An Introduction to R for Analyzing Acoustic Telemetry Data

Thomas Binder, Todd Hayden, and Christopher Holbrook

Version 2.0: February 19, 2018

GLATOS
Great Lakes Acoustic Telemetry Observation System

Great Lakes Fishery Commission
Great Lakes RESTORATION
MICHIGAN STATE UNIVERSITY
USGS

An Introduction to R for Analyzing Acoustic Telemetry Data

Thomas Binder, Todd Hayden, and Chris Holbrook

Table of Contents

Acknowledgements

We are grateful to Nancy Nate, Henry Thompson, Michael Lowe, and Christopher Wright for their assistance reviewing the materials in this document, and to Nancy Nate and Michael Lowe for additional time spent testing the functions in the glatos R package. We also thank R Workshop attendees from the 2017 Annual GLATOS Meeting for their constructive comments during and after the workshop, which guided substantial changes to both this document, and the glatos R package. Lastly, we thank Charles Krueger for supporting the creation of this document, and the Great Lakes Acoustic Telemetry Observation System (GLATOS) for funding to have this document printed for workshop attendees.

Purpose and Scope

The purpose of this guide and the associated workshop is to introduce GLATOS members to powerful tools for summarizing, visualizing, and analyzing acoustic telemetry data. THIS IS NOT A COMPREHENSIVE COURSE IN R, and we acknowledge that the tools and tips presented in this workshop may not be the only (or even the best) ways to apply R to the analysis of acoustic telemetry data. This workshop is geared towards members of GLATOS, and as such, many of the examples and exercises that we present in this workshop are designed to work from the detection data exported from the GLATOS detection database. Sample data are provided for use in the workshop, but example scripts should work on any GLATOS detection export.

Why Use R?

R is a scripting language designed for data manipulation, analysis, and visualization. There are many scripting languages, but R is popular among researchers because it is open source, and it is relatively intuitive for non-programmers. Advantages of using R over other common software (e.g., Microsoft Excel and SigmaPlot) to analyze data include:

1. R is free!
2. R is capable of handling large datasets. Microsoft Excel, for example, has a limit of just over one million rows. Many telemetry datasets will exceed this limit.
3. R uses data objects called data frames that can be manipulated similarly to spreadsheets in Microsoft Excel. This property helps to make the language more intuitive to non-programmers.
4. Because R is open-source, there are existing tools (i.e., functions) to serve most analysis needs. It typically is also very easy to find solutions to common (and even uncommon) problems with a quick Google search or posting questions online to the community of R users (e.g., Stack Overflow).

5. As a scripting language, R depends on written code to perform actions on data. This has at least four advantages:

- Transparency: The code serves as a written record of exactly what manipulations were done to the data.
- Reproducibility: The code allows researchers to easily repeat analyses.
- Adaptability: Implementing changes to the analysis and re-computing results is quick and efficient.
- Extensibility: Provides flexibility for improving and extending a process.

R Basics

Getting Started

R can be downloaded online from the R Project home page (https://www.r-project.org) by clicking the 'download R' link under 'Getting Started'.

R has a built-in text editor, but it is somewhat limited in functionality, so you will want to use a third-party script editor. RStudio is a popular script editor for R that can be downloaded at https://www.rstudio.com. Other script editors like Notepad++ (Windows), TinnR, and TextWrangler (Mac) can be used, but they may require some extra work to get them linked with R and are not as closely integrated with R.

If you are a new to R, we recommend RStudio as your script editor. One main advantage of RStudio is that the major components of an R session (text editor, R console, help files, plot output, data objects) are organized and accessible on one screen (see RStudio screen capture below). Another advantage of RStudio is that it offers drop-down auto-fill and auto-format capabilities, which can help to save on typing and, consequently, allows you to write code more quickly.

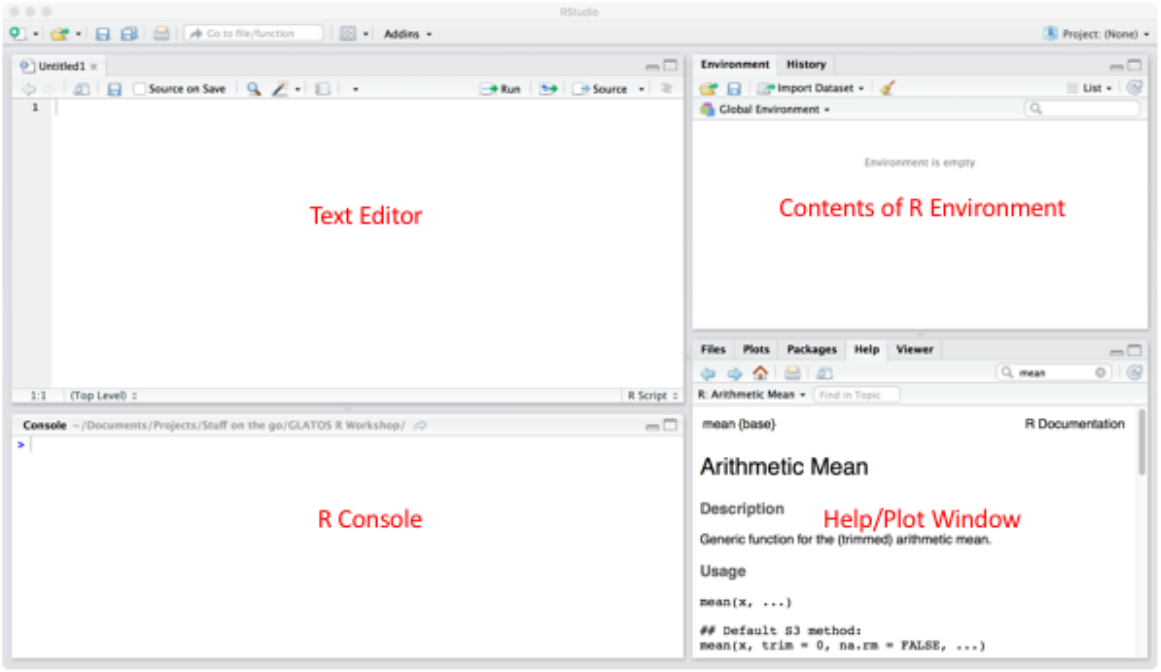


Figure 1. Screen capture of RStudio window with labels indicating the four major components.

The text editor is where you write code that will be passed to R. To create a new script, select 'New File –> R Script' from the 'File' menu. This 'script' serves as written record of what you have done. You can pass your code (i.e., script) to R in several different ways in RStudio, but the simplest is to highlight the specific code, or lines of code, you wish to pass with your mouse and then press ctrl+enter on your keyboard. You can also pass code to R using the 'Run' button at the top right of the text editor pane, or by selecting one of the run options from the main 'Code' menu.

