  Mean   :1.476e+09   Mean   :22660   Mean   : 355.9                      Mean   :50.14   Mean
##  3rd Qu.:1.477e+09   3rd Qu.:23316   3rd Qu.: 404.0                      3rd Qu.:50.19   3rd Q
##  Max.   :1.498e+09   Max.   :24303   Max.   :2474.0                      Max.   :51.80   Max.
##                                                                          NA's   :2025    NA's
```

```
# same summary for the filtered sql data
df.alltags.sub %>%
  select(ts, motusTagID, runLen, speciesEN, tagDepLat, tagDepLon,
         recvLat, recvLon) %>%
  summary()
```

```
##        ts                          motusTagID        runLen        speciesEN           tagDepLat
##  Min.   :2015-08-03 06:37:11   Min.   :16011   Min.   :   4.0   Length:48133       Min.   :50.
##  1st Qu.:2016-10-06 19:17:36   1st Qu.:22897   1st Qu.:  27.0   Class :character   1st Qu.:50.
##  Median :2016-10-09 21:48:11   Median :22897   Median :  97.0   Mode  :character   Median :50.
```

```
##  Mean   :2016-09-05 10:37:42   Mean   :22267   Mean   : 235.1                    M
##  3rd Qu.:2016-10-19 10:37:42   3rd Qu.:22897   3rd Qu.: 287.0                    3
##  Max.   :2017-04-20 22:33:19   Max.   :23316   Max.   :1371.0                    M
##
```

The `dplyr` package allows you to easily summarize data by group, manipulate variables, or create new variables based on your data.

We can manipulate existing variables or create new ones with `dplyr`'s `mutate()` function, here we'll convert ts to a POSIXct format, then make a new variable for year and day of year (doy).

We'll also remove the set of points with missing receiver latitude and longitudes. These may be useful in some contexts (for example if the approximate location of the receiver is known) but can cause warnings or errors when plotting.

```
df.alltags.sub <- df.alltags.sub %>%
  mutate(ts = as_datetime(ts, tz = "UTC"), # convert ts to POSIXct format
         year = year(ts), # extract year from ts
         doy = yday(ts)) %>% # extract numeric day of year from ts
  filter(!is.na(recvLat))

head(df.alltags.sub)
```

```
##    hitID runID batchID                  ts tsCorrected sig sigsd freq freqsd motusTa
## 1  45107  8886      53 2015-10-26 11:19:49  1445858390  52     0    4      0      16
## 2  45108  8886      53 2015-10-26 11:20:28  1445858429  54     0    4      0      16
## 3  45109  8886      53 2015-10-26 11:21:17  1445858477  55     0    4      0      16
## 4  45110  8886      53 2015-10-26 11:21:55  1445858516  52     0    4      0      16
## 5  45111  8886      53 2015-10-26 11:22:44  1445858564  49     0    4      0      16
## 6 199885 23305      64 2015-10-26 11:12:04  1445857924  33     0    4      0      16
##   pulseLen tagDeployID speciesID      tagDeployStart        tagDeployEnd tagDepLat t
## 1      2.5        1839      4670 2015-09-10 18:00:00 2016-03-10 18:00:00   51.4839
## 2      2.5        1839      4670 2015-09-10 18:00:00 2016-03-10 18:00:00   51.4839
## 3      2.5        1839      4670 2015-09-10 18:00:00 2016-03-10 18:00:00   51.4839
## 4      2.5        1839      4670 2015-09-10 18:00:00 2016-03-10 18:00:00   51.4839
## 5      2.5        1839      4670 2015-09-10 18:00:00 2016-03-10 18:00:00   51.4839
## 6      2.5        1839      4670 2015-09-10 18:00:00 2016-03-10 18:00:00   51.4839
##   recvUtcOffset antType antBearing antHeight speciesEN          speciesFR       spec
## 1            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris c
## 2            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris c
## 3            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris c
## 4            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris c
## 5            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris c
## 6            NA  yagi-9        243        NA  Red Knot Bécasseau maubèche Calidris c
##   year doy
## 1 2015 299
## 2 2015 299
```

```
## 3 2015 299
## 4 2015 299
## 5 2015 299
## 6 2015 299
```

We can also summarize information by group, in this case `motusTagID`, and apply various functions to these groups such as getting the total number of detections (n) for each tag, the number of receivers each tag was detected on, the first and last detection date, and the total number of days there was at least one detection:

```
tagSummary <- df.alltags.sub %>%
  group_by(motusTagID) %>%
  summarize(nDet = n(),
            nRecv = length(unique(recvDeployName)),
            tsMin = min(ts),
            tsMax = max(ts),
            totDay = length(unique(doy)))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
head(tagSummary)
```

```
## # A tibble: 6 x 6
##   motusTagID  nDet nRecv tsMin               tsMax               totDay
##        <int> <int> <int> <dttm>              <dttm>               <int>
## 1      16011   116     1 2015-08-03 06:37:11 2015-08-05 20:41:12      3
## 2      16035   415     5 2015-08-14 17:53:49 2015-09-02 14:06:09      6
## 3      16036    62     1 2015-08-17 21:56:44 2015-08-17 21:58:52      1
## 4      16037  1278     3 2015-08-23 15:13:57 2015-09-08 18:37:16     14
## 5      16038    70     1 2015-08-20 18:42:33 2015-08-22 22:19:37      3
## 6      16039  1044    10 2015-08-23 02:28:45 2015-09-19 06:08:31      8
```

We can also group by multiple variables; applying the same function as above but now grouping by `motusTagID` and `recvDeployName`, we will get information for each tag detected on each receiver. Since we are grouping by `recvDeployName`, there will be by default only one `recvDeployName` in each group, thus the variable `nRecv` will be 1 for each row. This in not very informative, however we include this to help illustrate how grouping works:

```
tagRecvSummary <- df.alltags.sub %>%
  group_by(motusTagID, recvDeployName) %>%
  summarize(nDet = n(),
            nRecv = length(unique(recvDeployName)),
            tsMin = min(ts),
            tsMax = max(ts),
            totDay = length(unique(doy)))
```

```
## `summarise()` regrouping output by 'motusTagID' (override with `.groups` argument)
```

```
head(tagRecvSummary)
```

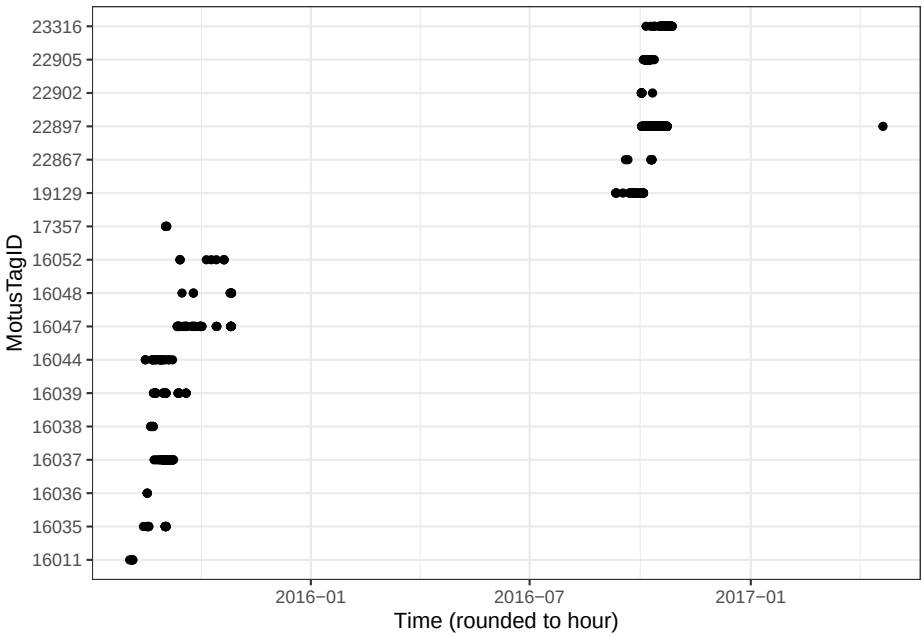
```
## # A tibble: 6 x 7
## # Groups:   motusTagID [2]
##   motusTagID recvDeployName  nDet nRecv tsMin               tsMax               totl
##        <int> <chr>          <int> <int> <dttm>              <dttm>               <i
## 1      16011 North Bluff      116     1 2015-08-03 06:37:11 2015-08-05 20:41:12
## 2      16035 Brier2            38     1 2015-09-02 14:03:19 2015-09-02 14:06:09
## 3      16035 D'Estimauville    32     1 2015-09-02 07:58:43 2015-09-02 08:04:24
## 4      16035 Netitishi        274     1 2015-08-14 17:53:49 2015-09-01 21:35:32
## 5      16035 Southwest Head    65     1 2015-09-02 13:06:13 2015-09-02 13:14:39
## 6      16035 Swallowtail        6     1 2015-09-02 13:21:27 2015-09-02 13:22:22
```

6.4 Plotting your data

Plotting your data is a powerful way to visualize broad and fine-scale detection patterns. This section will give you a brief introduction to plotting using `ggplot2`. For more in depth information on the uses of `ggplot2`, we recommend the Cookbook for R, and the Rstudio `ggplot2` cheatsheet.

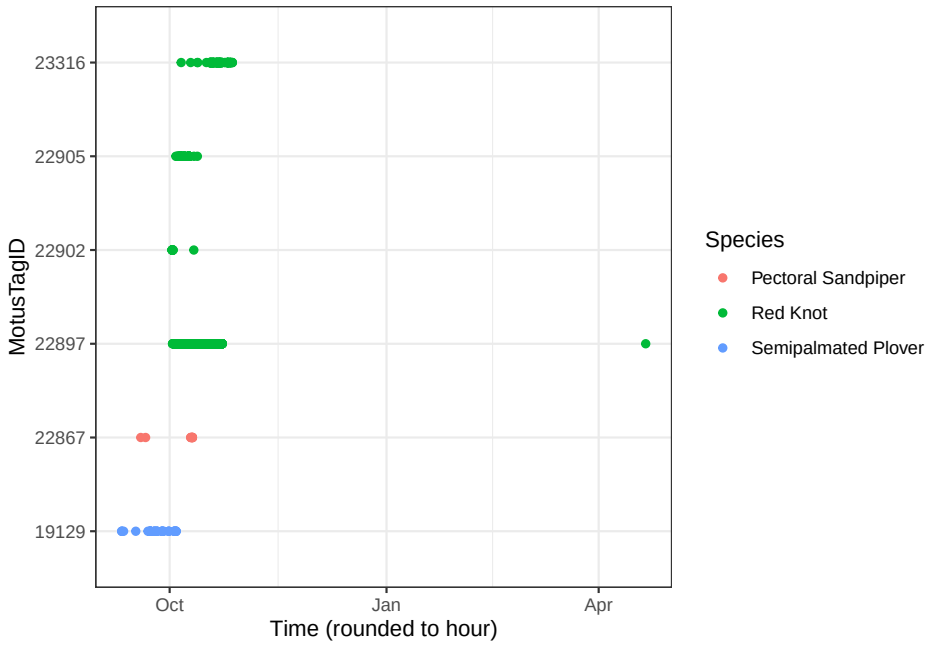
To make coarse-scale plots with large files, we suggest first rounding the detection time to the nearest hour or day so that processing time is faster. Here we round detection times to the nearest hour, then make a basic plot of hourly detections by `motusTagID`:

```
df.alltags.sub.2 <- df.alltags.sub %>%
  mutate(hour = as.POSIXct(round(ts, "hour"))) %>%
  select(motusTagID, port, tagDeployStart, tagDepLat, tagDepLon,
         recvLat, recvLon, recvDeployName, antBearing, speciesEN, year, doy, hour) %>%
  distinct()

ggplot(data = df.alltags.sub.2, aes(x = hour, y = as.factor(motusTagID))) +
  theme_bw() +
  geom_point() +
  labs(x = "Time (rounded to hour)", y = "MotusTagID")
```



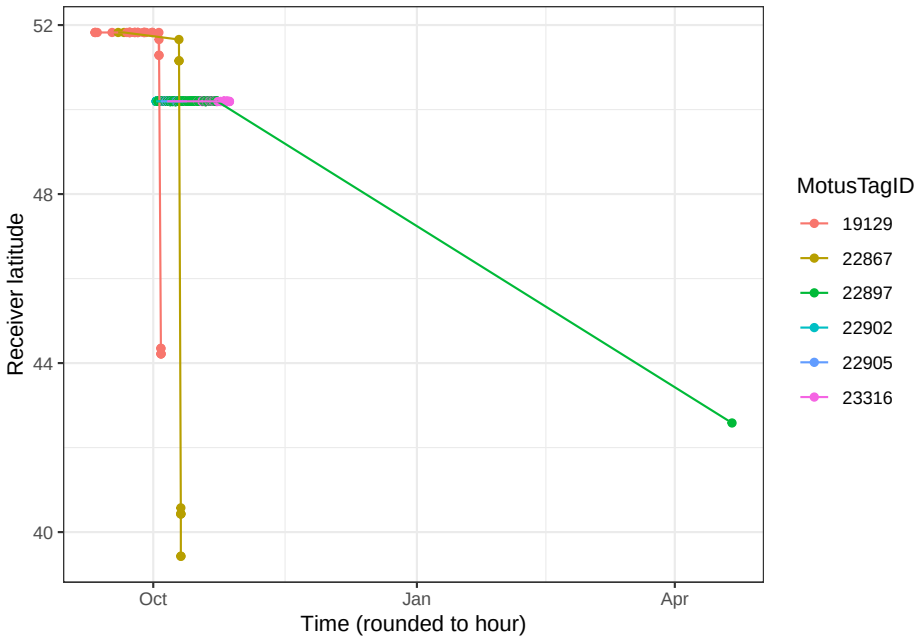
Let's focus only on tags deployed in 2016, and we can colour the tags by species:

```
ggplot(data = filter(df.alltags.sub.2, year(tagDeployStart) == 2016),
       aes(x = hour, y = as.factor(motusTagID), colour = speciesEN)) +
  theme_bw() +
  geom_point() +
  labs(x = "Time (rounded to hour)", y = "MotusTagID") +
  scale_colour_discrete(name = "Species")
```



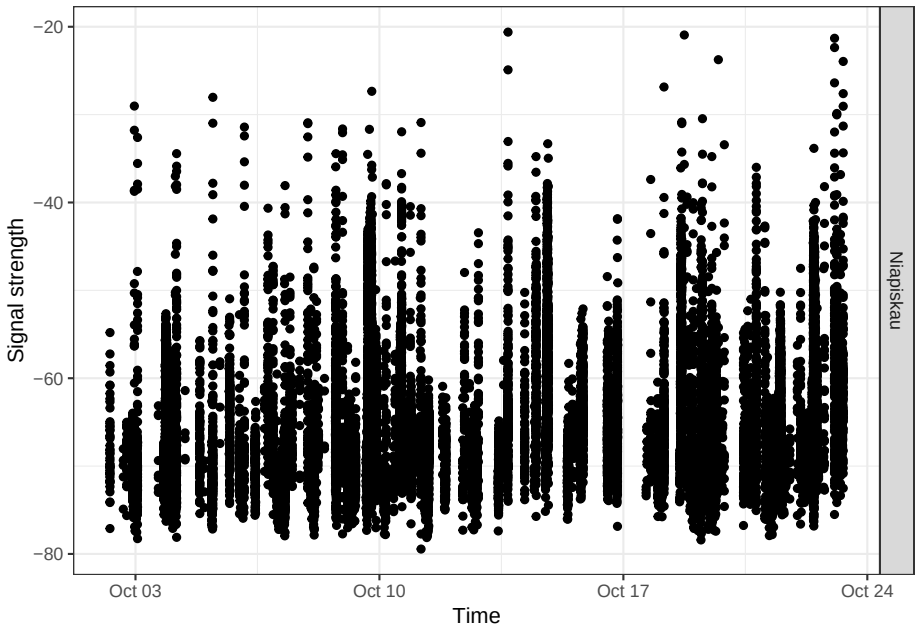
We can see how tags moved latitudinally by first ordering by hour, and colouring by `motusTagID`:

```
df.alltags.sub.2 <- arrange(df.alltags.sub.2, hour)

ggplot(data = filter(df.alltags.sub.2, year(tagDeployStart) == 2016),
       aes(x = hour, y = recvLat, col = as.factor(motusTagID),
           group = as.factor(motusTagID))) +
  theme_bw() +
  geom_point() +
  geom_path() +
  labs(x = "Time (rounded to hour)", y = "Receiver latitude") +
  scale_colour_discrete(name = "MotusTagID")
```



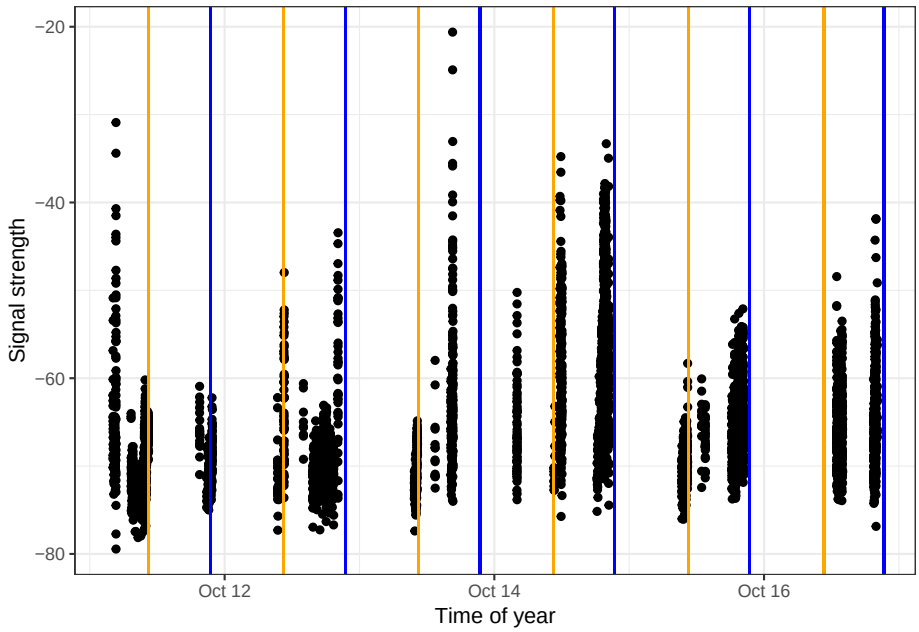
Now lets look at more detailed plots of signal variation. We use the full `df.alltags.sub` dataframe so that we can get signal strength for each detection of a specific tag. Let's examine fall 2016 detections of tag 22897 at Niapiskau; we facet the plot by deployment name, ordered by decreasing latitude:

```
ggplot(data = filter(df.alltags.sub,
                     motusTagID == 22897,
                     recvDeployName == "Niapiskau"),
       aes(x = ts, y = sig)) +
  theme_bw() +
  geom_point() +
  labs(x = "Time", y = "Signal strength") +
  facet_grid(recvDeployName ~ .)
```



We use the `sunRiseSet()` function available in the `motus` R package (see C.2) to get sunrise and sunset times for all detections. We then zoom in on a certain timeframe and add that information to the above plot by adding a `geom_vline()` statement to the code, which adds a yellow line for sunrise time, and a blue line for sunset time:

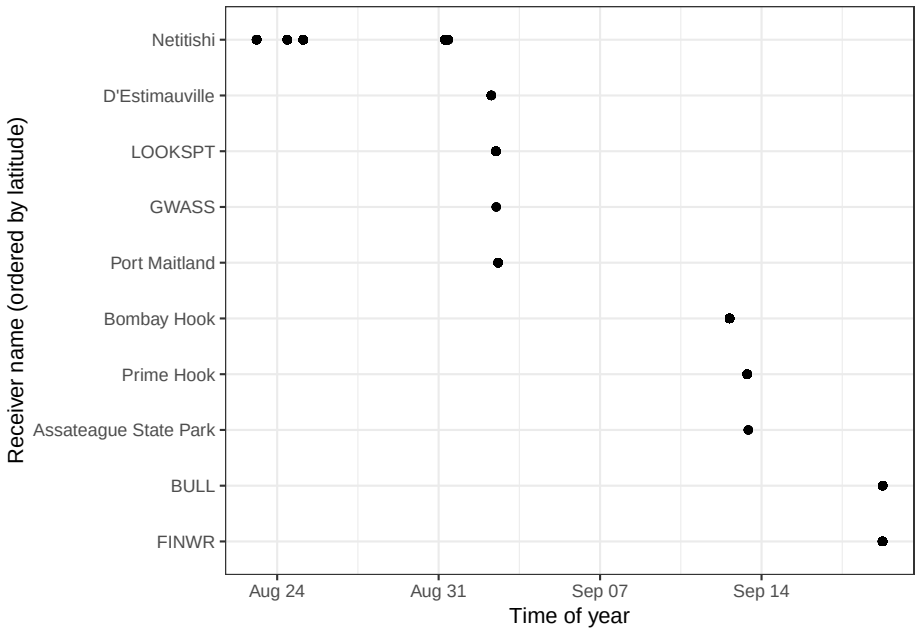
```
# add sunrise and sunset times to the dataframe
df.alltags.sub <- sunRiseSet(df.alltags.sub, lat = "recvLat", lon = "recvLon")

ggplot(data = filter(df.alltags.sub, motusTagID == 22897,
                     ts > ymd("2016-10-11"),
                     ts < ymd("2016-10-17"),
                     recvDeployName == "Niapiskau"),
       aes(x = ts, y = sig)) +
  theme_bw() +
  geom_point() +
  labs(x = "Time of year", y = "Signal strength") +
  geom_vline(aes(xintercept = sunrise), col = "orange") +
  geom_vline(aes(xintercept = sunset), col = "blue")
```



We can see that during this period, the tag was most often detected during the day, suggesting it may be actively foraging in this area during this time.

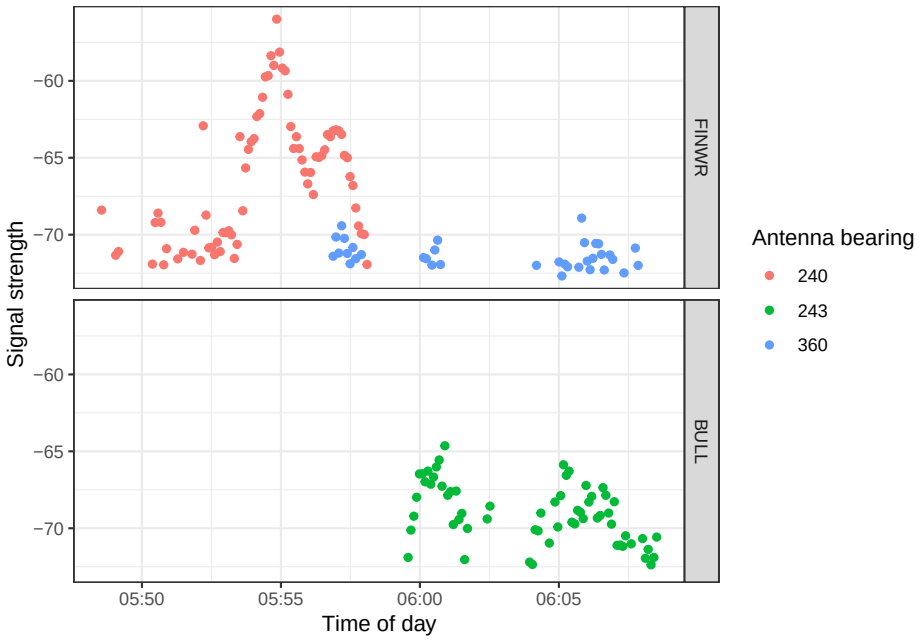
The same plots can provide valuable movement information when the receivers are ordered geographically. We do this for `motusTagID` 16039:

```
# We'll first order sitelat by latitude (for plots)
df.alltags.sub <- mutate(df.alltags.sub,
                         recvDeployName = reorder(recvDeployName, recvLat))

ggplot(data = filter(df.alltags.sub,
                     motusTagID == 16039,
                     ts < ymd("2015-10-01")),
       aes(x = ts, y = recvDeployName)) +
  theme_bw() +
  geom_point() +
  labs(x = "Time of year", y = "Receiver name (ordered by latitude)")
```



We zoom in on a section of this plot and look at antenna bearings to see directional movement past stations:

```
ggplot(data = filter(df.alltags.sub, motusTagID == 16039,
                     ts > ymd("2015-09-14"),
                     ts < ymd("2015-10-01")),
       aes(x = ts, y = sig, col = as.factor(antBearing))) +
  theme_bw() +
  geom_point() +
  labs(x = "Time of day", y = "Signal strength") +
  scale_color_discrete(name = "Antenna bearing") +
  facet_grid(recvDeployName ~ .)
```



This plot shows the typical flyby pattern of a migrating animal, with signal strength increasing and then decreasing as the tag moves through the beams of the antennas.

6.5 Mapping your data

To generate maps of tag paths, we will once again use summarized data so we can work with a much smaller database for faster processing. Here we'll summarize detections by day. As we did in Chapter 5, we create a simple function to summarize the data, since we will likely want to do this type of summary over and over again.

```
# Simplify the data by summarizing by the runID
# If you want to summarize at a finer (or coarser) scale, you can also create
# other groups. The simplest alternative is a rounded timestamp variable; for
# example by using mutate(ts.h = plyr::round_any(ts, 3600) function call. Other
# options are to just use date (e.g date = as_date(ts))

fun.getpath <- function(df) {
  df %>%
    filter(tagProjID == proj.num, # keep only tags registered to the sample project
           !is.na(recvLat) | !(recvLat == 0)) %>%
    group_by(motusTagID, runID, recvDeployName, ambigID,
             tagDepLon, tagDepLat, recvLat, recvLon) %>%
```

```
    summarize(max.runLen = max(runLen), ts.h = mean(ts)) %>%
    arrange(motusTagID, ts.h) %>%
    data.frame()
} # end of function call

df.alltags.path <- fun.getpath(df.alltags.sub)
```

```
## `summarise()` regrouping output by 'motusTagID', 'runID', 'recvDeployName', 'ambigI
```

```
df.alltags.sub.path <- df.alltags.sub %>%
  filter(tagProjID == proj.num) %>% # only tags registered to project
  arrange(motusTagID, ts) %>%       # order by time stamp for each tag
  mutate(date = as_date(ts)) %>%    # create date variable
  group_by(motusTagID, date, recvDeployName, ambigID,
           tagDepLon, tagDepLat, recvLat, recvLon)

df.alltags.path <- fun.getpath(df.alltags.sub.path)
```

```
## `summarise()` regrouping output by 'motusTagID', 'runID', 'recvDeployName', 'ambigI
```

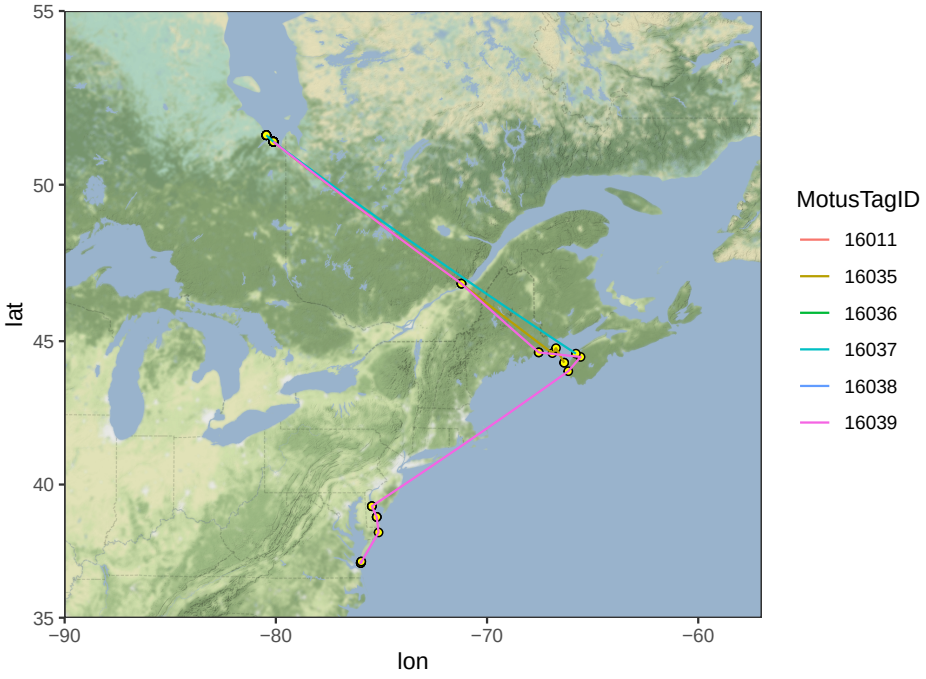
6.5.1 Mapping with the ggmap package

Mapping with `ggmap` can be a fast way to view flight paths and allows you to select from multiple base layers.
There are several ways to use the `ggmap` package. One is with Google Maps, which, as of October 16, 2018, requires a Google Maps API key (see Appendix B section B.4 for more details). An alternative method is to use an open source of map tiles, such as Stamen Map tiles, which do not require an API key. The following example uses Stamen Map tiles.

The first step is to create a map with a specified map are, maptype ("terrain", "toner", or "watercolor", among others), and level of zoom (integer for zoom 3-21, 3 being continent level, 10 being city-scale. For stamen maps this represents the level of detail you want). We then add points for receivers and lines connecting consecutive detections by `motusTagID`. We can also add points for all receivers that were active during a certain time period if we have already downloaded all metadata.

```
gmap <-  get_stamenmap(bbox = c(left = -90, right = -57, bottom = 35, top = 55),
                       maptype = "terrain-background", # select maptype
                       zoom = 6) # zoom, must be a whole number

# just use the tags that we have examined carefully and filtered (in the
# previous chapter)
df.tmp <- df.alltags.path %>%
  filter(motusTagID %in% c(16011, 16035, 16036, 16037, 16038, 16039)) %>%
```

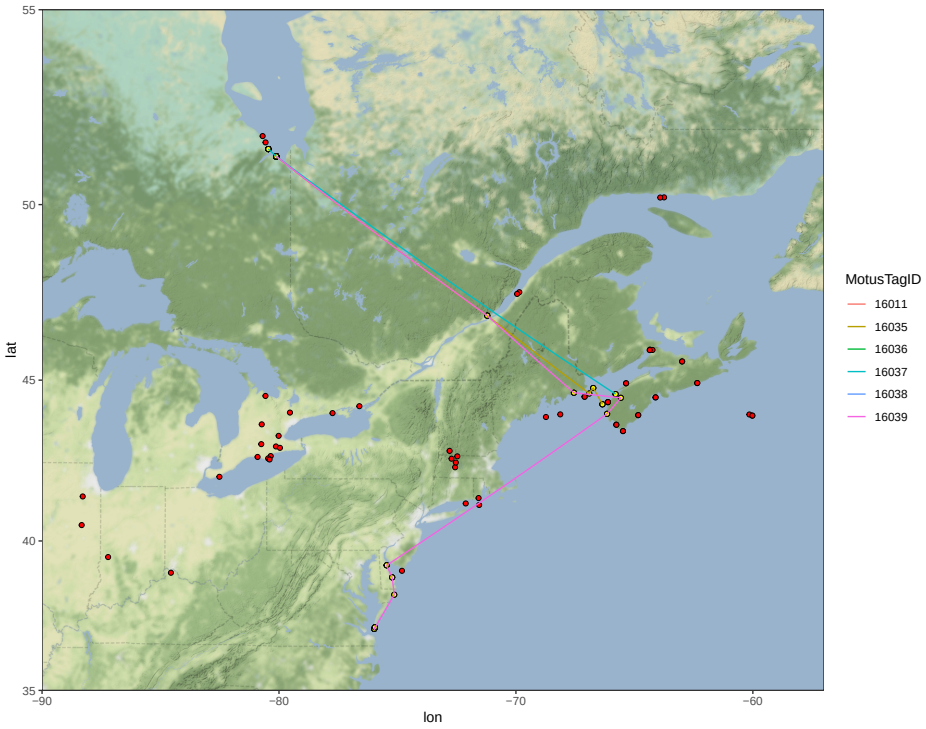
```
  arrange(ts.h)  %>% # arange by hour
  as.data.frame()

ggmap(gmap) +
  theme_bw() +
  geom_point(data = df.tmp, aes(x = recvLon, y = recvLat),
             shape = 21, colour = "black", fill = "yellow") +
  geom_path(data = df.tmp,
            aes(x = recvLon, y = recvLat, group = motusTagID, col = as.factor(motusTagID))) +
  scale_color_discrete(name = "MotusTagID")
```



We make the same plot, with additional points for all receivers that were active during a specified time (concentrating on our area of interest):

```
# get receiver metadata
tbl.recvDeps <- tbl(sql.motus, "recvDeps")
df.recvDeps <- tbl.recvDeps %>%
  collect() %>%
  as.data.frame() %>%
  mutate(tsStart = as_datetime(tsStart, tz = "UTC", origin = "1970-01-01"),
         tsEnd = as_datetime(tsEnd, tz = "UTC", origin = "1970-01-01"),
         # for deployments with no end dates, make an end date a year from now
         tsEnd = if_else(is.na(tsEnd),
                         as.POSIXct(format(Sys.time(), "%Y-%m-%d %H:%M:%S")) +
```

```
                            dyears(1),
                           tsEnd),
         tsEnd = as.POSIXct(tsEnd, tz = "UTC", origin = "1970-01-01"))

# get running intervals for all receiver deployments
siteOp <- with(df.recvDeps, interval(tsStart, tsEnd))

# set the date range you're interested in
dateRange <- interval(as.POSIXct("2015-08-01"), as.POSIXct("2016-01-01"))

# create new variable "active" which will be set to TRUE if the receiver was
# active at some point during your specified date range, and FALSE if not
df.recvDeps$active <- int_overlaps(siteOp, dateRange)

# create map with receivers active during specified date range as red, and
# receivers with detections as yellow
ggmap(gmap) +
  theme_bw() +
  geom_point(data = subset(df.recvDeps, active == TRUE),
             ggplot2::aes(x = longitude, y = latitude),
             shape = 21, colour = "black", fill = "red") +
  geom_point(data = df.tmp, aes(x = recvLon, y = recvLat),
             shape = 21, colour = "black", fill = "yellow") +
  geom_path(data = df.tmp,
            aes(x = recvLon, y = recvLat, group = motusTagID,
                col = as.factor(motusTagID))) +
  scale_color_discrete(name = "MotusTagID")
```



6.5.2 Creating simple outline maps

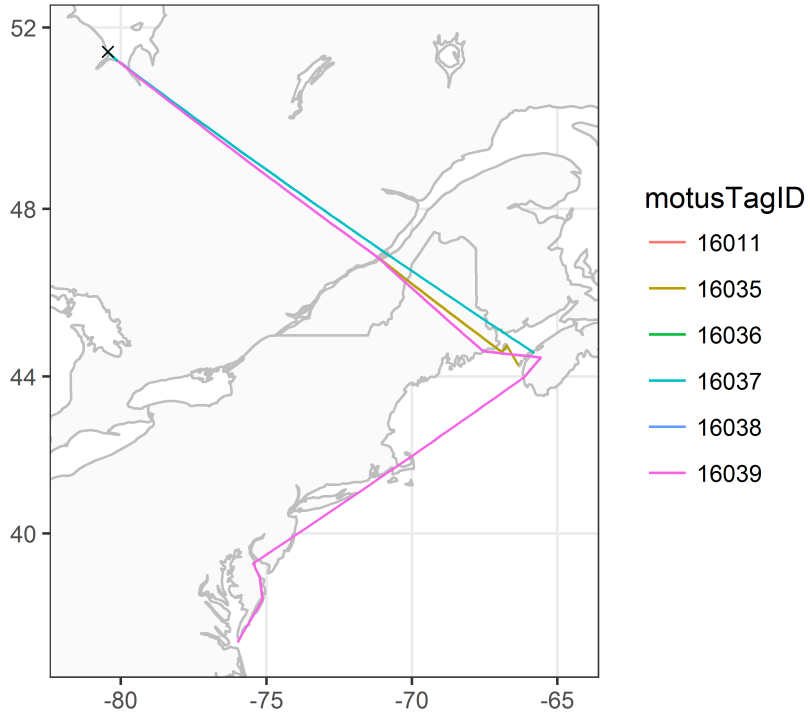
We load the base maps.