Code passed to R from the script editor will appear in the R console, along with any non-graphical outputs. You can run code directly from the R console manually by typing commands next to the '>' symbol. If you

see a '+' symbol instead of '>', then R is waiting for more code to be entered before it can finish running a command. This usually means that you have passed an incomplete command to R. If this is the case, simply submit the remaining code to complete the command. Otherwise, press the Esc key on your keyboard to abort the command and return to the command prompt ('>').

The top-right pane of RStudio lists data objects (i.e., named bundles of structure data) in the R global environment. The global environment is the memory space that R reads and writes to. When R encounters a call for an object within a function (packet of commands sent to R that does something), it will look for an object first within the environment that is created for that function, and then within the global environment. Objects created within a function are stored in that function's environment, unless they are 'returned' to the global environment. Objects created outside functions are stored in the global environment. This concept will be important when you start to write your own functions.

Lastly, in the bottom right pane of RStudio we have quick access to help files for packages and functions that have been loaded into the global environment. Packages are groups of functions that are loaded into R and typically used to perform a certain task. Packages can also be installed using the packages tab or running `install.packages()` function in R. After installing the package, the `library()` function is used to load the package into the R global environment. In addition to a packages tab, there is a 'Plots' tab for viewing plots that were created with our code. The global environment can be cleared using the 'Clear Workspace' option in the 'Sessions' menu of R Studio. It is also cleared when you close R Studio. If the workspace is cleared, packages must be re-loaded into the global environment before their functions can be accessed.

Getting Help for R

Functions available for use in R have associated help file that contain the information necessary to use the functions, including: a list of parameters (or input arguments), a detailed explanation of the function, and examples. To access a help file, type a question mark followed by the function name (e.g., `?mean`) into the R console and press enter. If you are using R Studio, you can access the help files by searching the function name in the 'Help' tab on the right-hand side of the screen.

When reading R scripts, you will often come across text preceded by a `#`. These lines of texts are called comments, and the `#` tells R not to evaluate the rest of the current line when the code is executed. Comments are often used to help readers understand exactly what specific lines of code are doing.

There are a number of online resources available to R users, ranging from simple Google searches to dedicated programming blogs. Stack Overflow (www.stackoverflow.com) is a useful forum for posing questions to the R community. You can sign up for an account and then post questions or provide answers to questions that have been posed by others. In our experience, questions posted on Stack Overflow are usually answered within a couple of days, but are often answered within minutes of being posted. Quick-R (www.statmethods.net) is another online resource that provides the user with easily understandable examples of basic and advanced concepts in R.

Excellent Supplementary R books: 'The Art of R Programming: A Tour of Statistical Software Design' by Norman Matloff 'R for Everyone: Advanced Analytics and Graphics' by Jared Lander 'Advanced R' by Hadley Wickham 'Discovering Statistics Using R' by Andy Field

A Note on the Format of This Guide

This guide is formatted to help the reader differentiate explanatory text from source code and R output. Sections of source code and their associated outputs are presented in shaded boxes. Functions and objects

referred to in the main body of the document appear in ***bold italics***. The source code and output sections of this document will not look exactly like they do in the R console. The biggest difference is with the R output. In the R console, the output will usually appear next to a number (or column of numbers) in square brackets ('[]'). These numbers are index numbers for the object being returned. In this guide, R output is preceded by double hash tags (**##**), not to be mistaken for a comment, which is preceded by a single hash tag (**#**). Another difference is that source code is preceded by the command prompt **>** in the R console.

Below is an example of how source code and associated output appears in this manual (Figure 2), followed by how the same code and output will look in your R console (Figure 3).

```
# This is a comment. A comment is not evaluated by R. In the next two lines of
code, we will assign the numeric value 8 to an object named'a'. We will then
print 'a' to return our stored value.
a <- 8
a

## [1] 8
```

Figure 2. Source code and output in this guide.

```
> # This is a comment. A comment is not evaluated by R. In the next two lines of code, we will assign the
numeric value 8 to an object named'a'. We will then print 'a' to return our stored value.
> a <- 8
> a
[1] 8
```

Figure 3. Source code and output in the R console.

Functions, Data Types, and Data Objects

Functions

A function is a self-contained piece of code that performs a specific task. A function can be thought of like an assembly line. Data and instructions (combined, referred to as 'arguments') are supplied to the function. The function then processes the data using the provided instructions and returns a final product. A simple example of a function in R is the ***seq()*** function, which creates a series of numeric values between to numbers and separated by a user-defined interval (e.g., a series of integers between 1 and 10).

To call a function, one must type the name of the function (e.g., ***seq***) followed in parentheses by a list of arguments to pass to the function. Arguments accepted by a function can be found in that function's help documentation (accessed by entering a **?** followed by the function name into the R console; e.g., ***?seq***). Arguments listed in the help file that don't have a default value are mandatory. The 'Usage' section of the help file typically lists which arguments have default values and which do not. The 'Value' section of the help documentation lists details of the function output(s). It is a good idea to familiarize yourself with the layout of the help files, as they are fairly consistent across packages.

Let's use the ***seq()*** function to generate a vector of sequential values between 1 and 10.

```
# Generate a vector of sequential values between 1 and 10.
seq(from=1, to=10, by=1)

##  [1]  1  2  3  4  5  6  7  8  9 10
```

By default, arguments in a function are evaluated in the order they appear in the default methods section of the help file. Therefore, if the arguments are supplied in exactly that order they do not need to be named. For example, the code below where arguments are not named, will produce the same results as above.

```
# Generate a vector of sequential values between 1 and 10.
seq(1, 10, 1)

##  [1]  1  2  3  4  5  6  7  8  9 10
```

R has many useful built-in functions for organizing, manipulating, visualizing, and analyzing data. In addition, there are many third-party packages containing functions that are useful for analyzing acoustic telemetry data. The easiest way to discover these packages and functions is to perform Google searches for solutions to issues or challenges you encounter. Asking your peers if they have encountered similar issues can also be helpful (no need to re-invent the wheel). Third-party packages are installed using the `install.packages()` function, and loaded using the `library()` function. When you run `install.packages()`, R attempts to download the requested package from the large repository of user-contributed packages on the Comprehensive R Archive Network (CRAN). Although the majority of R packages are housed on CRAN, some packages may be found hosted on other online repositories. Installation procedures for these packages vary.

R users can create their own function using the `function()` function. It is a good idea to make your code a function if you expect to re-use the same code multiple times, or if you plan to share specific code you've created with others. Creating functions is beyond the scope of this manual; however, more advanced users are encouraged to explore function making on their own. There are many helpful resources available in the form of books (see above for recommendations), and online blogs and forums.

At the end of this document in Appendix B, we have provided a list of existing functions that we have found useful for analyzing acoustic telemetry data. The list is not exhaustive, but it covers many of the common tasks applied to acoustic telemetry data. We will continue to add to the list as we encounter new functions. In addition, we have created a package called `glatos` that will serve as a repository for useful functions that can be used to quickly summarize and visualize data directly from the GLATOS data files. For more information on the `glatos` package, see Appendix A.

Data Types

R has a number of different data types. We will not discuss them all here, but we will introduce the most common data types you are likely to encounter when using R to analyze acoustic telemetry data. Table 1 describes the most common data types you will use in day-to-day R programming, though other types exist (e.g., integer, factor).

Table 1. Common data types used for analyses of acoustic telemetry data.

Type	Example	Notes
numeric	2, 19.3, 10300	Numeric data can be whole numbers, integers, or decimal numbers
character	"a", "hello", "this is a character string"	A character is a text character surrounded by single or double quotes. A character string is composed of two or more characters. Numbers can be characters if they are entered in quotes.
logical	TRUE, FALSE	Logical data are recognized as numeric data by a number of functions: TRUE = 1, FALSE = 0.
POSIXct	"2012-01-01 12:00:00 EST"	Data type used for time (can include dates also). Stored internally as the number of seconds since 1970-01-01 00:00:00 UTC, unless specified otherwise.

Date	"2012-01-01"	Date type used for dates. Stored internally as the number of days since 1970-01-01 UTC, unless otherwise specified.

Sometimes, you may want to convert data from one type to another. R has built-in functions (e.g., `as.numeric()`, `as.character()`, `as.logical()`, `as.POSIXct()`) to do this. For example, you can convert the number 5 to a character using the `as.character()` function. The R function `class()` can be used to view data type.

```
# Determine the type of data contained within parentheses.
class(5)

## [1] "numeric"

# Convert the number 5 to character and then display its data type.
class(as.character(5))

## [1] "character"
```

Logical data (i.e., TRUE, FALSE) are automatically coerced to numeric values in contexts that require numeric data (TRUE = 1, FALSE = 0). To demonstrate this, we will multiply the value TRUE by 2 (the * symbol means multiply). Note that R replaced the value TRUE with a 1 for this operation.

```
TRUE*2

## [1] 2
```

Dealing with Time in R: The as.POSIXct() Function

Time can be tricky in R. In most cases, when you bring date and time data (timestamps) into R, you will bring it in as a character string and will need to tell R that it is time data. This is accomplished using the `as.POSIXct()` function. This function converts a character string object to a POSIXct object (i.e., R's time class). The default format for datetime data in R is 'year-month-day hour:minute:second' ('2017-02-28 19:56:30'). If your datetime data is not in the default format, you must supply a `format` argument (e.g., `format='%Y-%m-%d %H:%M:%S'`) to `as.POSIXct()`. Table 2 contains a number of common formatting codes that can be used in the format argument - see the help file for `strptime()` for additional formatting codes.

BE CAREFUL WHEN MOVING BETWEEN R AND MS EXCEL: When MS Excel opens a file, and recognizes that data/time data are present, it converts those data to the format specified by the global settings of your operating system for viewing (often, this we be a form like 'mm/dd/yyyy'). If you then re-save that file, it saves the data in the new format, not the original format. When you then try to reload that file into R and convert the time stamps to `POSIXct` using the `as.POSIXct()` function, you may get the following error: 'character string is not in a standard unambiguous format'. When you see this error, it usually means your dates are not in 'yyyy-mm-dd hh:mm:ss' format and you must either convert them to that format in the original file, or specify `format` in the call to `as.POSIXct()`.

Note that if no time zone argument, `tz`, is supplied to `as.POSIXct()`, R assumes the datetime data are in the same time zone as your computer. To avoid making time zone errors, it is best to get into the habit of specifying `tz` every time you call `as.POSIXct()`, even if the time zone of the data is the same as the computer. For the Great Lakes, valid time zones include: 'GMT' = UTC time, 'US/Eastern'=Eastern time. 'US/Central'=Central time. You can also specify timezones using specific country/city abbreviations (i.e., 'America/Detroit' or 'America/New_york'). See https://en.wikipedia.org/wiki/List_of_tz_database_time_zones for a list of timezone abbreviations that are accepted by R. Note that 'GMT' and 'UTC' are equivalent and can be used interchangeably. Also note that

the time zone refers to a location, not an offset from UTC. Timestamps in R are interpreted as local time in the specified zone, so they account for daylight savings.

Table 2. Common formatting codes for use in the `format` *argument of the* `as.POSIXct()` *and* `as.Date()` *functions. For complete list of codes, see the help vignette for the* `strptime()` *function.*

Code	Description
%a	Abbreviated weekday name (also matches full name on input).
%A	Full weekday name (also matches abbreviated name on input).
%b	Abbreviated month name (also matches full name on input).
%B	Full month name (also matches abbreviated name on input).
%c	Date and time. Locale-specific on output, "%a %b %e %H:%M:%S %Y" on input.
%d	Day of the month as decimal number (01-31).
%D	Date format such as %m/%d/%y.
%e	Day of the month as decimal number (1-31), with a leading space for a single-digit number.
%F	Equivalent to %Y-%m-%d (the ISO 8601 date format).
%H	Hours as decimal number (00-23). Strings such as 24:00:00 are accepted for input.
%I	Hours as decimal number (01-12).
%j	Day of year as decimal number (001-366).
%m	Month as decimal number (01-12).
%M	Minute as decimal number (00-59).
%OS	Second as integer (00-61) with fractional seconds.
%p	AM/PM indicator in the locale. Used in conjunction with %I and not with %H.
%R	Equivalent to %H:%M.
%S	Second as integer (00-61), allowing for up to two leap-seconds.
%T	Equivalent to %H:%M:%S.
%u	Weekday as a decimal number (1-7, Monday is 1).
%V	Week of the year as decimal number (01-53) as defined in ISO 8601.
%x	Date. Locale-specific on output, "%y/%m/%d" on input.
%X	Time. Locale-specific on output, "%H:%M:%S" on input.
%y	Year without century (00-99). On input, values 00 to 68 are prefixed by 20 and 69 to 99 by 19.
%Y	Year with century (e.g., 1999).
%z	Signed offset in hours and minutes from UTC (e.g., -0800 is 8 hours behind UTC).
%Z	(Output only.) Time zone abbreviation as a character string (empty if not available).