```
na.lakes <- map_data(map = "lakes")
na.lakes <- mutate(na.lakes, long = long- 360)

# Include all of the Americas to begin
na.map <- map_data(map = "world2") %>%
  filter(region %in% c("Canada", "USA")) %>%
  mutate(long = long - 360)
```

Then, to map the paths, we set the x-axis and y-axis limits based on the location of receivers with detections. Depending on your data, these might need to be modified to encompass the deployment location of the tags, if tags were not deployed near towers with detections. We then use `ggplot` to plot the map and tag paths. Here we use the Mercator projection and are colouring the paths by `motusTagID`, including a point for where the tag was deployed:

```
# set limits to map based on locations of detections, ensuring they include the
# deployment locations
xmin <- min(df.tmp$recvLon, na.rm = TRUE) - 2
```

```
xmax <- max(df.tmp$recvLon, na.rm = TRUE) + 2
ymin <- min(df.tmp$recvLat, na.rm = TRUE) - 1
ymax <- max(df.tmp$recvLat, na.rm = TRUE) + 1

# map
ggplot(data = na.lakes, aes(x = long, y = lat))+
  theme_bw() +
  geom_polygon(data = na.map, aes(x = long, y = lat, group = group),
               colour = "grey", fill = "grey98") +
  geom_polygon(aes(group = group), colour = "grey", fill = "white") +
  coord_map(projection = "mercator", xlim = c(xmin, xmax), ylim = c(ymin, ymax)) +
  labs(x = "", y = "") +
  geom_path(data = df.tmp,
            aes(x = recvLon, y = recvLat,
                group = as.factor(motusTagID), colour = as.factor(motusTagID))) +
  geom_point(data = df.tmp,
             aes(x = tagDepLon, y = tagDepLat), colour = "black", shape = 4) +
  scale_colour_discrete("motusTagID")
```



The functions above provide examples of how you can begin exploring your data

and are by no means exhaustive. The next chapter will cover some common errors and troubleshooting you may encounter while trying to download and use the .motus sql files.

Chapter 7

Vanishing bearings

This chapter was contributed by Ana Morales and Tara Crewe, using data collected by Morbey et al. (2017) (see 1.3), and with financial support from a NSERC CREATE Environmental Innovation Program internship awarded to A.M. through McGill University, with partner Bird Studies Canada (T.C.).

Estimating the orientation of a bird departing a stopover site may be key in certain migration studies. In the context of an automated radio-telemetry study, the estimated departure direction is often called a vanishing bearing (Sjöberg and Nilsson 2015). The ability to use characteristics of signal detections to estimate the vanishing bearing of a migrating animal is particularly important at sites where additional stations are not available to capture the flight path following departure from a stopover site.

Sjöberg and Nilsson (2015) calculated vanishing bearings using ordinary circular statistics to estimate the mean of the receiving antenna bearings over the last 5-10 minutes of detections, with the bearing of each detection weighted by strength of the signal. In other words, every signal detected was a vector pointing in the direction of the receiving antenna, with the length of the vector represented by signal strength. A stronger signal would therefore get a longer vector and a higher weight than a weaker signal.

This chapter will show you how to estimate the departure bearing of your tagged birds using Sjöberg and Nilsson's method. You can modify these scripts to work with your own data, or use the sample data provided. Before you begin, there are a few assumptions and potential weaknesses you should be aware of:

7.1 Things to be aware of

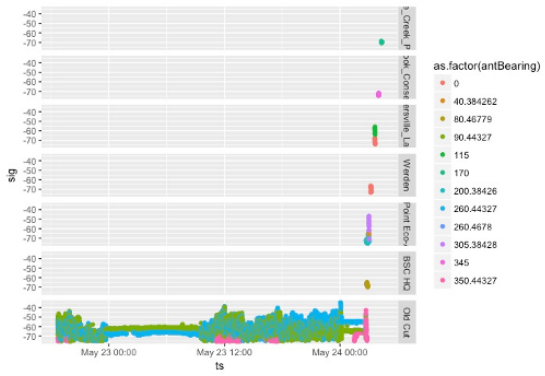
1) **This method assumes that the tagged animal is departing from the general location of the receiving station, and moving radially away from the station**.

It is really important to note that this method **WILL NOT WORK** unless this assumption is met. Because tagged individuals most likely aren't departing from the exact location of the receiver station, they should be considered coarse estimates of departure orientation.

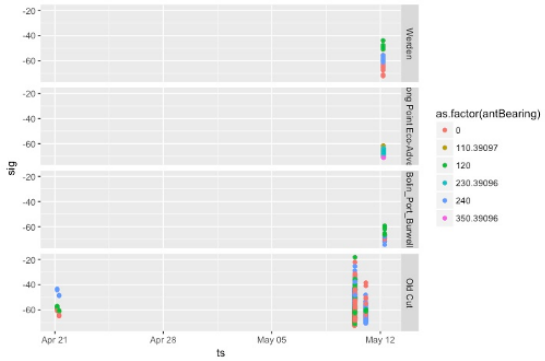
The deviation between estimated vanishing bearings and actual departure orientation that results from birds departing from locations removed from a station is called parallax error. Parallax error can be minimized by analyzing only departures in which we can be confident from signal strength characteristics that an individual departed from fairly close to the receiving station. Using radar data, Sjöberg and Nilsson (2015) estimated parallax error to be +- 10 degrees when the bird departed close to the station.

Manual telemetry can be used to determine the precise location of an individual during stopover. In the absence of known location, a good way to know if an individual departed from the general location of a receiving station is to look at the pattern of detections pre- and post-departure. If an individual departs from near a station, the data will show clear detection patterns by at least two different antennas prior to and during departure. The following plots use data collected for tagged warblers in spring 2014 and 2015 by Morbey et al. (2017), and show examples of a) a tag with a clear departure, and b) a tag that does not show a clear departure; more information on the data can be found in sections 1.3 and 7.2.2 below:

a) **Good departure**: you can see that the bird stayed in the vicinity of the Old Cut station during stopover, as shown by increased signal variation during the day, and decreased signal variation during the night. On the night of departure, the tag shows the usual decrease in activity at night, followed by an increase of activity at around 02:30 am as it departs; the tag stops being detected by the Old Cut's station soon after, and is then detected by other nearby receivers during its departure flight:



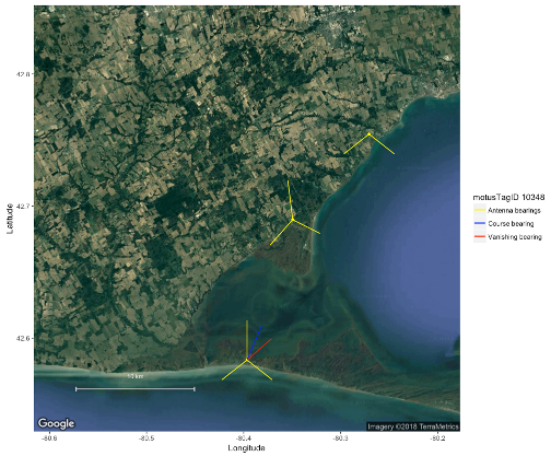
b) **Unclear departure**: this bird was not detected by the Old Cut station for quite a while, and was only detected during an apparent flyover; signal characteristics do not support a clear departure from the station. In this case, we do not know how far the bird was from the station during the flyby. We caution against using examples like this to estimate vanishing bearings:



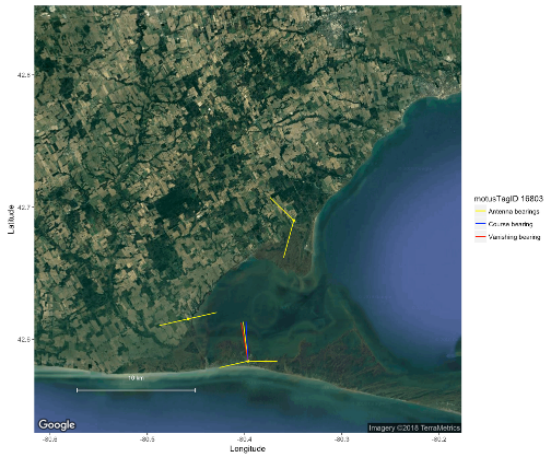
2) **The more closely spaced the antennas on a station, the greater the resolution to estimate an accurate vanishing bearing.**

At the Old Cut field station, three antennas were spaced 120 degrees apart in 2014, and 90 degrees apart in 2015 (facing approximately east, north and west). In both years we calculated a coarse departure bearing using the mean of station-to-station bearings for warblers that were detected by multiple stations during departure. This gave us a known departure direction. We then calculated vanishing bearings to compare.

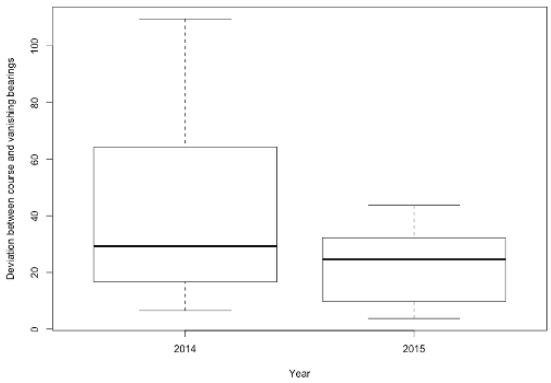
The following is a figure showing the estimated coarse and vanishing bearing for a warbler departing northwards from Old Cut in spring 2014, when the 3 antennas were spaced 120 degrees apart. The red line represents the estimated vanishing bearing of tag ID 277. The blue line represents the average coarse bearing based on detections from other stations (red points):



In the example below, estimated coarse and vanishing bearings are shown for a warbler departing to the north in spring 2015, when Old Cut's antennas were spaced 90 degrees apart. We can see that the estimated vanishing bearing is pretty close to the coarse bearing estimate.



Overall, mean deviation between coarse and vanishing bearings was higher in 2014 (n = 14 departures) when antennas were spaced 120 degrees apart, than in 2015 (n = 12 departures) when antennas were spaced 90 degrees apart:



This suggests that stations with closer spaced antennas yield more accurate vanishing bearings. If the estimation of vanishing bearings is a primary goal of your research, we suggest spacing antennas 60 degrees apart, as in Sjöberg and Nilsson (2015).

3) **For receivers with negative signal strength values, weights need to be normalized**

When calculating weighted circular means, negative weights cannot be used. We therefore recommend normalizing the signal strength values by subtracting the minimum signal strength, and dividing by the difference between min and max signal strength, i.e., using 'r sig.norm = (sig - min(sig))/(max(sig)-min(sig)), eval = FALSE'. Further, we suggest using the min and max signal strength at a station, across **all data** collected by the station, so that the full range of potential signal strength values are used in the normalization equation. We found that normalizing using the min and max signal strength for an individual departure can result in spurious vanishing bearing estimates if the range in signal strengths observed was not large. Using the full range of signal strength values in a project database will avoid this.

7.2 Estimate vanishing bearings: step-by-step

We now walk through how to estimate vanishing bearings in the following steps:

1. Load required R packages
2. Load the data
3. Select individuals that show clear departures from the station
4. Get departure time for each individual
5. Estimate vanishing bearings
6. Plot vanishing bearing on a map

7.2.1 Load the required packages

Follow the instructions in Chapter 2 to install the following packages before loading, if they are not already installed.

```
library(circular)
library(tidyverse)
library(motus)
library(ggplot2)
library(jpeg)
library(ggmap)

#Make sure working on UTC
Sys.setenv(TZ = "UTC")
```

7.2.2 Load the sample data

The sample data used in this chapter includes the detections of three Magnolia Warblers that were tagged at the Old Cut field research station of the Long Point Bird Observatory in Ontario, Canada, by Morbey et al. (2017). The warblers were tagged and released in spring 2015.

The sample data are included in the motusData R package as the "vanishBearing.rda" file. In order to access these data, you will need to first install the motusData package following the instructions in section 2, *prior* to running the code in this chapter:

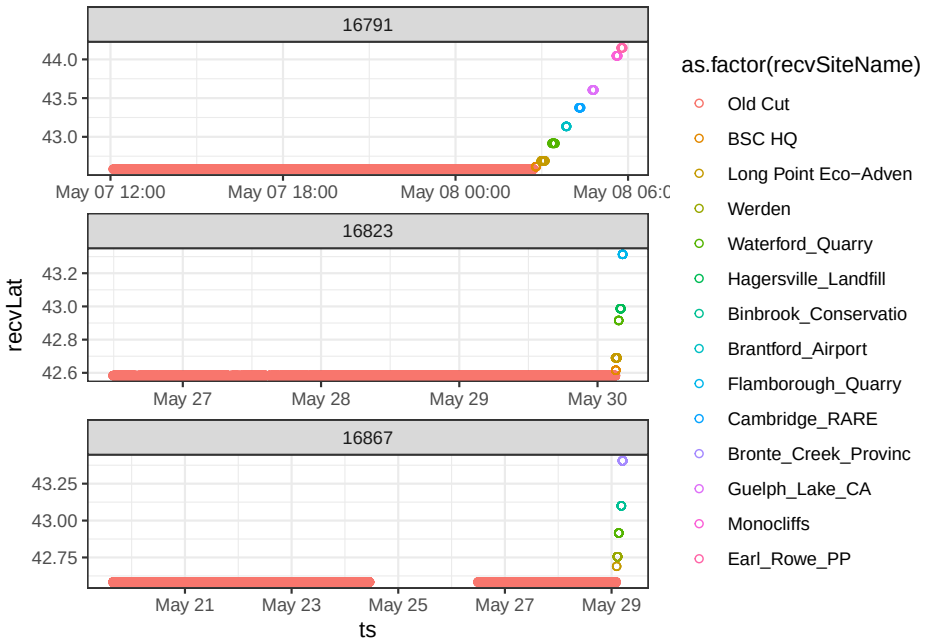
```
# load the motusData package, which contains the sample data for this chapter
library(motusData)

# Load the sample data we provided from 3 individual warblers departing Old Cut
# during the Spring of 2015.

# We also do a couple manipulations here, to reorder the levels of the
# recvSiteName factor, and order the data by ts.

df.vanish <- vanishBearing %>%
  mutate(recvSiteName = reorder(recvSiteName, recvLat),
         motusTagID = as.factor(as.character(motusTagID))) %>% # order sites by latitude
  arrange(ts) # arrange by ts
```

7.2.3 Select individuals that show clear departure detections

First, we subset the data to the individuals that showed clear departures from the stopover site of interest. In fact, the sample data only includes three departures of birds we *know* have clear departures from Old Cut, but we show this step here regardless!

By selecting only clear departures, we make sure that the bird departed from nearby the station, minimizing the potential for parallax error. We look for clear departures by first plotting latitude against time. In this case, we can see the birds departing north past several stations:

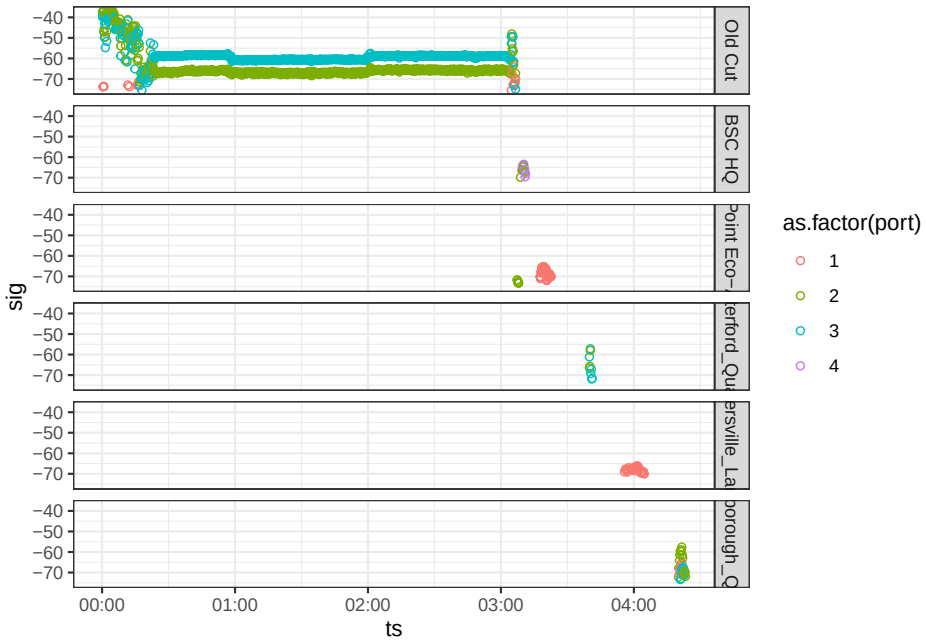
```
ggplot(data = df.vanish, aes(x = ts, y = recvLat, colour = as.factor(recvSiteName))) +
  geom_point(pch = 21) +
  facet_wrap(~ motusTagID, scales = "free", ncol = 1) +
  theme_bw()
```



We also make sure that the bird was being detected by several antennas at once during departure; if detections are on only one antenna, the vanishing bearing will simply be the direction of the lone antenna with detections. We plot signal strength by time for a subset of the data, to show the last few hours of detections of the bird at Old Cut. Let's try this with tag # 16823 from our sample data:

```
ggplot(data = filter(df.vanish, motusTagID == 16823, ts > "2015-05-30 00:00:00"),
       aes(x = ts, y = sig, colour = as.factor(port))) +
  theme_bw() +
```

```
  geom_point(pch = 21) +
  facet_grid(recvSiteName ~ .)
```



We can see the typical detection pattern of a bird departing from a site, shown by a decrease in movement (beginning to roost) at around 00:25 UTC, followed by an increase in activity and departure from the site later in the night after 03:00 UTC. Soon after leaving the site, it gets detected by five other stations further north. Because this bird was detected before and during departure, we can assume it has departed from somewhat close to the Old Cut station.

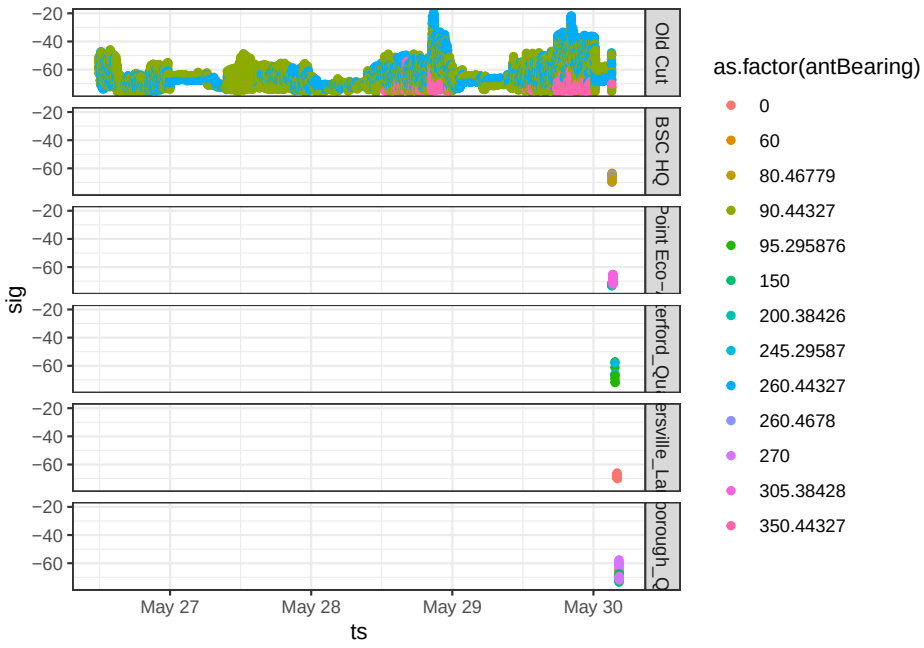
With your own data, look at plots like these for each of your tagged individuals, and for the purpose of vanishing bearings, remove any individuals that don't have a clear departure.

7.2.4 Obtain departure times

In order to estimate vanishing bearings based on the signal strength of departure detections, we must first determine the approximate time that the bird started its departure flight. For now, the best way to do this is by manually going through each set of detections from each bird and obtaining the approximate time by plotting the time versus signal strength.

Continuing with tag 16823, let's find the departure time.

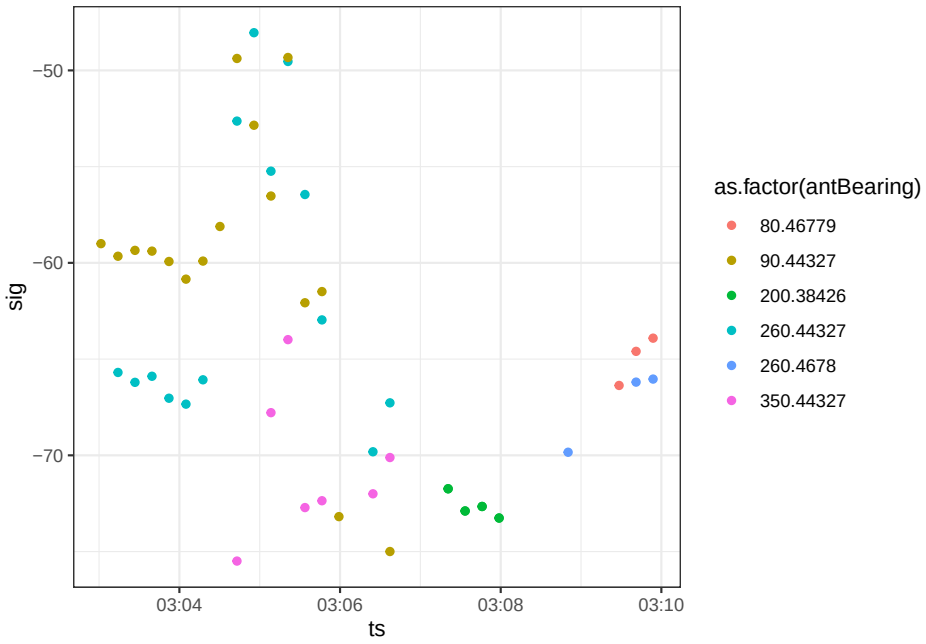
```
ggplot(data = filter(df.vanish, motusTagID == 16823),
       aes(x = ts, y = sig, colour= as.factor(antBearing))) +
  theme_bw() +
  geom_point() +
  facet_grid(recvSiteName ~ .)
```



We can see the complete set of detections of tag 16823 for Old Cut and other stations this bird was detected at post-departure.

We then subset the plot in order to zoom in to the last few minutes of detections at Old Cut, which is where the bird is departing from. By zooming in we can find the exact departure time.