Although POSIXct objects are displayed as datetime stamps, the data are stored internally as a number indicating the number of seconds that have passed since January 01, 1970 00:00:00 UTC. The ***`tz`*** argument tells R how to convert the incoming data to UTC and what time zone to use when displaying the data. By storing all POSIXct objects in UTC (regardless of incoming data time zone) mathematical operations involving multiple timestamp objects will be applied correctly and specific operations (except display or export) do not need to consider the original time zone of the data.

Let's convert a character object to POSIXct. We will specify that the timestamp is in UTC (i.e., GMT) time.

```
# Create a character string representing a specific timestamp.
tstamp <- '2017-02-20 10:10:06'
# View `tstamp` and return its class.
tstamp

## [1] "2017-02-20 10:10:06"

class(tstamp)

## [1] "character"

# Convert the `tstamp` character string to POSIXct.
tstamp <- as.POSIXct(tstamp, tz='GMT')
# View converted `tstamp` and return its class.
tstamp

## [1] "2017-02-20 10:10:06 GMT"

class(tstamp)

## [1] "POSIXct" "POSIXt"
```

We have successfully converted the character string to a POSIXct object.

As mentioned above, the new `tstamp` object looks like a timestamp, but internally it is a number representing the number of seconds that have passed since January 01, 1970 00:00:00 UTC time zone. To demonstrate this, we will display `tstamp` as a number using the `as.numeric()` function.

```
# Display the `tstamp` POSIXct object as a numeric value representing the
# number of seconds that have passed since January 01, 1970 00:00:00 UTC time
# zone.
as.numeric(tstamp)

## [1] 1487585406
```

Now, let's convert a character string that is not in the default format to POSIXct. This time we will specify Central time zone.

```
# Create a datetime character string.
tstamp2 <- '02/28/2017 01:24:40'
# Convert the character string to POSIXct, this time specifying the format of
# the character timestamp. If we try to run `as.POSIXct()` on this string
# without specifying the format, the function would return an error indicating
# that the character string is not in an unambiguous format.
tstamp2 <- as.POSIXct(tstamp2, format='%m/%d/%Y %H:%M:%S', tz='US/Central')
# View the converted datetime object.
tstamp2

## [1] "2017-02-28 01:24:40 CST"
```

Time zone matters. Incorrect specification of time zones in the `POSIXct()` call will result in errors. To demonstrate this, we will convert the same character string to POSIXct using two different `tz` arguments. We will then compare the numeric values of the two POSIXct objects to demonstrate that they are stored as different numbers by the computer.

```
# Create a timestamp character string.
tstamp3 <- '2017-02-01 17:28:56'
# Convert `tstamp3` to POSIXct with tz attribute to UTC time.
```

```
tstamp4 <- as.POSIXct(tstamp3, tz='GMT')
# Convert `tstamp3` to POSIXct with tz attribute to CST time.
tstamp5 <- as.POSIXct(tstamp3, tz='US/Central')
# Compare the numeric value of the two POSIXct objects.
as.numeric(tstamp4)

## [1] 1485970136

as.numeric(tstamp5)

## [1] 1485991736
```

Converting POSIXct Objects to a Different Time Zone

Often a user will wish to convert their data to a different time zone. For example, GLATOS detection data is exported in UTC time, but you might want to convert it to local time before plotting. To accomplish this, we will use the `attr()` function to change the `tzone` attribute (set by the `tz` argument in the call to `as.POSIXct`) of the POSIXct object.

Let's change the `tstamp2` POSIXct object, currently in CST, to UTC time.

```
# View tstamp2
tstamp2

## [1] "2017-02-28 01:24:40 CST"

as.numeric(tstamp2)

## [1] 1488266680

# Change the 'tzone' attribute of the `tstamp2` POSIXct object to 'GMT' (UTC).
attr(tstamp2, 'tzone') <- 'GMT'
# View tstamp2 with its new tzone attribute. Note that the stored value in
# seconds since January 01 1970 remains the same.
tstamp2

## [1] "2017-02-28 07:24:40 GMT"

as.numeric(tstamp2)

## [1] 1488266680
```

Note that the actual time (i.e., seconds since January 01, 1970 00:00:00 UTC time zone) did not change, only the time zone the object is displayed in changed. UTC time in Feb is 6 hours ahead of CST time.

Check your results carefully when working with timestamps in R. In many situations, a small typo when specifying timezones in your code will not result in an error. For example typing 'as.POSIXct('2018-02-18 02:30:00", tz ="america/detroit")'is incorrect and produces the wrong output yet does not result in an error (correct command: 'as.POSIXct("2018-02-18 02:30:00", tz="America/Detroit")')

The as.Date() Function

The `as.Date()` function is similar to the `as.POSIXct()` function except that it stores the object internally as the number of days since January 01, 1970 00:00:00 UTC time zone, rather than seconds. A practical use for this function is creating a grouping variable for summarizing data by day. The `as.Date()` function converts a character string object to a Date object. The default formats for date data in R are '2017-02-28' or '2017/02/28', so if the string you are converting is not in one of those two formats, you must supply a format

argument (e.g., `format='%Y-%m-%d'`) to `as.Date()` to tell R what format the datetime string is in. Table 2 contains a number of common formatting codes that can be using in the format argument - see the help file for `strptime()` for additional formatting codes. The time zone, `tz`, must also be supplied, or R will assume the time zone is that same as the computer. Below we will convert our `tstamp3` datetime character string to a 'Date' object and then convert it to a numeric value to demonstrate that it is stored internally as number of days since January 01, 1970 00:00:00 UTC time zone, rather than seconds.

```
# Convert `tstamp3` to a Date object.
tstamp6 <- as.Date(tstamp3, tz='GMT')
# Convert the Date object to numeric. Note that Date objects are stored
# internally as a number representing the number of days that have passed
# since January 01, 1970 00:00:00 UTC time zone.
as.numeric(tstamp6)

## [1] 17198
```

Math with POSIXct Objects

As mentioned previously, POSIXct objects are stored internally as numbers, so we can perform mathematical operations on them. A practical example of this is subtracting two timestamps to determine how much time elapsed between to them.

```
# Subtract tstamp from tstamp2 to determine the amount of time that elapsed.
tstamp2 - tstamp

## Time difference of 7.885116 days

# Display the time difference as a number.
as.numeric(tstamp2 - tstamp)

## [1] 7.885116
```

When you subtract two POSIXct objects, R outputs the difference in units it feels will be most useful to the user. In this example, R returned the difference in days, but it might have been returned in minutes, or even years. What if we want to use that difference in some other calculation? We need consistency. The simplest approach is to coerce the timestamps to number format (using the `as.numeric()` function) before subtracting them. When you do this, the difference will always be in seconds. You can always convert from seconds to other units after the fact.

```
# Subtract tstamp from tstamp2 to determine the amount of time that elapsed.
as.numeric(tstamp2) - as.numeric(tstamp)

## [1] 681274
```

Objects

R is an object-oriented scripting language. Data are stored as named objects in the global environment. You can think of objects as containers for data that are stored in your computer's memory. There are a number of object types, each with its own set of properties. We will briefly discuss the most common object types you will encounter in this section. Table 3 summarizes common objects, properties, and methods for creating them. More detailed information for each data object type is presented below. You can use the `class()` function to determine the type of an existing object in the global environment.