```
ggplot(data = filter(df.vanish,
                     motusTagID == 16823,
                     ts > "2015-05-30 03:03:00",
                     ts < "2015-05-30 03:10:00"),
       aes(x = ts, y = sig, colour= as.factor(antBearing))) +
  theme_bw() +
  geom_point()
```



In this case, we can see an increase in signal strength beginning at about 3:04 for the 90 and 260 degree antennas at Old Cut. These reach a peak in signal strength at about 3:04:59, after which signal strength declines. This suggests that the bird was likely south of the station when it began it's departure - signal strength increased and peaked as it passed through the antenna beams, and then declined again as the bird moved away (north) from the Old Cut station. We choose the peak signal strength, i.e., 3:04:59 as the departure time for this bird, to exclude those detections where the bird was likely moving towards the station; we would not want the 'approaching' station signals to contribute to the vanishing bearing, because it assumes the bird is moving radially away from the station.

We create a dataframe with the departure times of each bird and their motusTagIDs, to use later for data filtering. We add departure times for the other two birds in the sample dataset; if you are keen, you can try the plots above on tags 16897 and 16791 to see how we came up with those departure times:

```
## create dataframe and assign column names
dep.16823 <- as.data.frame(cbind(16823, "2015-05-30 03:04:59"))

## create dataframes for the other two tags:
dep.16867 <- as.data.frame(cbind(16867, "2015-05-29 01:56:00"))
dep.16791 <- as.data.frame(cbind(16791, "2015-05-08 02:41:40"))

## put them all together
df.departTime <- rbind(dep.16823, dep.16867, dep.16791)
```

```
names(df.departTime) <- c("motusTagID", "ts_depart")

## convert time to posixCT using Lubridate functionality
df.departTime <- mutate(df.departTime, ts_depart = ymd_hms(ts_depart))

df.departTime
```

```
##   motusTagID           ts_depart
## 1      16823 2015-05-30 03:04:59
## 2      16867 2015-05-29 01:56:00
## 3      16791 2015-05-08 02:41:40
```

```
## optionally, save to .RDS file to preserve time structure (you could save to
## .csv, but time structure will not be preserved) not run here:

# saveRDS(df.departTime, file = "./data/departureTimes.RDS")
```

7.2.5 Calculate vanishing bearings for individuals with departure times.

Now, we subset our data to one individual to calculate its vanishing bearing.

We will continue with the same individual, `motusTagID` 16823, using all detections after and including our specified departure time:

Now that we have only the post-departure detections for the bird we are interested in, we calculate its vanishing bearing.

First, we normalize the signal strengths, as discussed above, using the minimum and maximum observed signal strength across all data collected by the receiver. If there are differences in the range of signal strengths detected by antennas on a receiver, you might want to instead normalize by antenna. We do not have access to the entire database with the sample data, but we know the minimum and maximum signal strengths detected at the Old Cut stations are -78.0691 and -17.8707, respectively.

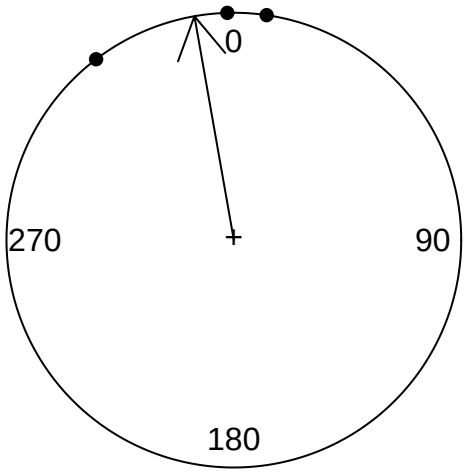
We then calculate a weighted mean of departure angle across the entire departure period using the circular function. The numbers -78.0691 and -17.8707 are the minimum and maximum signal strength values for the Old Cut station. If using your own data, make sure you instead use the min and max signal strength for your station (using the full stations's data, not only from the subset of the tags you are analyzing).

```
## Merge sample data with departure times, subset data, and calculate vanishing bearin

## Note that we use the recvSiteName to specify the departure station of
## interest. Depending on whether the station has moved or changed names with
```

```
## deployments, recvDeployID might be more appropriate.

depart.station <- "Old Cut"
min.sig <- -78.0691 # normally max/min sig comes from the complete raw data for a station
max.sig <- -17.8707

# in this case, right join should drop any individuals that don't have departure
# times in df.departTime
df.vanishBearing <- right_join(df.vanish, df.departTime, by = "motusTagID") %>%
  filter(ts >= ts_depart,
         recvSiteName == depart.station) %>%
  distinct() %>%
  mutate(sig.norm = (sig - (min.sig))/((max.sig)-(min.sig)),
         circ.bear = circular(antBearing,
                              type = c("angles"),
                              units = c("degrees"),
                              rotation = c("clock"))) %>%
  group_by(motusTagID, recvSiteName, recvLat, recvLon) %>%
  summarise(vanish.bearing = weighted.mean(circ.bear, sig.norm, na.rm = FALSE,
                                           control.circular = list(type = "angles",
                                                                   units = "degrees",
                                                                   template = "none",
                                                                   rotation = "clock")),
            minutes.used = as.duration(min(ts) %--% max(ts))) %>%
  as.data.frame()
```

The resulting `df.vanish` dataframe contains the `motusTagID`, the vanishing bearing of the tag, the time in seconds/minutes used to estimate the bearing, and the receiver's name and coordinates. We can make a circular plot with points for the individual vanishing bearings and an arrow for the mean bearing as follows:

```
# if you have many bearings/points, can use stack = TRUE
plot.circular(df.vanishBearing$vanish.bearing, zero = pi/2)
arrows.circular(mean(df.vanishBearing$vanish.bearing), zero = pi/2)
```



7.2.6 Plot the vanishing bearings on a map

Mapping your vanishing bearing(s) with `ggmap` can also be a great way to visualize the departure direction of your bird(s). Here we use Stamen maps to show the stations and orientations of antennas that detected the bird during departure, along with the orientation of the vanishing bearing. For information on using `ggmap` to create Google Maps, see Appendix B in section B.4.

First, pick a tag to map:

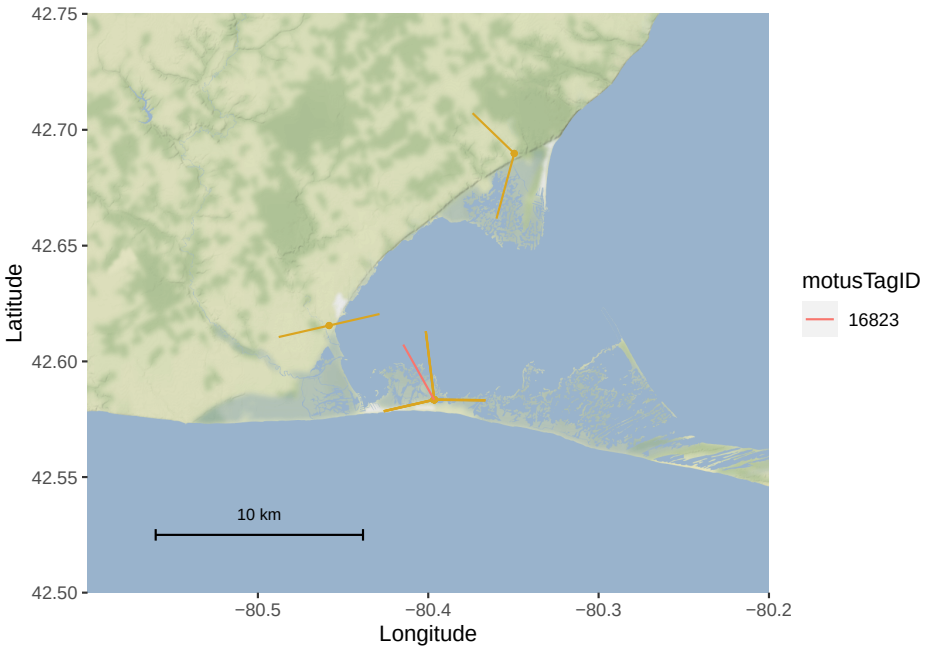
```
tagID <- 16823
```

Then, create a map with a specified map centre, maptype ("terrain", "roadmap", "satellite", or "hybrid"), and level of zoom (integer for zoom 3-21, 3 being continent level, 10 being city-scale). We add a yellow point for the stations with detections, yellow lines to represent antenna bearings with detections, and a red line for the vanishing bearing:

```
## First we obtain a map of our location of interest, in this case Old Cut
map.OC <-  get_stamenmap(bbox = c(left = -80.6, right = -80.2, bottom = 42.5, top = 42
                  maptype = "terrain-background",
                  zoom = 12,
                  color = "color")

## Do the following to make a scale bar
bb <- attr(map.OC,"bb")
sbar <- data.frame(lon.start = c(bb$ll.lon + 0.1*(bb$ur.lon - bb$ll.lon)),
                  lon.end = c(bb$ll.lon + 0.25*(bb$ur.lon - bb$ll.lon)),
                  lat.start = c(bb$ll.lat + 0.1*(bb$ur.lat - bb$ll.lat)),
                  lat.end = c(bb$ll.lat + 0.1*(bb$ur.lat - bb$ll.lat)))
```

```
sbar$distance <- geosphere::distVincentyEllipsoid(c(sbar$lon.start,sbar$lat.start),
                                                  c(sbar$lon.end,sbar$lat.end))
scalebar.length <- 10
sbar$lon.end <- sbar$lon.start +
  ((sbar$lon.end-sbar$lon.start)/sbar$distance)*scalebar.length*1000
ptspermm <- 2.83464567  # need this because geom_text uses mm, and themes use pts.

## To map antenna bearings:
## Create a station dataframe with antenna bearings for all antennas with
## detections for the tag of interest
df.stations <- df.vanish %>%
  filter(motusTagID == tagID) %>%
  select(recvSiteName, antBearing, port, recvLon, recvLat) %>%
  distinct()

# determines length of the vectors for antenna bearings and vanishing bearing lines
arr.sc <- 0.03
rad <- function(x) {x * pi/180}

## Now we make the map

ggmap(map.OC) +
    geom_point(data = df.stations, aes(x = recvLon, y = recvLat), size = 1,
               colour = "goldenrod") +
    # Add antenna bearings
    geom_segment(data = df.stations,
                 aes(x = recvLon, xend = recvLon + (sin(rad(antBearing))*arr.sc),
                     y = recvLat, yend = recvLat + (cos(rad(antBearing))*arr.sc)),
                 colour = "goldenrod") +

    # add vanishing bearings
    geom_segment(data = filter(df.vanishBearing, motusTagID == tagID),
                 aes(x = recvLon, xend = recvLon + (sin(rad(vanish.bearing))*arr.sc),
                     y = recvLat, yend = recvLat + (cos(rad(vanish.bearing))*arr.sc),
                     colour = motusTagID)) +

    # Add scale bar
    geom_segment(data = sbar,
                 aes(x = lon.start, xend = lon.end,
                     y = lat.start, yend = lat.end),
                 col = "black",
                 arrow = arrow(angle = 90, length = unit(0.1, "cm"),
                               ends = "both", type = "open")) +
    geom_text(data = sbar,
              aes(x = (lon.start + lon.end)/2,
```

```
                  y = lat.start + 0.025*(bb$ur.lat - bb$ll.lat),
                  label = paste(format(scalebar.length), 'km')),
              hjust = 0.5,
              vjust = 0,
              size = 8/ptspermm, col = "black")  +
    labs(x = "Longitude", y = "Latitude")
```



Lengths of the yellow and red lines do not represent antenna range or exact path of the bird. For this departure, we have detections on two other stations which corroborate the vanishing bearing.

If desired, you can print the map to file as follows:

```
tiff(file = paste(tagID, "vanishBearing.tiff", sep=""))
print(out.map)
dev.off()
```

7.3 Literature Cited

Morbey, Y.E., K.A. Jonasson, J.E. Deakin, A.T. Beauchamp, and C.G. Guglielmo. 2017. Studies of migratory birds and bats in southern Ontario, 2014-2017 (Projects #20 and #50). Data accessed from the Motus Wildlife Tracking System. Bird Studies Canada. Available: http://www.motus-wts.org. Accessed: May 1, 2018.

Sjöberg, S., and C. Nilsson. 2015. Nocturnal migratory songbirds adjust their travelling direction aloft: evidence from a radiotelemetry and radar study. Biology Letters 11:20150337.

Chapter 8

Advanced modelling and analysis

Numerous investigators have been developing more advanced tools for analyzing movement and behavioural data using Motus such as triangulation, position error and probability of detection, state-space movement modelling, and calculation of tag life histories (stopover duration, movement between subsequent sites, etc.). Aspects of some of these processes have been touched on in previous chapters. For investigators wishing to delve more into these subjects in more detail see:

Baldwin, Justin W., Katie Leap, John T. Finn, and Jennifer R. Smetzer. "Bayesian State-Space Models Reveal Unobserved off-Shore Nocturnal Migration from Motus Data." Ecological Modelling 386 (October 24, 2018): 38–46. Researchers propose new biologically informed Bayesian state-space models for animal movement in JAGS that include informed assumptions about behaviour. Building from the bsam package in R a simple localization routine predicts bird location and spatial uncertainty.

Baldwin, Justin. "Modelling Bird Migration with Motus Data and Bayesian State-Space Models." University of Massachusetts Amherst, 2017. New biologically informed Bayesian state-space models are proposed for animal movements in JAGS that include informed assumptions about behaviour. The models are evaluated using a simulation study, and are then applied by employing a localization routine on a Motus data set to estimate unobserved locations and behaviours.

Janaswamy, Ramakrishna, Pamela H. Loring, and James D. McLaren. "A State Space Technique for Wildlife Position Estimation Using Non-Simultaneous Signal Strength Measurements." ArXiv.org, May 28, 2018. Combining a movement model to ensure biologically-consistent trajectories in three-dimensions, and an observation model to account for the effect of range, altitude, and bearing angle

on the received signal strength, this novel state-space technique can estimate the location of airborne movements of VHF tags within the Motus array.

As more advanced analysis and modelling tools become available they will be posted here.

We strongly encourage participants to offer sample scripts and functions that can be integrated into this book.

Appendix A

Appendix - `alltags` and `alltagsGPS` structure

The following variables are included in `alltags` and `alltagsGPS` view in the SQLite file (note that the final three `gps` related fields are only available in `alltagsGPS`).

Field	Description
hitID	unique Motus ID for this tag detection
runID	unique Motus ID for the run this detection belongs to
batchID	unique Motus ID for the processing batch this detection came from
ts	timestamp, in seconds since 1 Jan, 1970 UTC; precision: 0.1 ms (SG); 2.5 ms (Lotek).
sig	signal strength in "native units"; for SG: dB (max) (logarithmic, 0 = max possible, -10 = 0.1 * max, etc.); for Lotek: raw value (0...255)
sigsd	std. dev. in signal strength among pulses in this burst. SG Only; NA for Lotek
noise	estimate of background radio noise when tag detected, in dB (max) for SG; NA for Lotek
freq	frequency offset from antenna listening frequency, in kHz for SG only; NA for Lotek
freqsd	std. dev. of freq offset among pulses in this burst, in kHz. Values larger than 0.1 kHz suggest a bogus detection. SG only; NA for Lotek.
slop	total absolute difference (milliseconds) in inter-pulse intervals for this burst between registration and detection. SG only; NA for Lotek

Field	Description
burstSlop	signed difference (seconds) between detection and registration burst intervals. This is always 0 for the first burst in a run (see posInRun)
done	logical: is run finished?
motusTagID	Motus tag ID - unique to each individual tag registered
ambigID	ambiguous ID assigned to ambiguous tags
port	the port number that the detection occurred on
runLen	number of tag bursts in the current run; constant for all records having the same runID
bootnum	boot session of receiver for SG; NA for Lotek
tagProjectID	ID of the project that manages this tag.
mfgID	manufacturer ID
tagType	
codeSet	for coded tags, the name of the codeset (e.g. 'Lotek-3')
mfg	tag manufacturer
tagModel	manufacturer's model name for a tag (e.g. 'NTQB-3-2')
tagLifespan	estimated lifespan of tag (days)
nomFreq	nominal tag frequency (MOTUS: Nominal frequency receiver was tuned to, in Mhz. This really only applies to SG, where we usually tune 4 kHz below the nominal tag frequency. So in that case, antFreq = 166.376 while nomFreq = 166.380
tagBI	burst interval of tag, in seconds (e.g., 5.8984)
pulseLen	tag pulse length (milliseconds), if applicable. This value is only assigned based on the sample recording of the tag.
tagDeployID	Motus tag deployment ID
speciesID	unique numeric Motus ID (integer) for the species on which the tag was deployed
markerNumber	number for any additional marker placed on organism (e.g., bird band #)
markerType	type of additional marker (e.g. metal band)
tagDeployStart	tag deployment start date
tagDeployEnd	tag deployment end date
tagDeployLat	latitude of tag deployment, in decimal degrees N - negative values for Southern hemisphere
tagDeployLon	longitude of tag deployment, in decimal degrees E - negative values for Western hemisphere
tagDeployAlt	altitude of tag deployment, in meters ASL
tagDepComments	additional comments or unclassified metadata for tag (often in JSON format)
fullID	full tag ID as PROJECT#MFGID:BI@NOMFREQ. Not necessarily unique over time. See motusTagID for a unique tag

Field	Description
deviceID	Motus device ID associated with the receiver serial number
recvDeployID	Receiver deployment ID
recvDeployLat	latitude of receiver deployment, in decimal degrees N - negative values for Southern hemisphere
recvDeployLon	longitude of receiver deployment, in decimal degrees E - negative values for Western hemisphere
recvDeployAlt	altitude of receiver deployment, in meters ASL
recv	serial number of the receiver; e.g., SG-1234BBBK5678 or Lotek-12345
recvDeployName	name assigned to the receiver deployment by the project manager
recvSiteName	name assigned to a site by the project manager (e.g. location name). The same label can be used for multiple deployments.
isRecvMobile	logical; whether the sensor is deployed on a mobile platform (eg. a ship)
recvProjID	unique (numeric) ID of the project that deployed this receiver (e.g., 8)
antType	character; antenna type, e.g. “9-element Yagi”, “Omni”
antBearing	numeric; compass direction antenna main axis is pointing at (degrees clockwise from local magnetic North 0-360)
antHeight	numeric; height (meters) of antenna main axis above ground
speciesEN	species English (common) name
speciesFR	species French (common) name
speciesSci	species scientific name
speciesGroup	species group, e.g., BIRDS, BATS
tagProjName	short label of project that deployed the tag, e.g., “HolbSESA”
recvProjName	short label of project that deployed the receiver e.g., “HolbSESA”
gpsLat	latitude of receiver GPS location at time of writing the hourly detections file (degrees North)
gpsLon	longitude of receiver GPS location at time of writing the hourly detections file (degrees East)
gpsAlt	altitude of receiver GPS at time of writing the hourly detections file (meters)

Appendix B

Appendix - Troubleshooting

As a first step, always ensure you are using the latest version of the motus package (see Appendix C.1), and you have all required packages installed, loaded, and up to date (see Chapter 2).

While attempting to download data with the motus package, you may encounter errors, many of which are likely due to an interrupted connection. **Always ensure you are connected to the internet when using the `tagme()` function with `update = TRUE`.** Most issues can be solved by either logging out of the motus package, or by restarting R and resuming the download using `tagme()`. If errors persist and you are unable to download your data, the server may be temporarily offline. Please contact Motus with any concerns at motus@birdscanada.org.

B.1 General Problems

Many, *many*, problems arise from conflicts between R packages which may be out of date. If you have a problem that you can't seem to resolve, try the following steps in order (stopping when the problem goes away):

1. Update `motus` and packages that `motus` depends on. (You may first need to install the `remotes` package). **Re-start R**