Table 3. Common objects, their properties, and methods used to create them.

Object	Properties	Method	Example
vector	1-dimensional, single data type	c()	c(1,2,3,4,5)
		":"	01:05
		seq()	seq(from=1, to=5, by=1)
matrix	2-dimensional, single data type	matrix()	matrix(1:16, nrow=4)
		as.matrix()	as.matrix(1:10)
list	1-demensional, multiple data types	list()	list(a=1, b=2, c=3, d=4, e=5)
		as.list()	as.list(1:5)
data frame	2-demensional, multiple data types	data.frame()	data.frame(letters=letters[1:5], numbers=numbers[1:5])
		as.data.frame()	as.data.frame(matrix(1:16, ncol = 4))
		read.csv()	read.csv('c:/Users//UserName/Desktop/test.csv')
		read.table()	read.table('c:/Users//UserName/Desktop/test.csv', sep = " ")

Vectors

The most common object type is a vector. Vectors are one-dimensional objects that can contain various types of data (e.g., numeric, character), but all elements within a single vector must be of the same type. For example, you cannot create a vector with both numeric and character data; in this case, the numeric data will be coerced to characters. A vector is a container for data, similar to how a tennis ball tube is a container for tennis balls. Each ball in the tube is an element of the vector. Elements in a vector are indexed by their position. A one-element vector is sometimes called a scalar.

There are a number of ways to create a vector. Here we present three common ways:

1. Create a vector by manually typing out each value in the vector.

To combine values manually into a vector we must surround the individual values in `c()`, separated by commas. `c()` is a built-in R function that combines the comma-separated values entered within the parentheses into a single vector of values. In most cases, if you are creating a vector, you will use the `c()` (combine) function.

```
# Create a character string vector consisting of VEMCO transmitter ids.
transmitter_ids <- c('A69-9001-15344', 'A69-9001-15345', 'A69-9001-15346',
           'A69-9001-15347', 'A69-9001-15348')
# Display the vector in the R Console.
transmitter_ids

## [1] "A69-9001-15344" "A69-9001-15345" "A69-9001-15346" "A69-9001-15347"
## [5] "A69-9001-15348"
```

2. Create a vector using colon notation.

Often, we may wish to create a numeric vector of sequential integers. This can be accomplished using colon(`:`) notation. The colon operator returns a vector of sequential integers ranging between the two

integers provided on either side of the colon. Note that the colon notation returns a vector by default, so you do not need to enclose the code in `c()`.

```
# Create a vector of sequential values between 1 and 20.
numbers <- 1:20
# Display the vector.
numbers

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

3. Use the `seq()` function to create a vector of sequential values separated by regular intervals.

The `seq()` function is another built-in R function. Similar to the colon operator (`:`), `seq()` returns a vector by default, so you are not required to use `c()`. The `seq()` function returns a sequence of numeric values between two numbers, separated by a user-determined interval.

```
# Create a vector containing every other value between 1 and 100 using
# the `seq()` function.
sequence <- seq(from=1, to=100, by=2)
# Display the vector.
sequence

##  [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45
## [24] 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91
## [47] 93 95 97 99
```

Aside: Binning Data Using the seq() and findInterval() Functions

The `findInterval()` function is a very useful function for organizing data into discrete bins. A common practical application of this function for telemetry data is to bin detection data by timestamp for summarizing detection data over time. This is a two-step process. First, we create a vector of POSIXct data representing the start time of each time bin using the `seq()` function. Second, we use the `findInterval()` function to assign each detection to a specific time bin (i.e., interval) in the sequence vector.

For example, we will create a sequence of timestamps separated by 1 day. Recall that POSIXct objects are stored as numeric values representing the number of seconds since January 01, 1970, so we need our separator to be in seconds also (12 hours = 86400 seconds). However, the `by` argument of the `seq()` function can also accept character strings like ‘hour’, ‘day’, ‘week’, ‘month’, ‘quarter’, and ‘year’, which is convenient when dealing with timestamps.

```
# Create a vector of timestamps between Jan 01, 2017 and Jun 01, 2017,
# separated by 1 day.
time_bins <- seq(from=as.POSIXct('2017-01-01 00:00:00', tz='GMT'),
         to=as.POSIXct('2017-06-01 00:00:00', tz='GMT'),
         by='day')
# Display the time_bins vector. The head() function outputs the first 20
# elements of the vector (default is 6).
head(time_bins, 20)

##  [1] "2017-01-01 GMT" "2017-01-02 GMT" "2017-01-03 GMT" "2017-01-04 GMT"
##  [5] "2017-01-05 GMT" "2017-01-06 GMT" "2017-01-07 GMT" "2017-01-08 GMT"
##  [9] "2017-01-09 GMT" "2017-01-10 GMT" "2017-01-11 GMT" "2017-01-12 GMT"
## [13] "2017-01-13 GMT" "2017-01-14 GMT" "2017-01-15 GMT" "2017-01-16 GMT"
## [17] "2017-01-17 GMT" "2017-01-18 GMT" "2017-01-19 GMT" "2017-01-20 GMT"
```

Note that a positive or negative integer followed by a space can be added to a character string supplied to the ***by*** argument to create a sequence of timestamps separated by a multiple of the time unit. An 's' after the character string is optional.

For example, we can make the same vector as above, but this time we will separate the timestamps by seven days instead of just one.

```
# Create a vector of timestamps between Jan 01, 2017 and Jun 01, 2017,
# separated by 7 days.
time_bins_7 <- seq(from=as.POSIXct('2017-01-01 00:00:00', tz='GMT'),
          to=as.POSIXct('2017-06-01 00:00:00', tz='GMT'),
          by='7 days')
# Display the time_bins_7 vector.
time_bins_7
```

```
##  [1] "2017-01-01 GMT" "2017-01-08 GMT" "2017-01-15 GMT" "2017-01-22 GMT"
##  [5] "2017-01-29 GMT" "2017-02-05 GMT" "2017-02-12 GMT" "2017-02-19 GMT"
##  [9] "2017-02-26 GMT" "2017-03-05 GMT" "2017-03-12 GMT" "2017-03-19 GMT"
## [13] "2017-03-26 GMT" "2017-04-02 GMT" "2017-04-09 GMT" "2017-04-16 GMT"
## [17] "2017-04-23 GMT" "2017-04-30 GMT" "2017-05-07 GMT" "2017-05-14 GMT"
## [21] "2017-05-21 GMT" "2017-05-28 GMT"
```

Next, we will use the ***findInterval()*** function to determine in what time bin each of a series of timestamps occurred. We first need to create a vector of timestamps to compare against the ***time_bins_7*** vector.

```
# Create a vector of timestamps to compare against the `time_bins_7` vector.
timestamps <- as.POSIXct(c('2017-01-01 12:34:00', '2017-02-01 18:12:00',
              '2017-03-01 09:01:00', '2017-04-01 11:37:00',
              '2017-05-01 02:14:00'), tz='GMT')
# Display the timestamps vector.
timestamps
```

```
## [1] "2017-01-01 12:34:00 GMT" "2017-02-01 18:12:00 GMT"
## [3] "2017-03-01 09:01:00 GMT" "2017-04-01 11:37:00 GMT"
## [5] "2017-05-01 02:14:00 GMT"
```

Now we use ***findInterval()*** to assign each timestamp to a time bin in the ***time_bins_7*** vector.

```
# Create a vector containing the interval (within the `time_bins_7` vector)
# for each element in the timestamp vector using the `findInterval()`
# function.
intervals <- findInterval(timestamps, time_bins_7)

# Display the intervals vector.
intervals
```

```
## [1]  1  5  9 13 18
```

Note that the ***intervals*** vector is the same length as the ***timestamps*** vector. See ***?findInterval*** for optional arguments that control inclusion of values at interval boundaries.

Matrices

A matrix is a two-dimensional object for storing data. A matrix stores data in grid form, which is indexed by its rows and columns. Like vectors, matrices can contain various types of data, but all data in a single matrix

must be of the same type. Matrices are often used for gridded data and for doing fast mathematical operations, but they are used much less commonly than vectors and data frames (see below), so we will not spend much time on them here, other than to introduce them and demonstrate how they are created. We encourage interested readers explore matrices further on their own.

Here, we will create a matrix containing the values 1 through 50. The `matrix()` function converts a vector to a matrix with user-defined dimensions. The `byrow` argument tells R whether to populate the matrix by row or by column. By default, matrices are populated and read by column.

```
# Create a matrix with values from 1 through 50. The matrix will have 10 rows
# and will be filled by column.
v_matrix <- matrix(1:50, nrow=10, byrow=FALSE)
# View the matrix.
v_matrix
```

```
##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    1   11   21   31   41
##  [2,]    2   12   22   32   42
##  [3,]    3   13   23   33   43
##  [4,]    4   14   24   34   44
##  [5,]    5   15   25   35   45
##  [6,]    6   16   26   36   46
##  [7,]    7   17   27   37   47
##  [8,]    8   18   28   38   48
##  [9,]    9   19   29   39   49
## [10,]   10   20   30   40   50
```

Lists

A list is an object that can contain multiple objects with differing data types. Structurally, lists are similar to vectors, except that each element of the list contains an entire object (or even list of objects) rather than a single value. Lists are often used to return multiple objects from a function. If you have ever used R to run statistical test like ANOVAs and t-test, you will likely have encountered lists before in their outputs.

Here we provide a simple example of a list object that contains two matrices, but understand that the list can contain any number of different objects.

```
# Create two matrices: one with 3 rows and 4 columns, and a second 4x4.
matrix_1 <- matrix(seq(1,15), nrow=3, ncol=5)
matrix_2 <- matrix(seq(16,31), nrow=4)
# Store the two matrices in a list object named `example_list`.
example_list <- list (matrix_1, matrix_2)
# View `example_list`.
example_list
```

```
## [[1]]
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15
##
## [[2]]
##      [,1] [,2] [,3] [,4]
## [1,]   16   20   24   28
## [2,]   17   21   25   29
```

```
## [3,]   18   22   26   30
## [4,]   19   23   27   31
```

Elements in a list can also be named. Names can be assigned after the list has been created using the `names()` function. Below, we will name the two matrices in list `example_list` using `names()`. `names()` can also be used to change existing list names.

```
# Assign names to the elements in `example_list`.
names(example_list) <- c('m1', 'm2')
# View `example_list`.
example_list

## $m1
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15
##
## $m2
##      [,1] [,2] [,3] [,4]
## [1,]   16   20   24   28
## [2,]   17   21   25   29
## [3,]   18   22   26   30
## [4,]   19   23   27   31
```

List names can also be assigned when a list is created.

```
# Create a named list containing 2 matrices.
example_list_2 <- list(matrix_1=matrix_1, matrix_2=matrix_2)
# View `example_list_2`.
example_list_2

## $matrix_1
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15
##
## $matrix_2
##      [,1] [,2] [,3] [,4]
## [1,]   16   20   24   28
## [2,]   17   21   25   29
## [3,]   18   22   26   30
## [4,]   19   23   27   31
```

A beginning R user is more likely to read from lists rather than create them, so we encourage readers to read the section below on subsetting objects (Table 4 in particular) for hints about accessing data that is stored in list objects.

Data frames

Data frames are objects that will be familiar to most researchers because their appearance is similar to spreadsheets in Microsoft Excel. A data frame is a two-dimensional object (technically a list of vectors), but unlike matrices that are also two-dimensional, the columns in a data frame can each contain data of different classes. Another difference between data frames and matrices is that data frames have column and row

names that can be used to subset data (using named list notation: `$`). Matrices can have user-specified column and row names also, but by default they do not nor are matrices able to be subset using list notation.

Here we'll use the `data.frame()` function to create an example data frame containing biological data for tagged fish. Note that the column headers can be assigned in the call to `data.frame()`. If the inputted column data are objects, `data.frame()` will use the object name as the column header unless otherwise specified.

In our call to `data.frame()` we will pass the argument `stringsAsFactors=FALSE`. The default behavior of `data.frame()` is to convert character strings to factors. The `stringsAsFactors=FALSE` argument prevents this behavior. A factor is a vector whose elements must belong to a specific set of values, referred to as levels. Factors can be difficult to distinguish from character strings, but can be identified by running `str()` on a data object to view the data structure. While there are times when using factors is advantageous (e.g., statistical modeling), use of factors should generally be avoided because many functions are not designed to work with factors and manipulating factors can be difficult. Most functions that require factor input will automatically coerce character strings to factors for you. It is best to avoid factors unless they are specifically required.

Let's build our data frame of biological data. The `rep()` function returns a vector with `x` replicated `times` number of times.