```
remotes::update_packages("motus")
```

2. Update all your packages. **Re-start R**

```
remotes::update_packages()
```

3. Update R https://cran.r-project.org/. (You may have to reinstall packages)

B.2 Logging out of motus

```
motusLogout()
```

B.3 Resume data download

To resume your data download, run `tagme()` again, but do not include `new = TRUE`:

```
tagme(project.num, update = TRUE, dir = ...)
```

B.4 Google Maps

As of October 16, 2018 recent updates require the use of a Google key to access google maps. To obtain an access key, you must be a registered Google user with **up to date billing information**, however you do not have to pay for the service (for the first little while at least). To obtain a key:

1. login to the Google Cloud Platform.
2. If you do not already have a project then create one.
3. Check that you have current billing information - you will not be charged but it must be present and up to date.
4. Under the navigation menu on the left, click APIs & Services > Credentials, then click Create credentials > API key.
5. You may need to enable Google Maps Static API. You can do this through the navigation menu in the upper left corner, and selecting APIs & Services > library, choosing "Google Maps Static API" and clicking "Enable".

Full details are listed under "Detailed Guide" here. Note that you may have to enable Google Maps Static API. For troubleshooting see here and here.

Once you have your access key, you'll need to provide it with the call `register_google()`, **each time you start a new R session you will be required to enter your key.**

Then you can create Google maps with the `ggmap` package using the `get_googlemap()` function and specifying the lat/lon center of your map (as opposed to bounding box as with stamen maps).

B.5 Common problems and solutions:

B.5.1 I cannot access Project 176 / I download 0 records from Project 176

Remember that project 176 is *only* accessible with the username and password: `motus.sample`.

B.5.2 I get the message "Auto-disconnecting SQLiteConnection" one or multiple times after using tagme()

If this occurs after data download has finished, this message can be ignored. If it occurs during an active download, the connection will usually be maintained and the download will continue. However if the download stops, simply run `tagme()` again. If that does not work, we suggest logging out of the motus package or restarting R (see sections B.2 and B.3).

B.5.3 I get an "Internal Server Error" message when using tagme(..., update = TRUE)

If you get this message while updating your .motus file, use `tagme()` again to continue the download.

B.5.4 I get an "Error: Forbidden" message when using tagme()

This error may occur if you are attempting to download multiple projects simultaneously from the same user account. If you get this error, please logout of the motus package, and try `tagme()` again (see sections B.2 and B.3).

B.5.5 I get an error "Object 'xxxx' not found", referring to a table or field name, or some of your examples in the book do not work.

Be sure to start the steps from the top of the chapter and run them in sequential order. Another possibility is that your .motus database hasn't been updated to support the latest version of the motus package.

To ensure that your .motus file is up-to-date with the motus package:

```
sql.motus <- tagme(project.num, dir= ...)
checkVersion(sql.motus)
```

To correct any warnings, you should follow these steps:

1. download the latest version of the motus package (refer to Chapter 2).
2. terminate and restart your R session.
3. load the motus library using `library(motus)` in your R console.
4. load your sqlite file. Look for notes on the console indicating that your database is being updated.
5. check the version again.

```
library(motus)
sql <- tagme(project.num, dir= ...)
checkVersion(sql)
```

B.5.6 I get an error `Error in rsqlite_connect(dbname, loadable.extensions, flags, vfs) : Could not connect to database: unable to open database file` when attempting to run `tagme()`

If you get this message, it's likely that you're attempting a new download or update to a nonexistant directory. The directory is specified in the **`dir = ""`** command of the `tagme()` function. If the directory is not specified, files will be saved to your working directory. Use `getwd()` to determine your current working directory. Use `setwd()` to set a new working directory. To specify a location to save files from your working directory use "./" followed by the file path.

```
getwd() # show working directory, in this case it's "C:/Documents"

# downloads data to your working directory
tagme(proj.num, new = TRUE, update = TRUE)

# downloads data to the data folder within your working directory
# ie. the file path C:/Documents/data
tagme(proj.num, new = TRUE, update = TRUE, dir = "./data/")

# downloads data to the file path C:/Downloads
tagme(proj.num, new = TRUE, update = TRUE, dir = "C:/Downloads")
```

B.5.7 I see GPS coordinates of "0" or "999"

These values are recorded from the GPS unit in the field. Values like these should be ignored, you can replace these with `NA`, or insert the

`recvDeployLat`/`recvDeplOyLon` values which are taken from receiver deployment metadata entered by users.

```
## to replace gpsLat/gpsLon values of NA, 0, or 999 with that of receiver deployment metadata in
df.alltags <- tbl.alltags %>%
  mutate(recvLat = if_else((is.na(gpsLat)|gpsLat == 0|gpsLat == 999),
                           recvDeployLat, gpsLat),
         recvLon = if_else((is.na(gpsLon)|gpsLon == 0|gpsLon == 999),
                           recvDeployLon, gpsLon)
  collect() %>% as.data.frame
```

B.5.8 I see port with a value of -1

Port "-1" represents the "A1+A2+A3+A4" compound antenna which is sometimes reported in Lotek .DTA file detections. This likely means the receiver has combined the signal from 4 antennas to detect the tag.

B.5.9 I get the error `Error in rsqlite_fetch(res@ptr, n = n) : database disk image is malformed`

This is likely due to corrupt files which can occur during download. The easiest solution is to delete your current .motus file and download from scratch.

Of course, there is always the possibility that the book contains errors! If this does not work, please contact motus@birdscanada.org.

Appendix C

Appendix - `motus` - Summary and plotting functions

The `motus` R package offers functions that work with .motus data to do common computations, summaries and plots. This appendix outlines these functions and provides examples on function use. Many of these functions work with both `tbl` and `data.frame` formats, however some require the data to be in sql format as specified below. Detailed instructions on accessing and formatting data are available in Chapter 3. The examples throughout this chapter work with the sample data which can be accessed and converted to various formats through the following code:

```
# download and access sample data in sql format
# username: motus.sample, password: motus.sample
sql.motus <- tagme(176, new = TRUE, update = TRUE, dir = "./data")

# extract "alltags"" table from sql file "sql.motus"
tbl.alltags <- tbl(sql.motus, "alltagsGPS")

## convert the tbl "tbl.alltags" to a data.frame called "df.alltags"
df.alltags <- tbl.alltags %>%
  collect() %>%
  as.data.frame()
```

You can access the function help pages using '?sunRiseSet in the R console. Or view the underlying function code like this:

```
sunRiseSet
```

```
## function (data, lat = "recvDeployLat", lon = "recvDeployLon",
##     ts = "ts")
## {
##     data <- data %>% dplyr::collect() %>% as.data.frame()
##     data$ts <- lubridate::as_datetime(data$ts, tz = "UTC")
##     cols <- c(lat, lon, ts)
##     loc_na <- data[!stats::complete.cases(data[cols]), ]
##     loc <- data[stats::complete.cases(data[cols]), ]
##     if (nrow(loc) == 0)
##         stop("No data with coordinates '", lat, "' and '", lon,
##             "'", call. = FALSE)
##     loc$sunrise <- maptools::sunriset(as.matrix(dplyr::select(loc,
##         lon, lat)), loc$ts, POSIXct.out = T, direction = "sunrise")$time
##     loc$sunset <- maptools::sunriset(as.matrix(dplyr::select(loc,
##         lon, lat)), loc$ts, POSIXct.out = T, direction = "sunset")$time
##     data <- merge(loc, loc_na, all = TRUE)
##     return(data)
## }
## <bytecode: 0x5653b57a1ab0>
## <environment: namespace:motus>
```

C.1 checkVersion

C.1.1 Description

When you call the `tagme()` function to load the sqlite database, there is a process that will verify that your database has the version matching the most current version of the motus package and store the version in a new table called `admInfo`. Over time, changes will be made that require adding new tables, views or fields to the database. The following call will check that your database has been updated to the version matching the current version of the motus package. Refer to Appendix B if this call returns a warning; if you do not have the most recent version, see Chapter 2 to update `motus`.

C.1.2 Arguments

sql.motus an sqlite database of .motus data downloaded using `tagme()`

C.1.3 Example

```
checkVersion(sql.motus)
```

C.2 sunRiseSet

C.2.1 Description

Creates and adds a sunrise and sunset variable to a data.frame containing latitude, longitude, and a date/time as POSIXct or numeric.

C.2.2 Arguments

data can be either a selected table from .motus detection data e.g. **alltags**, or a data.frame of detection data including at a minimum variables for date/time, latitude, and longitude
lat variable with latitude values, defaults to recvDeployLat
lon variable with longitude values, defaults to recvDeployLon
ts variable with time in UTC as numeric or POSIXct, defaults to ts

C.2.3 Example

Add sunrise/sunset variables to the **alltags** data.frame

```
alltags.df.sun <- sunRiseSet(df.alltags)
head(alltags.df.sun)
```

```
##    hitID runID batchID                  ts tsCorrected sig sigsd noise freq freqsd
## 1  45107  8886      53 2015-10-26 11:19:49  1445858390  52     0   -96    4      0
## 2  45108  8886      53 2015-10-26 11:20:28  1445858429  54     0   -96    4      0
## 3  45109  8886      53 2015-10-26 11:21:17  1445858477  55     0   -96    4      0
## 4  45110  8886      53 2015-10-26 11:21:55  1445858516  52     0   -96    4      0
## 5  45111  8886      53 2015-10-26 11:22:44  1445858564  49     0   -96    4      0
## 6 199885 23305      64 2015-10-26 11:12:04  1445857924  33     0   -96    4      0
##   codeSet   mfg tagModel tagLifespan nomFreq  tagBI pulseLen tagDeployID speciesID
## 1  Lotek4 Lotek NTQB-3-2          NA  166.38 9.6971      2.5        1839      4670
## 2  Lotek4 Lotek NTQB-3-2          NA  166.38 9.6971      2.5        1839      4670
## 3  Lotek4 Lotek NTQB-3-2          NA  166.38 9.6971      2.5        1839      4670
## 4  Lotek4 Lotek NTQB-3-2          NA  166.38 9.6971      2.5        1839      4670
## 5  Lotek4 Lotek NTQB-3-2          NA  166.38 9.6971      2.5        1839      4670
## 6  Lotek4 Lotek NTQB-3-2          NA  166.38 9.6971      2.5        1839      4670
##
## 1 {"ageID":"HY","bill":36.5,"blood":"Y","country":"Canada","culmen":36.5,"fatScore"
## 2 {"ageID":"HY","bill":36.5,"blood":"Y","country":"Canada","culmen":36.5,"fatScore"
## 3 {"ageID":"HY","bill":36.5,"blood":"Y","country":"Canada","culmen":36.5,"fatScore"
## 4 {"ageID":"HY","bill":36.5,"blood":"Y","country":"Canada","culmen":36.5,"fatScore"
## 5 {"ageID":"HY","bill":36.5,"blood":"Y","country":"Canada","culmen":36.5,"fatScore"
## 6 {"ageID":"HY","bill":36.5,"blood":"Y","country":"Canada","culmen":36.5,"fatScore"
```

```
##                              fullID deviceID recvDeployID recvDeployLat recvDeployLon recvDep
## 1 SampleData#378:9.7@166.38(M.16047)      486         2510      42.60699     -72.71657
## 2 SampleData#378:9.7@166.38(M.16047)      486         2510      42.60699     -72.71657
## 3 SampleData#378:9.7@166.38(M.16047)      486         2510      42.60699     -72.71657
## 4 SampleData#378:9.7@166.38(M.16047)      486         2510      42.60699     -72.71657
## 5 SampleData#378:9.7@166.38(M.16047)      486         2510      42.60699     -72.71657
## 6 SampleData#378:9.7@166.38(M.16047)      515         2512      42.68067     -72.47392
##   recvUtcOffset antType antBearing antHeight speciesEN          speciesFR       speciesSci spe
## 1            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris canutus
## 2            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris canutus
## 3            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris canutus
## 4            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris canutus
## 5            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris canutus
## 6            NA  yagi-9        243        NA  Red Knot Bécasseau maubèche Calidris canutus
##                 sunrise              sunset
## 1 2015-10-26 11:16:49 2015-10-26 21:52:11
## 2 2015-10-26 11:16:49 2015-10-26 21:52:11
## 3 2015-10-26 11:16:49 2015-10-26 21:52:11
## 4 2015-10-26 11:16:49 2015-10-26 21:52:11
## 5 2015-10-26 11:16:49 2015-10-26 21:52:11
## 6 2015-10-26 11:15:58 2015-10-26 21:51:06
```

C.3 plotAllTagsCoord

C.3.1 Description

Plot latitude/longitude vs time (UTC rounded to the hour) for each tag using .motus detection data. Coordinate is by default taken from a receivers GPS latitude recordings.

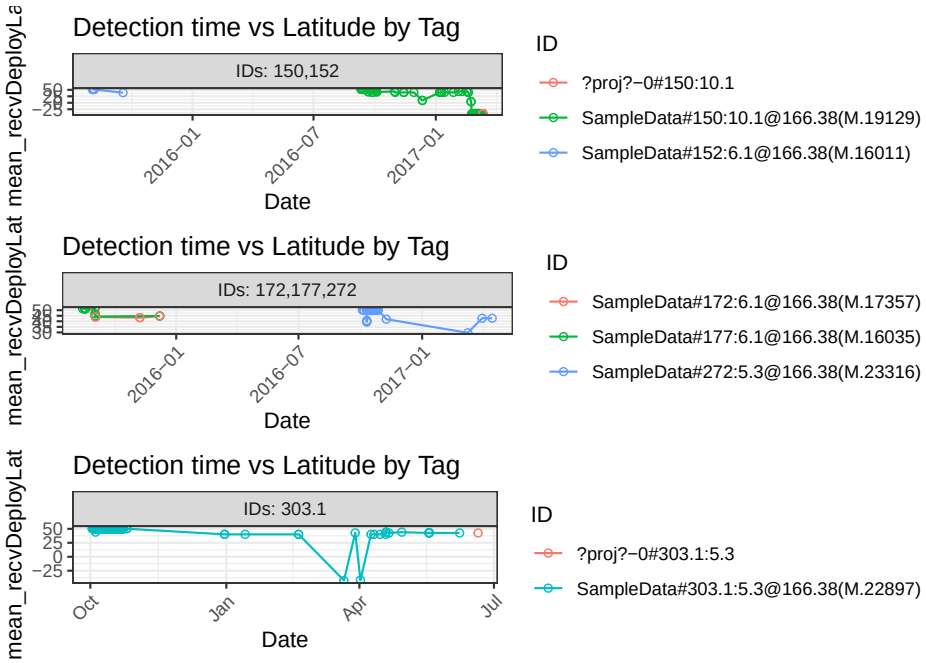
C.3.2 Arguments

data a selected table from .motus detection data, e.g. **alltags**, or a data.frame of detection data including at a minimum variables for date/time, and either latitude or longitude
tagsPerPanel number of tags in each panel of the plot, by default this is 5
coordinate variable name from which to obtain location values, by default it is set to **recvDeployLat**
ts variable for a date/time object as numeric or POSIXct, defaults to **ts**
recvDepName variable consisting of receiver deployment name
fullID variable consisting of a tag fullID
mfgID variable consisting of a tags manufacturer ID

C.3.3 Example

Plot select tags from `tbl.alltags` with 3 tags per panel

```
plotAllTagsCoord(
  filter(tbl.alltags, motusTagID %in% c(19129, 16011, 17357, 16035, 22897, 23316)),
  tagsPerPanel = 3)
```



C.4 `plotAllTagsSite`

C.4.1 Description

Plot latitude/longitude vs time (UTC rounded to the hour) for each tag using .motus detection data. Coordinate is by default taken from a receivers GPS latitude recordings.

C.4.2 Arguments

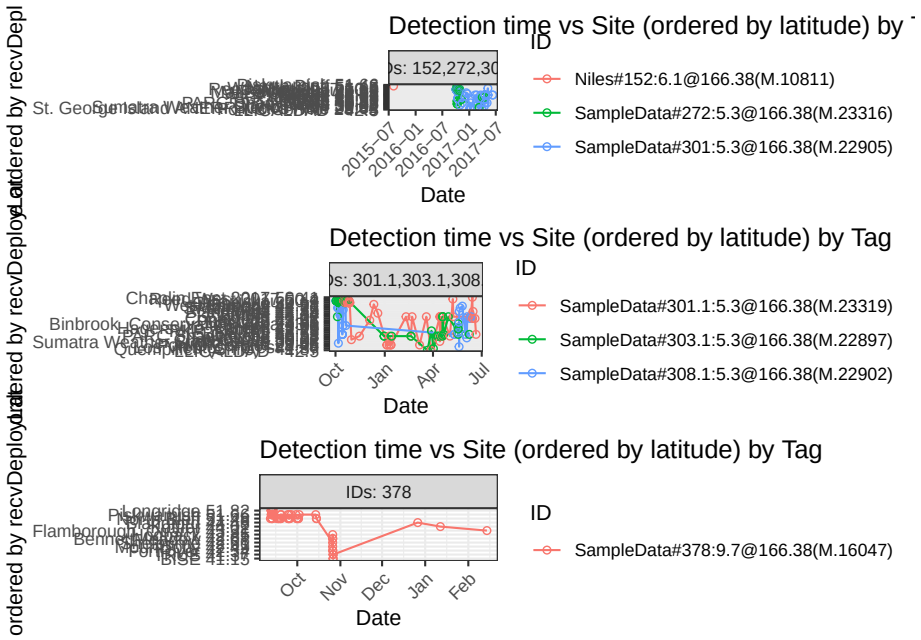
`data` a selected table from .motus detection data, e.g. **`alltags`**, or a data.frame of detection data including at a minimum variables for date/time, and either latitude or longitude
`tagsPerPanel` number of tags in each panel of the plot, by default this is 5

coordinate variable name from which to obtain location values, by default it is set to **recvDeployLat**
ts variable for a date/time object as numeric or POSIXct, defaults to **ts**
recvDepName variable consisting of receiver deployment name
fullID variable consisting of a tag fullID
mfgID variable consisting of a tags manufacturer ID

C.4.3 Example

Plot tbl file **tbl.alltags** using gpsLat and 3 tags per panel for select species Red Knot