```
# Create a data frame using the data.frame() function. Note that strings are
# automatically converted to factors. This behavior can be prevented using the
# `stringsAsFactors` argument.
bio_data <- data.frame(fish_id=seq(from=1, to=10, by=1),
            transmitter_code=c('A69-9001-15123', 'A69-9001-15124',
                     'A69-9001-15125', 'A69-9001-15126',
                     'A69-9001-15127', 'A69-9001-15128',
                     'A69-9001-15129', 'A69-9001-15130',
                     'A69-9001-15131', 'A69-9001-15132'),
            tag_date=rep(x='2017-02-10', times=10),
            length=c(53.7, 64.4, 53.5, 63.8, 64.2, 55.0, 66.4,
                64.0, 56.2, 65.1),
            age=c(12, 14, 9, 14, 10, 10, 7, 14, 8, 14),
            stringsAsFactors=FALSE)
# View `bio_data`.
head(bio_data)

##   fish_id transmitter_code   tag_date length age
## 1       1   A69-9001-15123 2017-02-10   53.7  12
## 2       2   A69-9001-15124 2017-02-10   64.4  14
## 3       3   A69-9001-15125 2017-02-10   53.5   9
## 4       4   A69-9001-15126 2017-02-10   63.8  14
## 5       5   A69-9001-15127 2017-02-10   64.2  10
## 6       6   A69-9001-15128 2017-02-10   55.0  10
```

Rows can be appended to an existing data frame using the `rbind()` function. For example, suppose we tagged a new fish and wanted to add the data for that fish to `bio_data` (i.e., we want to add a new row of data to `bio_data`). We will first create a one-row data frame containing the relevant data for our fish and then append that data frame to `bio_data` using `rbind()`. Note that the number of columns and column names in the two data frames must be the same. Attempting to call `rbind()` on two data frames with different number of columns or different names will result in an error.

```
# Create a data frame for our newly-tagged fish, which will be appended to
# `bio_data`.
```

```
new_fish <- data.frame(fish_id=11,
            transmitter_code='A69-9001-15133',
            tag_date='2017-02-14',
            length=59.6,
            age=10,
            stringsAsFactors=FALSE)
# Append the new data to `bio_data`.
bio_data <- rbind(bio_data, new_fish)
# View last 6 rows of `bio_data`.
tail(bio_data)

##    fish_id transmitter_code   tag_date length age
## 6        6   A69-9001-15128 2017-02-10   55.0  10
## 7        7   A69-9001-15129 2017-02-10   66.4   7
## 8        8   A69-9001-15130 2017-02-10   64.0  14
## 9        9   A69-9001-15131 2017-02-10   56.2   8
## 10      10   A69-9001-15132 2017-02-10   65.1  14
## 11      11   A69-9001-15133 2017-02-14   59.6  10
```

Columns from two data frames can be combined into a single data frame with `cbind()`. Although this is useful, you typically add columns to a data frame one at a time as they are created using named list (`$`) notation.

For example, let's add a column to our `bio_data` data frame indicating the release location for each fish using `$` notation. If the column name indicated in the named list assignment statement already exists, the existing column of data will be overwritten by the new data. If the column does not already exist, R will append the new column to the data frame. Note that the length of the vector being appended must be equal to the number of rows in the original data frame.

```
# Append a vector of release locations to `bio_data`.
bio_data$release_loc <- c('L. Huron', 'L. Huron', 'L. Michigan',
             'L. Michigan', 'L. Huron', 'L. Michigan',
             'L. Michigan', 'L. Huron', 'L. Huron',
             'L. Michigan', 'L. Huron')
# View bio_data.
head(bio_data)

##   fish_id transmitter_code   tag_date length age release_loc
## 1       1   A69-9001-15123 2017-02-10   53.7  12    L. Huron
## 2       2   A69-9001-15124 2017-02-10   64.4  14    L. Huron
## 3       3   A69-9001-15125 2017-02-10   53.5   9 L. Michigan
## 4       4   A69-9001-15126 2017-02-10   63.8  14 L. Michigan
## 5       5   A69-9001-15127 2017-02-10   64.2  10    L. Huron
## 6       6   A69-9001-15128 2017-02-10   55.0  10 L. Michigan
```

Now, let's look at an example where we overwrite a column of data using `$` notation. Suppose we are planning to tag fish over several years and want to append a year to the `fish_id`. We can create a new vector of fish ids and then replace the original `fish_id` column with the new one in our data frame.

```
# Overwrite the fish_id column in `bio_data` with fish ids that have tagging
# year appended.
bio_data$fish_id <- c('2017_001', '2017_002', '2017_003','2017_004',
           '2017_005', '2017_006', '2017_007', '2017_008',
           '2017_009', '2017_010', '2017_011')

# View `bio_data`. Note that the `fish_id` column now contains ids with
```

```
# tagging year appended.
head(bio_data)