```
plotAllTagsSite(filter(tbl.alltags, speciesEN == "Red Knot"),
                coordinate = "recvDeployLat",
                tagsPerPanel = 3)
```



C.5 plotDailySiteSum

C.5.1 Description

Plots total number of detections across all tags, and total number of tags detected per day for a specified site. Depends on siteSumDaily function.

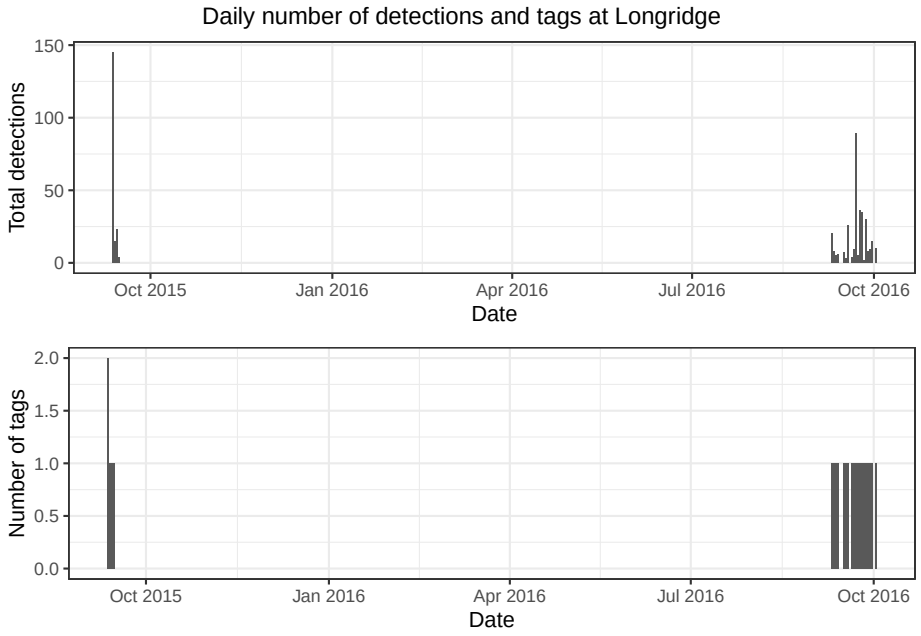
C.5.2 Arguments

data a selected table from .motus data, e.g. "alltags", or a data.frame of detection data including at a minimum variables for motusTagID, sig, recvDepName, ts
motusTagID variable consisting of a motus tag ID
sig variable consisting a signal strength variable
recvDepName variable consisting of receiver deployment name
ts variable for a date/time object as numeric or POSIXct, defaults to ts

C.5.3 Example

Plot of all tag detections at site Longridge using data.frame df.alltags

```
plotDailySiteSum(df.alltags, recvDeployName = "Longridge")
```



C.6 plotRouteMap

C.6.1 Description

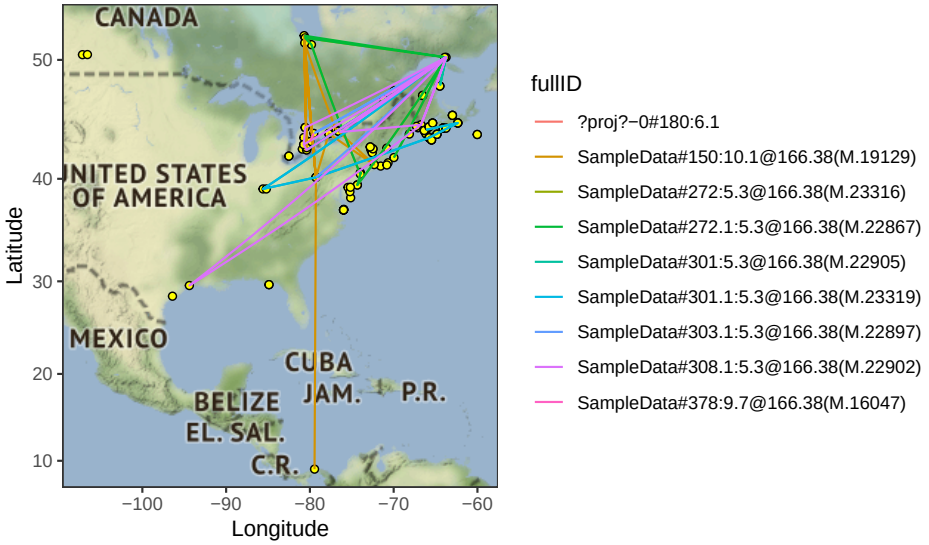
ggmap map of routes of Motus tag detections coloured by motusTagID. User defines a date range to show points for receivers that were operational at some point during specified date range. ### Arguments **data** a .motus sql file
maptype map type to display, can be: "terrain" , "terrain-background", "toner",

"watercolor", etc. **lat** top and bottom latitude bounds. If NULL (default) this is calculated from the data **lon** left and right longitude bounds. If NULL (default) this is calculated from the data **zoom** integer for zoom 3-21, 3 being continent level, 10 being city-scale
recvStart start date for date range of active receivers
recvEnd end date for date range of active receivers

C.6.2 Example

Plot routemap of all detection data, with "terrain" maptype, and receivers active between 2016-01-01 and 2017-01-01

```
plotRouteMap(sql.motus,
             maptype = "terrain",
             recvStart = "2016-01-01", recvEnd = "2016-12-31")
```



C.7 plotSite

C.7.1 Description

C.7.2 Arguments

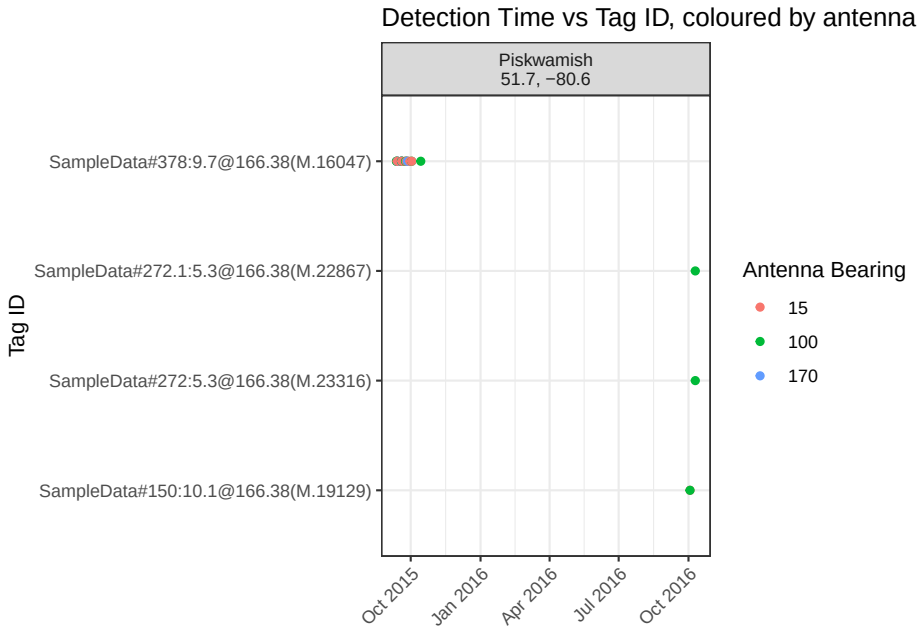
data a selected table from .motus data, e.g. **alltags**, or a data.frame of detection data including at a minimum variables for ts, **antBearing**, **fullID**, **recvDepName**
ts variable for a date/time object as numeric or POSIXct, defaults to ts
antBearing variable consisting antenna bearing variable

fullID variable consisting of a tag fullID
recvDepName variable consisting of receiver deployment name

C.7.3 Example

Plot only detections at a specific site; Piskwamish for data.frame `df.alltags`

```
plotSite(filter(df.alltags, recvDeployName == "Piskwamish"))
```



C.8 plotSiteSig

C.8.1 Description

Plot signal strength vs time for all tags detected at a specified site, coloured by antenna

C.8.2 Arguments

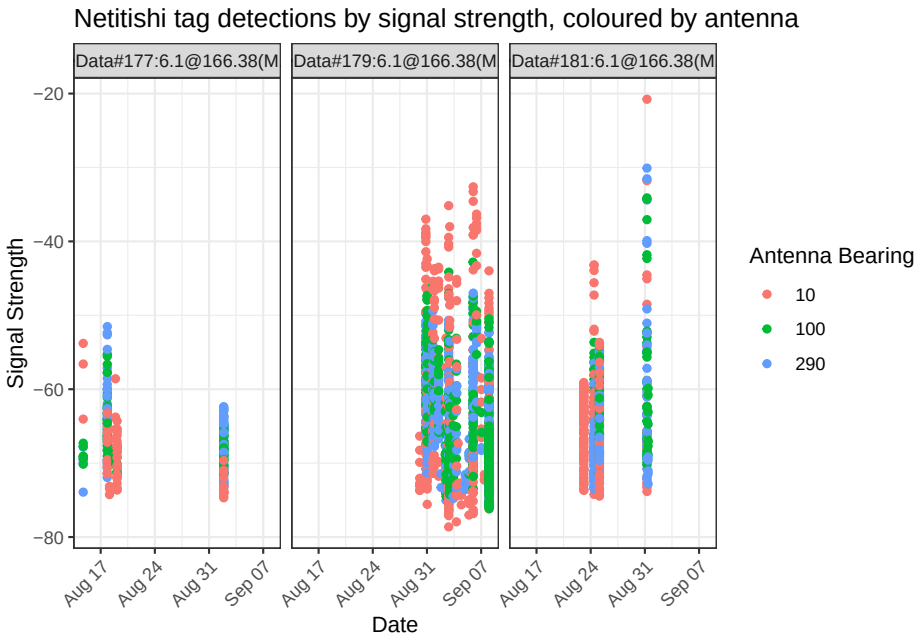
data a selected table from .motus data, e.g. `alltags`, or a data.frame of detection data including at a minimum variables for antBearing, `ts`, `lat`, `sig`, `fullID`, `recvDepName`
antBearing variable consisting antenna bearing variable

ts variable for a date/time object as numeric or POSIXct, defaults to ts
recvDeployLat variable consisting of receiver deployment latitude
sig variable consisting a signal strength variable
fullID variable consisting of a tag fullID
recvDepName variable consisting of receiver deployment name

C.8.3 Example

Plot select tags for site Piskwamish

```
plotSiteSig(filter(df.alltags, motusTagID %in% c(16037, 16039, 16035)),
            recvDeployName = "Netitishi")
```



C.9 plotTagSig

C.9.1 Description

Plot signal strength vs time for specified tag, faceted by site (ordered by latitude) and coloured by antenna

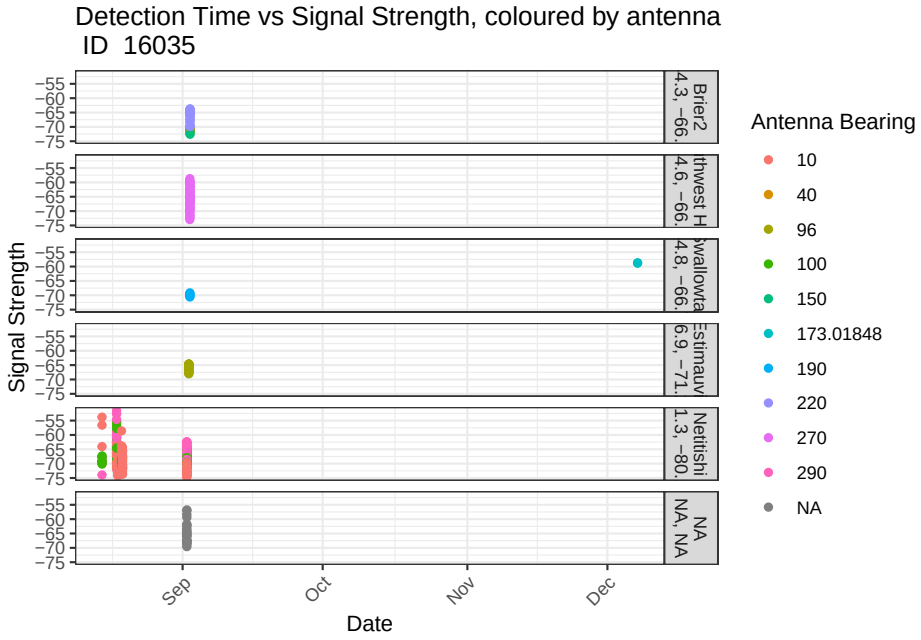
C.9.2 Arguments

data a selected table from .motus data, e.g. "alltags", or a data.frame of detection data including at a minimum variables for **motusTagID**, **sig**, **ts**, **antBearing**, **recvDeployLat**, **fullID**, **recvDepName**
motusTagID variable consisting of a motus tag ID
antBearing variable consisting antenna bearing variable
ts variable for a date/time object as numeric or POSIXct, defaults to **ts**
recvDeployLat variable consisting of receiver deployment latitude
sig variable consisting a signal strength variable
fullID variable consisting of a tag fullID
recvDepName variable consisting of receiver deployment name

C.9.3 Example

Plot signal strength of a specified tag using **tbl** file **tbl.alltags**

```
plotTagSig(tbl.alltags, motusTagID = 16035)
```



C.10 simSiteDet

C.10.1 Description

Creates a data.frame consisting of only detections of tags that are detected at two or more receivers at the same time.

C.10.2 Arguments

data a selected table from .motus data, e.g. **alltags**, or a data.frame of detection data including at a minimum variables for **ts**, **motusTagID**, **recvDepName**
ts variable for a date/time object as numeric or POSIXct, defaults to **ts**
motusTagID variable consisting of a motus tag ID
recvDepName variable consisting of receiver deployment name

C.10.3 Example

To get a data.frame called **simSites** of just simultaneous detections from a data.frame **df.alltags**

```
simSites <- simSiteDet(df.alltags)
head(simSites)
```

```
##  [1] motusTagID     ts             num.dup        hitID          runID          batchID
## [11] freq           freqsd         slop           burstSlop      done           ambigID
## [21] mfgID          tagType        codeSet        mfg            tagModel       tagLifespan
## [31] speciesID      markerNumber   markerType     tagDeployStart tagDeployEnd   tagDepLat
## [41] deviceID       recvDeployID   recvDeployLat  recvDeployLon  recvDeployAlt  recv
## [51] recvUtcOffset  antType        antBearing     antHeight      speciesEN      speciesFR
## [61] gpsLat         gpsLon         gpsAlt
## <0 rows> (or 0-length row.names)
```

C.11 siteSum

C.11.1 Description

Creates a summary of the first and last detection at a site, the length of time between first and last detection, the number of tags, and the total number of detections at a site. Plots total number of detections across all tags, and total number of tags detected at each site.

C.11.2 Arguments

data a selected table from .motus data, e.g. **alltags**, or a data.frame of detection data including at a minimum variables for **motusTagID**, **sig**, **recvDeployLat**, **recvDepName**, and **ts**
motusTagID variable consisting of a motus tag ID
sig variable consisting a signal strength variable
recvDeployLat variable consisting of receiver deployment latitude
recvDepName variable consisting of receiver deployment name
ts variable for a date/time object as numeric or POSIXct, defaults to ts
units units to display time difference, defaults to "hours", options include "secs", "mins", "hours", "days", "weeks"

C.11.3 Example

Create site summaries for select sites with time in minutes

```
site_summary <- siteSum(filter(df.alltags,
                               recvDeployName %in% c("Niapiskau", "Netitishi",
                                                     "Old Cur", "Washkaugou")),
                        units = "mins")
```


```
head(site_summary)
```

```
## # A tibble: 3 x 6
##   recvDeployName         first_ts            last_ts             tot_ts           r
```

```
##   <fct>                  <dttm>              <dttm>              <drtn>               <int>
## 1 Niapiskau_50.2, -63.7  2016-10-01 23:47:57 2016-10-27 00:03:21 36015.414767 mins        6
## 2 Washkaugou_51.2, -79.8 2016-10-09 23:52:45 2016-10-10 00:00:42     7.946833 mins        2
## 3 Netitishi_51.3, -80.1  2015-08-14 17:53:49 2015-09-08 01:10:13 34996.399048 mins        7
```

C.12 siteSumDaily

C.12.1 Description

Creates a summary of the first and last daily detection at a site, the length of time between first and last detection, the number of tags, and the total number of detections at a site for each day. Same as siteSum, but daily by site.

C.12.2 Arguments

data a selected table from .motus data, e.g. **alltags**, or a data.frame of detection data including at a minimum variables for **motusTagID**, **sig**, **recvDepName**, **ts**
motusTagID variable consisting of a motus tag ID
sig variable consisting a signal strength variable
recvDepName variable consisting of receiver deployment name
ts variable for a date/time object as numeric or POSIXct, defaults to ts
units units to display time difference, defaults to "hours", options include "secs", "mins", "hours", "days", "weeks"

C.12.3 Example

Create site summaries for all sites within detection data with time in minutes using **tbl** file **tbl.alltags**

```
daily_site_summary <- siteSumDaily(tbl.alltags, units = "mins")
head(daily_site_summary)
```

```
##                            recvDeployName       date            first_ts             last_ts
## 1      Assateague State Park\n38.2, -75.1 2015-09-13 2015-09-13 10:12:50 2015-09-13 10:14:40
## 2                    Baccaro\n43.5, -65.5 2017-05-19 2017-05-19 16:01:21 2017-05-19 16:02:40
## 3              BennettMeadow\n42.7, -72.5 2015-10-26 2015-10-26 11:12:04 2015-10-26 11:30:49 1
## 4 Binbrook_Conservation_Area\n43.1, -79.8 2017-05-18 2017-05-18 03:13:25 2017-05-18 03:13:46
## 5                       BISE\n41.2, -71.6 2015-10-26 2015-10-26 17:55:47 2015-10-26 19:16:55 8
## 6                  Blandford\n44.5, -64.1 2015-12-26 2015-12-26 14:58:27 2015-12-26 14:58:47
```

C.13 siteTrans

C.13.1 Description

Creates a data.frame of transitions between sites; detections are ordered by detection time, then "transitions" are identified as the period between the final detection at site x (possible "departure"), and the first detection (possible "arrival") at site y (ordered chronologically). Each row contains the last detection time and lat/lon of site x, first detection time and lat/lon of site y, distance between the site pair, time between detections, rate of movement between detections, and bearing between site pairs.

C.13.2 Arguments

data a selected table from .motus data, e.g. **alltags**, or a data.frame of detection data including at a minimum variables for **ts**, **motusTagID**, **tagDeployID**, **recvDeployLat**, **recvDeployLon**, **recvDepName**
ts variable for a date/time object as numeric or POSIXct, defaults to ts
motusTagID variable consisting of a motus tag ID
tagDeployID variable consisting of Motus tag deployment ID
recvDeployLat variable consisting of receiver deployment latitude
recvDeployLon variable consisting of receiver deployment longitude
recvDepName variable consisting of receiver deployment name

C.13.3 Example

View site transitions for only tag 16037 from data.frame **df.alltags**

```
transitions <- siteTrans(filter(df.alltags, motusTagID == 16037),
                         latCoord = "recvDeployLat", lonCoord = "recvDeployLon")
head(transitions)
```

```
## # A tibble: 6 x 16
## # Groups:   motusTagID, tagDeployID [1]
##   motusTagID tagDeployID data   consec  ts.x                lat.x lon.x recvDeployNa
##        <int>       <int> <list> <list>  <dttm>              <dbl> <dbl> <chr>
## 1      16037        1825 <tibb~ <tibbl~ 2015-08-17 17:02:39  NA    NA   NP mobile_NA
## 2      16037        1825 <tibb~ <tibbl~ 2015-08-28 16:40:18  51.5 -80.4 North Bluff_
## 3      16037        1825 <tibb~ <tibbl~ 2015-09-08 01:10:13  51.3 -80.1 Netitishi_5
## 4      16037        1825 <tibb~ <tibbl~ 2015-09-08 18:37:16  44.6 -65.8 Comeau (Mars
## 5      16037        1825 <tibb~ <tibbl~ 2015-09-13 19:46:27  39.0 -74.8 NWW_39, -74
## 6      16037        1825 <tibb~ <tibbl~ 2015-09-14 15:56:49  37.1 -76.0 BULL_37.1, -
```

C.14 tagSum

C.14.1 Description

Creates a summary for each tag of it's first and last detection time, first and last detection site, length of time between first and last detection, straight line distance between first and last detection site, rate of movement, and bearing

C.14.2 Arguments

data a selected table from .motus data, e.g. `alltags`, or a data.frame of detection data including at a minimum variables for `motusTagID`, `fullID`, `recvDeployLat`, `recvDeployLon`, `recvDepName`, `ts`
motusTagID variable consisting of a motus tag ID
fullID variable consisting of a tag fullID
recvDeployLat variable consisting of receiver deployment latitude
recvDeployLon variable consisting of receiver deployment longitude
recvDepName variable consisting of receiver deployment name
ts variable for a date/time object as numeric or POSIXct, defaults to ts

C.14.3 Example

Create tag summary for all tags within detection data using tbl file `tbl.alltags`

```
tag_summary <- tagSum(tbl.alltags)
head(tag_summary)
```

```
##                               fullID            first_ts             last_ts              firs
## 1      Niles#152:6.1@166.38(M.10811) 2015-08-03 06:37:11 2015-08-03 06:37:35 North Bluff_51.5,
## 2                  ?proj?-0#395:9.7 2015-07-23 10:10:54 2015-09-02 20:06:13     Machias_44.5,
## 3      Selva#172:6.1@166.38(M.17021) 2015-09-02 04:06:07 2015-09-03 00:27:16   Netitishi_51.3,
## 4 SampleData#152:6.1@166.38(M.16011) 2015-08-03 06:37:11 2015-09-18 09:37:39 North Bluff_51.5,
## 5 SampleData#395:9.7@166.38(M.16052) 2015-09-12 17:38:04 2015-10-20 20:43:43   Longridge_51.8,
## 6 SampleData#179:6.1@166.38(M.16037) 2015-08-17 17:01:38 2015-11-02 13:21:42        NP mobile_
##             tot_ts    dist       rate    bearing num_det
## 1      24.386 secs       0  0.0000000 -180.00000       4
## 2 3578118.674 secs 5087724  1.4218993  -38.36268      12
## 3   73268.958 secs 1211577 16.5360200  127.53144     279
## 4 3985228.228 secs 1452237  0.3644051  160.20437     127
## 5 3294338.784 secs 1472844  0.4470832  111.25011     147
## 6 6639604.044 secs      NA         NA         NA    1353
```

C.15 tagSumSite

C.15.1 Description

Creates a summary for each tag of it's first and last detection time at each site, length of time between first and last detection of each site, and total number of detections at each site.

C.15.2 Arguments

data a selected table from .motus data, e.g. "alltags", or a data.frame of detection data including at a minimum variables for **motusTagID**, **fullID**, **recvDepName**, **ts**
motusTagID variable consisting of a motus tag ID
fullID variable consisting of a tag fullID
recvDepName variable consisting of receiver deployment name
ts variable for a date/time object as numeric or POSIXct, defaults to **ts**

C.15.3 Example

Create tag summaries for only select tags with time in default hours with data.frame **df.alltags**

```
tag_site_summary <- tagSumSite(filter(df.alltags,
                                      motusTagID %in% c(16047, 16037, 16039)))
head(tag_site_summary)
```

```
##                                fullID                    recvDeployName            
## 1 SampleData#179:6.1@166.38(M.16037)                   BULL\n37.1, -76 2015-09-14 1
## 2 SampleData#179:6.1@166.38(M.16037) Comeau (Marshalltown)\n44.6, -65.8 2015-09-08 1
## 3 SampleData#179:6.1@166.38(M.16037)           Hillman_Marsh\n42, -82.5 2015-11-02 1
## 4 SampleData#179:6.1@166.38(M.16037)             Netitishi\n51.3, -80.1 2015-08-30 0
## 5 SampleData#179:6.1@166.38(M.16037)           North Bluff\n51.5, -80.5 2015-08-23 1
## 6 SampleData#179:6.1@166.38(M.16037)                 NP mobile\nNA, NA 2015-08-17 1
```

C.16 timeToSunriset

C.16.1 Description

Creates and adds variables for time to, and time from sunrise/sunset based on a variable of POSIXct dates/times data.frame must contain latitude, longitude, and a date/time variable

C.16.2 Arguments

data a selected table from .motus data, e.g. **alltags**, or a data.frame of detection data including at a minimum variables for date/time, latitude, and longitude
lat variable with latitude values, defaults to **recvDeployLat**
lon variable with longitude values, defaults to **recvDeployLon**
ts variable for a date/time object as numeric or POSIXct, defaults to **ts**
units units to display time difference, defaults to "hours", options include "secs", "mins", "hours", "days", "weeks"

C.16.3 Example

Get sunrise and sunset information with units in minutes using tbl file `tbl.alltags`

```
sunrise <- timeToSunriset(tbl.alltags, units = "mins")
head(sunrise)
```

```
##    hitID runID batchID                  ts tsCorrected sig sigsd noise freq freqsd  slop burst
## 1  45107  8886      53 2015-10-26 11:19:49  1445858390  52     0   -96    4      0 1e-04    0.
## 2  45108  8886      53 2015-10-26 11:20:28  1445858429  54     0   -96    4      0 1e-04   -0.
## 3  45109  8886      53 2015-10-26 11:21:17  1445858477  55     0   -96    4      0 1e-04    0.
## 4  45110  8886      53 2015-10-26 11:21:55  1445858516  52     0   -96    4      0 1e-04   -0.
## 5  45111  8886      53 2015-10-26 11:22:44  1445858564  49     0   -96    4      0 1e-04    0.
## 6 199885 23305      64 2015-10-26 11:12:04  1445857924  33     0   -96    4      0 1e-04    0.
##   codeSet   mfg tagModel tagLifespan nomFreq  tagBI pulseLen tagDeployID speciesID markerNumbe
## 1  Lotek4 Lotek NTQB-3-2          NA  166.38 9.6971      2.5        1839      4670    1352681C
## 2  Lotek4 Lotek NTQB-3-2          NA  166.38 9.6971      2.5        1839      4670    1352681C
## 3  Lotek4 Lotek NTQB-3-2          NA  166.38 9.6971      2.5        1839      4670    1352681C
## 4  Lotek4 Lotek NTQB-3-2          NA  166.38 9.6971      2.5        1839      4670    1352681C
## 5  Lotek4 Lotek NTQB-3-2          NA  166.38 9.6971      2.5        1839      4670    1352681C
## 6  Lotek4 Lotek NTQB-3-2          NA  166.38 9.6971      2.5        1839      4670    1352681C
##
## 1 {"ageID":"HY","bill":36.5,"blood":"Y","country":"Canada","culmen":36.5,"fatScore":3,"locatic
## 2 {"ageID":"HY","bill":36.5,"blood":"Y","country":"Canada","culmen":36.5,"fatScore":3,"locatic
## 3 {"ageID":"HY","bill":36.5,"blood":"Y","country":"Canada","culmen":36.5,"fatScore":3,"locatic
## 4 {"ageID":"HY","bill":36.5,"blood":"Y","country":"Canada","culmen":36.5,"fatScore":3,"locatic
## 5 {"ageID":"HY","bill":36.5,"blood":"Y","country":"Canada","culmen":36.5,"fatScore":3,"locatic
## 6 {"ageID":"HY","bill":36.5,"blood":"Y","country":"Canada","culmen":36.5,"fatScore":3,"locatic
##                                fullID deviceID recvDeployID recvDeployLat recvDeployLon recvDep
## 1 SampleData#378:9.7@166.38(M.16047)      486         2510      42.60699     -72.71657
## 2 SampleData#378:9.7@166.38(M.16047)      486         2510      42.60699     -72.71657
## 3 SampleData#378:9.7@166.38(M.16047)      486         2510      42.60699     -72.71657
## 4 SampleData#378:9.7@166.38(M.16047)      486         2510      42.60699     -72.71657
## 5 SampleData#378:9.7@166.38(M.16047)      486         2510      42.60699     -72.71657
## 6 SampleData#378:9.7@166.38(M.16047)      515         2512      42.68067     -72.47392
```

```
##   recvUtcOffset antType antBearing antHeight speciesEN          speciesFR       spec
## 1            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris c
## 2            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris c
## 3            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris c
## 4            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris c
## 5            NA  yagi-9        127        NA  Red Knot Bécasseau maubèche Calidris c
## 6            NA  yagi-9        243        NA  Red Knot Bécasseau maubèche Calidris c
##               sunrise              sunset ts_to_set ts_since_set  ts_to_rise ts_sinc
## 1 2015-10-26 11:16:49 2015-10-26 21:52:11  632.3533     806.2056 1438.215974      3
## 2 2015-10-26 11:16:49 2015-10-26 21:52:11  631.7104     806.8485 1437.573097      3
## 3 2015-10-26 11:16:49 2015-10-26 21:52:11  630.9109     807.6480 1436.773577      4
## 4 2015-10-26 11:16:49 2015-10-26 21:52:11  630.2513     808.3076 1436.113997      5
## 5 2015-10-26 11:16:49 2015-10-26 21:52:11  629.4518     809.1071 1435.314477      5
## 6 2015-10-26 11:15:58 2015-10-26 21:51:06  639.0310     799.5242    3.896393   1437
```

Appendix D

Appendix - `motus` - Data filtering functions

The `motus` R package offers functions that can be used to assign probabilities to tag detections, and to filter detections based on those probabilities. For example, as you work through your data to clean false positive and ambiguous detections (see Chapter 5), you may determine that some detections do not belong to your tag(s). Instead of simply using an R script to filter out those detections, you can use these filter functions to create and save a custom filter in your .motus file, which assigns a probability value between 0 and 1 to the runIDs supplied in the filter.

The data filtering functions in the R package work at the level of a run. A run is a group of consecutive detections of a tag detected on a receiver. In general, a detection with a run length of 2 has a high probability of being a false positive detection. The probabilities associated with each runID can be generated in a number of possible ways, including at the simplest level, generating a list of 0's and 1's for records that you would like to exclude or include. Alternatively, you might develop a model that assigns a probability to each runID in your data.

D.1 `listRunsFilters`

D.1.1 Description

Returns a dataframe containing the filterIDs, logins, names, projectIDs and descriptions for a given tag or receiver projectID available in the local database.

D.2 Arguments

src is the SQLite object that you get from loading a .motus file into R, e.g., 'sql.motus' file in Chapter 3.

D.3 Example

```
filt.df <- listRunsFilters(src = sql.motus)
```

D.4 createRunsFilter

D.4.1 Description

This function can be used mostly by users to modify properties of existing filters (e.g., filter description or projectID), but it is also being called internally by 'writeRunsFilter' (section D.6) to generate a new filterID. To save the actual filter records, you must use `writeRunsFilter` (section D.6). The function returns the `filterID` (integer) in the local database that matches the new or existing filter with the provided filterName. If a filter with the same name already exists, the function generates a warning and returns the ID of the existing filter.

D.4.2 Arguments

`src` is the SQLite object that you get from loading a .motus file into R, e.g., 'sql.motus' file in Chapter 3.
`filterName` the name you would like to assign to the filter. The function only creates a new filter if the name does not already exist locally.
`motusProjID` the numeric ID associated with a project, e.g., 176 for the sample data used throughout this book. The function defaults to motusProjID = 'NA' when project ID is not supplied, which is the recommended value for now. The project ID assigned to a filter will mostly be useful for future synchronization of filters with the Motus server. The detection records contained in the filter do not have to be assigned to the projectID assigned to the filter. **`descr`** default 'NA'. Optional description of the filter. **`update`** boolean (default = FALSE). If the filter already exists, determines if the properties (e.g. descr are preserved or updated)

D.4.3 Example

Create a new filter called "myfilter" for the sql.motus database which is not attached to a specific project:

```
createRunsFilter(sql.motus, "myfilter")

# OR add assignment to project

createRunsFilter(sql.motus, "myfilter", motusProjID = 176)

# OR add project and description, possibly updating any previous version called myfilter.

createRunsFilter(sql.motus, "myfilter", motusProjID = 176,
                 descr = "assign probability of 0 to false positives", update = TRUE)
```

D.5 getRunsFilters

D.5.1 Description

Returns a sqlite table reference to the runsFilters records saved in the database (`runID`, `motusTagID`, and `probability`) associated with a specific name (and optionally project) from the local database. For examples on how you can use the returned table to merge with your detection data, refer to section 5.9.2 in chapter 5.

D.5.2 Arguments

`src` is the SQLite object that you get from loading a .motus file into R, e.g., 'sql.motus' file in Chapter 3.
`filterName` the name you used when you created or saved your filter. Function returns a warning if the `filterName` doesn't exist.
`motusProjID` the numeric ID associated with a project, e.g., 176 for the sample data used throughout this book. The function defaults to `motusProjID = 'NA'` when project ID is not supplied.

D.5.3 Example

```
tbl.filt <- getRunsFilters(src = sql.motus, filterName = "myfilter")
tbl.filt2 <- getRunsFilters(sql.motus, "myfilter2")

# filter records from df that are in tbl.filt
```

```
df <- left_join(df, tbl.filt, by = c("runID", "motusTagID")) %>%
  mutate(probability = if_else(is.na(probability), 1, probability)) %>%
  filter(probability > 0)

# you can apply a second filter, tbl.filt2, to the result of the previous filter
df <- left_join(df, tbl.filt2, by = c("runID", "motusTagID")) %>%
  mutate(probability = if_else(is.na(probability), 1, probability)) %>%
  filter(probability > 0)
```

D.6 writeRunsFilter

D.6.1 Description

Writes to the local database (SQLite file) the content of a dataframe containing `runID`, `motusTagID`, and assigned `probability`. If the `filterName` provided does not exist, the function will call `createRunsFilter` (section D.4) to create one in your database. The default behaviour of the function is that any new records from the dataframe are appended to the existing or new filter called filterName, those that already are present (same `runID` and `motusTagID`) are replaced (`overwrite=TRUE`), but those that are not included in the dataframe are retained in the existing filter table (`delete=FALSE`). To entirely replace the existing filter values with those of the new dataframe, use `delete=TRUE`. The function returns a sqlite table reference to the filter, similarly to `getRunsFilter` (section D.5).

D.6.2 Arguments

`src` is the SQLite object that you get from loading a .motus file into R, e.g., 'sql.motus' file in Chapter 3.
`filterName` the name of the filter you would like to assign the database to.
`motusProjID` the numeric ID associated with a project, e.g., 176 for the sample data used throughout this book. Default = 'NA' when project ID is not supplied.
`df` dataframe which contains the `runID` (integer), `motusTagID` (integer), and `probability` (float) of detections you would like to assign a filter to. `motusTagID` should be the actual tag ID, and not the negative `ambigID` associated with ambiguous detections. **`overwrite`** `Default = "TRUE"`. When `TRUE`, ensures that existing records (same `runID` and `motusTagID`) matching the same `filterName` and `runID` get replaced in the local database. **`delete`** `Default = "FALSE"`. When `TRUE`, removes all existing filter records associated with the `filterName` and re-inserts the ones contained in df. This option should be used if df contains the entire set of filters you want to save.

D.6.3 Examples

```
# write a dataframe containing filter records (runID, motusTagID and
# probability) to "myfilter"
writeRunsFilter(src = sql.motus, filterName = "myfilter", df = filter.df)

# write a dataframe containing filter records (runID, motusTagID and
# probability) to "myfilter", overwriting a previous version entirely
writeRunsFilter(src = sql.motus, fileName = "myfilter", df = filter.df, delete = TRUE)

# write a dataframe containing filter records (runID, motusTagID and
# probability) to "myfilter", but only append new records, leaving previously
# created ones intact
writeRunsFilter(src = sql.motus, "myfilter", df = filter.df, overwrite = FALSE)
```

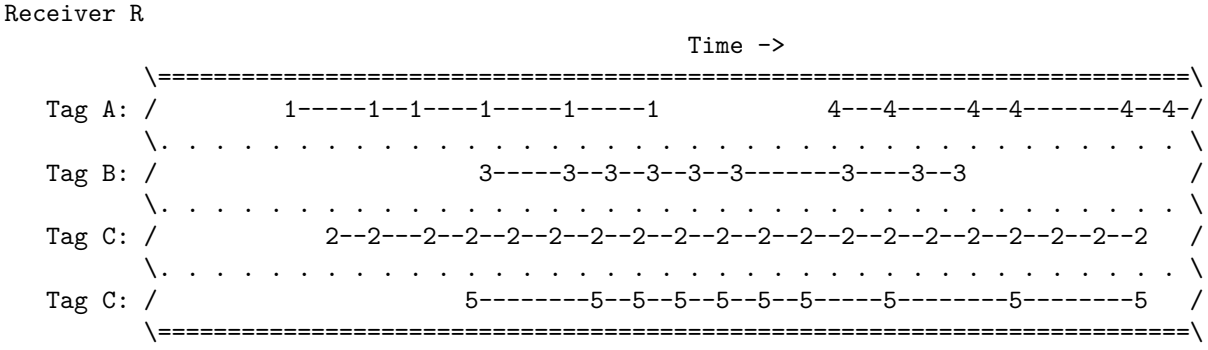
Appendix E

Bird's Eye View of Motus Data

This document has been adapted from a version written by John Brzustowsky in 2017. It provides an overview of the fundamentals of how Motus data processing works. The document has been adapted to incorporate more recent developments, particularly the addition of new digital tags and base stations manufactured by Cellular Tracking Technologies (CTT), who took over the development from Cornell University.

E.1 What data look like

Here's a segment of data from a receiver (with a single antenna):



- time increases from left (earlier) to right (later)
- horizontal *lanes*, separated by . . . correspond to individual tags, labelled at left

- **hits** (detections) of a tag are plotted as single digits within its lane
- in the diagram above, runs are labelled by digits. So all hits of tag A above are either in run `1` or in run `4`.
- the same tag can be detected simultaneously on multiple antennae, and therefore be part of runs that overlap in time. For example, tag C is detected on antenna 1 and 2.
- In principle, the same tag should never be part of overlapping runs on the SAME antenna, but in practice this can happen in noisy environments with Lotek tags, and it should be assumed that these runs are likely from other sources than actual tags (false positives).
- Overlapping run would not happen with CTT tags because of the way runs are built, but the rate of hits
  would still be expected to be around 1 per second or lower.
- for Lotek coded ID tags, individual hits are not sufficient to determine which tag is being detected, because multiple tags transmit the same ID. However, the period (spacing between consecutive transmissions) is precise, and differs among tags sharing the same ID.
- the fundamental unit of detection for Lotek coded ID tags is the **run**, or sequence of hits of a given ID, with spacing **compatible** with the value for that tag. i.e. two hits might differ by the tag's period, or by twice its period, or three times its period, ..., up to a limit required by drift between clocks on the transmit and receive sides.
  - for Lotek ID tags, a run can have gaps (missing hits) up to a certain size. Beyond that size, measurement error and clock drift are large enough that we can't be sure that the next detection of the same ID code is after a compatible time interval. So for tag A above, we're not sure the gap between the last detection in run 1 and the first detection in run 4 is really compatible with tag A, so run 1 is ended and a new run is begun. We could link runs 1 and 4 post-hoc, once we saw that run 4 was being built with gaps compatible with tag A, but at present, the run-building algorithm doesn't backtrack.)
  - for CTT tags, there are 2^32 unique codes, which ensures that each of them is unique. Contrary to Lotek
    tags, runs are not limited to consecutive detections matching a precise period, in part because the period may vary depending on the energy available from the photovoltaic cell. Run are still assembled based on consecutive detections on a single antenna (or node, if applicable), as long as they are not spaced by more than an arbitrary period (600 seconds).

E.2 False positives

False positives (apparent detections of tags actually caused by other sources) exist in all technologies and need to be taken into account. These can happen for quite a number of reasons, and will sometimes affect Lotek and CTT tags in different measure. False positives are difficult to identify, but there are approaches to identify conditions in which the are more likely to occur, and potential ways to mitigate them.

- Noisy environments: radio noise from interference can create bursts that look like tags. As the amoung of radio pulses from other sources increases in the environment, so it the expected number of false positives.
- Bit errors: actual tag signals may be incorrectly transmitted. Those should rarely produce valid ID's, but they will be more likely with CTT tags, mostly because where is a very high number of possible combinations (4 billions). They should still rarely produce an ID of a tag actually manufactured. In CTT tags, bit errors has been found to lead more often to certain patterns (e.g. the last trailing digits being all 7 or F's: xxxFFFF due to only a partial signal being received), and those patterns have since been excluded from the list of valid tags.
- Aliasing: aliasing happens when the combination of multiple tags create the appearance of a tag that is not present but matches another known tag. This can happen in at least 2 ways: 1) 2 tags with distinct ID's, but with the same period (both Lotek and CTT). 2) 2 tags with the same ID, but with distinct periods (only for Lotek). In the first case, if the burst of both tags overlap, they may interfere with each other and create the appearance of a new tag for a while. Assuming that the 2 tags do not have exactly the same period, their bursts should eventually drift apart and the precise alias probably will not persist over a very long run. In the second case, if you have multiple tags with the same ID but distinct periods, this may potentially also result in new periods, but those would likely rarely exceed 2 consecutive hits (run length = 2). Both types of aliasing is mostly problematic in environments where you have many tags present simultaneously that increases the incidence of overlapping tags (e.g. colonies).

For both Lotek and CTT tags, their manufacturers have integrated various methods to reduce the incidence of false detections into their proprietary technologies. Data collected by Sensor Gnomes are processed in Motus by the "Tag finder", which looks at properties of the signal to exclude likely false positives (e.g. higher deviation in the frequency of pulses within a burst). The parameters can potentially be adjusted, but an aggressive approach aimed at reducing false positives will also result in an increase in false negatives.

- Run length: With both types of tag technologies, the likelihood of obtaining false positives should decrease as the run length increases. For Lotek tags,

we generally recommend ignoring runs comprised only of 2 or 3 hits. Some (many?) probably relate to real tag detections, but the vast majority probably do not. For CTT tags, we do not yet have a suggested minimum threshold. In most cases, given that tag period is short, one would expect that true single-hit runs to be quite rare, so those should likely be excluded to be safer. The likelihood of obtaining the same false ID in consecutive detections is probably very low, except for some specific tag ID's that are more prone to error. Runs of 2 detections or more are probably safe in most instances.

- Measures of noise. A higher amount of radio pulses is likely a good predictor of false positives, though other factors are also at play, and not all radio noise is problematic in the same way. We recommend that you use the number of runs comprised of only 2 hits, which is provided by the activity table (see details here). In noisy environments, shorter runs are much more likely to be false positives and should be excluded.
- Missed detections: False positives should more often lead to gaps in detections during a run. A run that contains several gaps in detection would therefore be deemed less reliable. A simple metric for this would be to divide the number of hits in a run (the run lenght) by the duration of the run (tsEnd - tsStart). Longer runs with few or no gaps in detection would be optimal.
- Overlapping runs. Runs of the same tag on the same antenna should also be an indication of false positives, but this should be highly correlated with the number and/or ratio of short runs described above.
- Spatio-temporal models. State-space models and other approach can be used to assess whether movement make sense from a biological point of view, and can help assign probabilities that individual detections are valid.
- If you have any suggestions about techniques you are using to separate true detections from false positives, we encourage you to share them with us!

E.3 False negatives

False negatives are usually even more difficult to detect.

- Faulty equipement or installation is always possible of course. Please refer to the installation guidelines to make sure you follow all the recommendations.
- DO NOT FORGET to register your tags! (details here)
- DO NOT FORGET to activate your tags before deploying them!

- Make sure that you report your tag and receiver deployment details BEFORE uploading your data. Motus tag finder only seek tags that are known to be deployed.

E.4 Complication: data are processed in batches

The picture above is complicated by several facts:

- receivers are often deployed to isolated areas so that we can only obtain raw data from them occasionally
- receivers are not aware of the full set of currently-active tags
- sensorgnome receivers do not "know" the full Lotek code set; they record pulses thought to be from Lotek coded ID tags, but are only able to assemble them into tag hits for a small set of tags for which they have an on-board database, built from the user's own recordings of their tags. This limitation is due to restrictions in the agreement between Lotek and Acadia University for our use of their codeset.
- Lotek receivers report each detected tagID independently, and do not assemble them into runs. This means a raw Lotek .DTA file does not distinguish between tag 123:4.5 and tag 123:9.1 (i.e. between tags with ID `123` and burst intervals 4.5 seconds and 9.1 seconds).

It follows that:

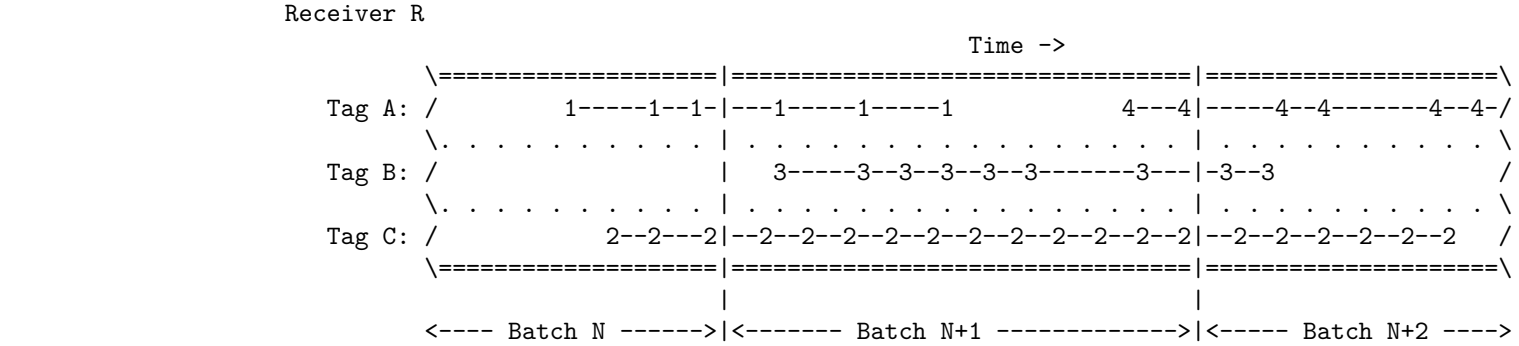
- raw receiver data must be processed on a server with full knowledge of:
  - which tags are deployed and likely active
  - the Lotek codeset(s)
- raw data should be processed in incremental batches
- processed data should be distributed to users in incremental batches, especially if they wish to obtain results "as they arrive", rather than in one lump after all their tags have expired.

E.5 Batches

A **batch** is the result of processing one collection of raw data files from a receiver. Batches are not inherent in the data, but instead reflect how data were processed. Batches arise in these ways:

- a user visits an isolated receiver, downloads raw data files from it, then uploads the files to motus.org once they are back from the field
- a receiver connected via WiFi, ethernet, or cell modem is polled for new data files; this typically happens every hour, with random jitter to smooth the processing load on the motus server
- an archive of data files from a receiver is re-processed on the motus server, because important metadata have changed (e.g. new or changed tag deployment records), or because a significant change has been made to processing algorithms.

Batches are artificial divisions in the data stream, so runs of hits will often cross batch boundaries. Adding this complication to the picture above gives this:



E.5.1 Receiver Reboots

A receiver **reboots** when it is powered down and then (possibly much later) powered back up. Reboots often correspond to a receiver:

- being redeployed
- having its software updated
- or having a change made to its attached radios,

so motus treats receiver reboots in a special way:

- a reboot always begins a new batch; i.e. batches never extend across reboots. This simplifies determination of data ownership. For example, all data in a boot session (time period between consecutive reboots) are deemed to belong to the same motus project. This reflects the fact that a receiver is (almost?) always turned off between the time it is deployed by one project, and the time it is redeployed by another project.
- any active tag runs are ended when a receiver reboots. Even if the same tag is present and broadcasting, and even if the reboot takes only a few minutes, hits of a tag before and after the reboot will belong to separate runs. This is partly for convenience in determining data ownership, as

mentioned above. It is also necessary because sometimes receiver clocks are not properly set by the GPS after a reboot, and so the timestamps for that boot session will revert to a machine default, e.g. 1 Jan 2000. Although runs from these boot sessions could in principle be re-assembled post hoc if the system clock can be pinned from information in field notes, this is not done automatially at present.

- parameters to the tag-finding algorithm are set on a per-batch basis. At some field sites, we want to allow more lenient filtering because there is very little radio noise. At other sites, filtering should be more strict, because there is considerable noise and high false-positive rates for tags. motus allows projects to set parameter overrides for individual receivers, and these overrides are applied by boot session, because redeployments (always?) cause a reboot.
- when reprocessing data (see below) from an archive of data files, each boot session is processed as a batch.

E.5.2 Incremental Distribution of Data

The Motus R package allows users to build a local copy of the database of all their tags' (or receivers') hits incrementally. A user can regularly call the `tagme()` function to obtain any new hits of their tags. Because data are processed in batches, `tagme()` either does nothing, or downloads one or more new batches of data into the user's local DB.

Each new batch corresponds to a set of files processed from a single receiver. A batch record includes these items: - receiver device ID - how many of hits of their tags occurred in the batch - first and last timestamp of the raw data processed in this batch

Each new batch downloaded will include hits of one or more of the users's tags (or someone's tags, if the batch is for a "receiver" database).

A new batch might also include some GPS fixes, so that the user knows where the receiver was when the tags were detected.

A new batch will include information about runs. This information comes in three versions:

- information about a new run; i.e. one that begins in this batch
- information about a continuing run; i.e. a run that began in a previous batch, has some hits in this batch, and is not known to have ended
- information about an ending run; i.e. a run that began in a previous batch, might have some hits in this batch, but which is also known to end in this batch (because a sufficiently long time has elapsed since the last detection of its tag)

Although the unique `runID` identifier for a run doesn't change when the user calls `tagme()`, the number of hits in that run and its status (`done` or not), might change.

E.6 Reprocessing Data

motus will occasionally need to reprocess raw files from receivers. There are several reasons:

- new or modified tag deployment records. The tag detection code relies on knowing the active life of each tag it looks for, to control rates of false positive and false negative hits. If the deployment record for a tag only reaches the server after it has already processed raw files overlapping the tag's deployment, then those files will need to be reprocessed in order to (potentially) find the tag therein. Similary, if a tag was mistakenly sought during a period when it was not deployed, it will have "used up" signals that could instead have come from other tags, thereby causing both its own false positives, and false negatives on other tags. (This is only true for Lotek ID tags; CTT should be unaffected, provided deployed tags are well dispersed in the ID codespace.)
- bug fixes or improvements in the tag finding algorithm
- corrections of mis-filed data from receivers. Sometimes, duplication among receiver serial numbers (a rare event) is only noticed *after* data from them has already been processed. Those data will likely have to be reprocessed so that hits are assigned to the correct station. Interleaved data from two receivers having the same serial number will typically prevent hits from at least one of them, as the tag finder ignores data where the clock seems to have jumped backwards significantly.

E.7 The (eventual) Reprocessing Contract

Reprocessing can be very disruptive from the user's point of view ("What happened to my hits?"), so motus reprocessing will be:

1. optional: users should be able to obtain new data without having to accept reprocessed versions of data they already have.
2. reversible: users should be able to "go back" to a previous version of any reprocessed data they have accepted.
3. transparent: users will receive a record of what was reprocessed, why, when, what was done differently, and what changed

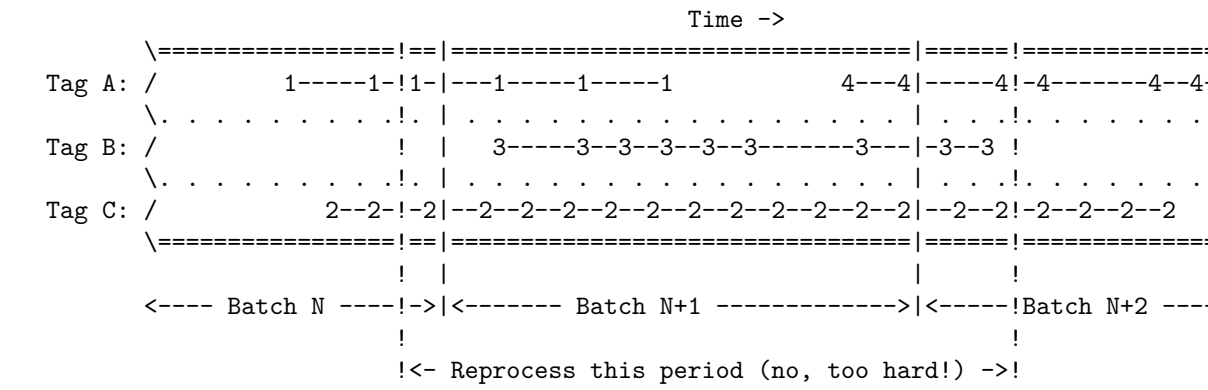
4. all-or-nothing: for each receiver boot session for which users have data, these data must come entirely from either the original processing, or a subsequent single reprocessing event. The user *must not* end up with an undefined mix of data from original and reprocessed sources.

5. in-band: the user's copy of data will be updated to incorporate *reprocessed* data as part of the normal process of updating to obtain *new* data, unless they choose otherwise. We expect that most users will want to accept reprocessed data most of the time.

Initially, motus data processing might not adhere to this contract, but it is an eventual goal.

E.8 Reprocessing simplified: only by boot session

A general reprocessing scenario would look like this:



if raw data records from an arbitrary stretch of time could be reprocessed. However, this is complicated because runs like `1` `2`, and `4` above might lose or gain hits within the reprocessing period, but not outside of it. This might even break an existing run into distinct new runs.

This situation is challenging (NB: not impossible; might be a TODO) to formalize and represent in the database if we want to maintain a full history of processing. For example, if reprocessing deletes some hits from run `2`, how do we represent both the old and the new versions of that run?

The complications arise due to runs crossing the reprocessing period boundaries, so for simplicity we should *choose a reprocessing period that* ***no*** *runs cross*. Currently, that means a boot session, as discussed above.

E.9 Distributing reprocessed data

The previous section shows why we only reprocess data by boot session. Given that, how do we get reprocessed data to users while fulfilling the reprocessing contract?

Note that a reprocessed boot session will fully replace one or more existing batches and one or more runs, because batches and runs both nest within boot sessions.

Replacement of data by reprocessed versions should happen in-band (`5` above), so one approach is this:

- the `batches_for_XXX` API entries should mark new batches which result from reprocessing, so that the client can deal with them appropriately. This can be done by adding a field called `reprocessID` with these semantics:
  - `reprocessID == 0`: data in this batch are from new raw files; the normal situation
  - `reprocessID == X > 0`: data in this batch are from reprocessing existing raw files.
  - `X` is the ID of the reprocessing event, and a new API entry `reprocessing_info (X)` can be called to obtain details about it.
  - if the user chooses to accept the reprocessed version, then existing batches, runs, hits and GPS fixes from the same receiver and boot session are retired before adding the new batches.
  - if the user chooses to reject the reprocessed version, then `X` is added to a client-side blacklist, and the user will not receive any data from batches whose reprocessID is on the blacklist.
  - later, if a user decides to accept a reprocessing event they had earlier declined, then the IDs of new batches for that event can be fetched from another new API `reprocessing_batches (X)`, and the original batches will be deleted
- to let users more efficiently fetch the "best" version of their dataset (i.e. accepting all reprocessing events), we should also mark batches which are subsequently replaced by a reprocessing event. For this, we add the field `replacedIn` with these semantics:
  - `replacedIn == 0`: data in this batch have not been replaced by any reprocessing event
  - `replacedIn == X > 0`: data in this batch have been replaced by reprocessing event `X`. Then the client can ignore any batches for which `replacedIn > 0`. We could also add a new boolean parameter `unreplacedOnly` to the `batches_for_XXX` API entries. It defaults to `false`, but if `true`, then only batches which have not been replaced by subsequent reprocessing events are returned.
- users can choose a policy for how reprocessed data are handled by setting the value of `Motus$acceptReprocessedData` in their workspace before calling `tagme()`:

- `Motus$acceptReprocessedData <- TRUE`; always accept batches of data from reprocessing events
- `Motus$acceptReprocessedData <- FALSE`; never accept batches of data from reprocessing events
- `Motus$acceptReprocessedData <- NA` (default); ask about each reprocessing event