##    fish_id transmitter_code   tag_date length age release_loc
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron
## 2 2017_002   A69-9001-15124 2017-02-10   64.4  14    L. Huron
## 3 2017_003   A69-9001-15125 2017-02-10   53.5   9 L. Michigan
## 4 2017_004   A69-9001-15126 2017-02-10   63.8  14 L. Michigan
## 5 2017_005   A69-9001-15127 2017-02-10   64.2  10    L. Huron
## 6 2017_006   A69-9001-15128 2017-02-10   55.0  10 L. Michigan
```

Subsetting Objects

Subsetting is the basis for virtually all data manipulations in R. Often, we will want to access a subset of the data stored in an object and use the extracted data in additional calculations. For example, we may want to extract detections for a particular transmitter and calculate the number of receivers visited, or view the biological data for one sex or age class. There are a variety of methods for subsetting objects, but generally they fall into three categories: 1) square bracket subsetting, 2) named list subsetting, and 3) logical subsetting. In square bracket subsetting, data are subsetted using numeric column and row indices, or when available, name attributes (e.g., column names in data frames). In Table 4, we summarize commonly used subsetting methods for each data object type. There are number of skill-building exercises dealing with subsetting objects in Appendix A. Answers to the exercises are located in Appendix B.

Table 4. Methods used to subset data from common data objects.

Object	Method	Notation	Example	Notes
vector	square bracket	[index]	V1[c(1,3,5,7)]	subset indices inputted as vector
matrix	square bracket (2D)	[rows index, columns index]	M1[1:20, 1]	empty subset value means ‘all rows’ or ‘all columns’
list	square bracket	[index]	L1[2]	returns list element as list object
	double square bracket	[[ ]]	L1[[2]]	returns actual value of an element
	named list	$	L1$EL2	returns actual value of an element
data frame	square bracket (2D)	[rows, columns]	DF1[1:20, 1]	empty subset value means ‘all rows’ or ‘all columns’
	square bracket with column/row names	[, ‘Column1’]	DF1[, ‘column_name’]	column/row names can be used in the square bracket
	named list	$	L1$column_name	columns subsetted using named list notation
	named list and square bracket	$ and [ ]	DF1$column_name[c(1,3,5,7)]	columns subsetted using names list notation behave like a vector

To demonstrate the various subsetting methods, we will return to our ***bio_data*** data frame object, which can be subsetted using both square bracket and named list notation. Here are the contents of ***bio_data***:

```
# View bio_data.
head(bio_data)

##    fish_id transmitter_code   tag_date length age release_loc
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron
## 2 2017_002   A69-9001-15124 2017-02-10   64.4  14    L. Huron
## 3 2017_003   A69-9001-15125 2017-02-10   53.5   9 L. Michigan
## 4 2017_004   A69-9001-15126 2017-02-10   63.8  14 L. Michigan
## 5 2017_005   A69-9001-15127 2017-02-10   64.2  10    L. Huron
## 6 2017_006   A69-9001-15128 2017-02-10   55.0  10 L. Michigan
```

1. Subsetting rows and columns using square bracket subsetting.

We will first explore various ways to subset data frames using square bracket notation. Because data frames are 2-dimensional objects, we must provide two comma-separated index values within the square brackets. The first value corresponds to rows, and the second value corresponds to columns. The index values can be a single index value (or name), or a vector of index values (or names). An empty index tells R to return all rows or columns.

Examples:

First, we will use square bracket notation to subset the first row of ***bio_data***. Note that we do not supply a value for columns, so R returns all columns in that row. Although we only provide examples of square bracket subsetting with data frames, this notation is used to extract subsets from matrices, lists, and vectors.

```
# Subset the first row of bio_data.
bio_data[1,]

##    fish_id transmitter_code   tag_date length age release_loc
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron
```

Now we will subset the ***transmitter_code*** from the first row of ***bio_data***.

```
# Subset the transmitter_code from the first row of bio_data.
bio_data[1,2]

## [1] "A69-9001-15123"
```

Instead of a numeric index value, we can use the column names attribute to subset the data frame. Let's subset out the ***transmitter_code*** column again, this time using the column name attribute.

```
# Subset the transmitter_code column of bio_data.
bio_data[,'transmitter_code']

##  [1] "A69-9001-15123" "A69-9001-15124" "A69-9001-15125" "A69-9001-15126"
##  [5] "A69-9001-15127" "A69-9001-15128" "A69-9001-15129" "A69-9001-15130"
##  [9] "A69-9001-15131" "A69-9001-15132" "A69-9001-15133"
```

What if we want to subset the first and second rows of ***bio_data***? We can supply a vector of indexes to subset more than one row or column.

```
# Subset the first and second rows of bio_data.
bio_data[c(1,2),]
```

```
##    fish_id transmitter_code   tag_date length age release_loc
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron
## 2 2017_002   A69-9001-15124 2017-02-10   64.4  14    L. Huron
```

2. Subsetting using named list notation.

List and data frame objects can be subsetted using named list notation (**`$`**). For data frames, named list notation returns a single column of data as a vector. For lists, named list notation returns a single named element of the list in the form it is stored, unlike square bracket notation which returns list elements as list objects unless double square brackets are used (see Table 4 above).

Subset **`transmitter_code`** using named list notation:

```
# Subset the transmitter_code from bio_data.
bio_data$transmitter_code

##  [1] "A69-9001-15123" "A69-9001-15124" "A69-9001-15125" "A69-9001-15126"
##  [5] "A69-9001-15127" "A69-9001-15128" "A69-9001-15129" "A69-9001-15130"
##  [9] "A69-9001-15131" "A69-9001-15132" "A69-9001-15133"
```

Note that we can combine named list notation and square bracket notation to subset data from objects. For example, lets subset the third value in the **`transmitter_code`** column of **`bio_data`**. Recall that applying named list notation to data frames returns the specified column as a vector.

```
# Subset the third value from the transmitter_code column of bio_data.
bio_data$transmitter_code[3]

## [1] "A69-9001-15125"
```

3. Subsetting with Logic Statements

Logic statements are an important component of programming. When writing logic statements, we use logical operators (Table 5) to evaluate whether a statement is TRUE or FALSE. Each logic statement will contain one or more logical operators.

Table 5. Logical operators in R.

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or
equ	al to
==	equal to
!=	not equal to
%in%	within
&	and
|	or

An important practical application for logic statements is data subsetting. Logic statements allow you to subset data based on specific criteria. Data returned are those elements of the object that evaluate to TRUE.

Consider our fish data frame, ***bio_data***.

```
head(bio_data)

##    fish_id transmitter_code   tag_date length age release_loc
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron
## 2 2017_002   A69-9001-15124 2017-02-10   64.4  14    L. Huron
## 3 2017_003   A69-9001-15125 2017-02-10   53.5   9 L. Michigan
## 4 2017_004   A69-9001-15126 2017-02-10   63.8  14 L. Michigan
## 5 2017_005   A69-9001-15127 2017-02-10   64.2  10    L. Huron
## 6 2017_006   A69-9001-15128 2017-02-10   55.0  10 L. Michigan
```

Suppose we only want to return fish that were tagged in Lake Huron. In other words, we only want to return rows where the column ***release_loc == 'L. Huron'***.

```
# Subset out fish that have `release_loc == 'L. Huron'`.
bio_data[bio_data$release_loc == 'L. Huron',]

##     fish_id transmitter_code   tag_date length age release_loc
## 1  2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron
## 2  2017_002   A69-9001-15124 2017-02-10   64.4  14    L. Huron
## 5  2017_005   A69-9001-15127 2017-02-10   64.2  10    L. Huron
## 8  2017_008   A69-9001-15130 2017-02-10   64.0  14    L. Huron
## 9  2017_009   A69-9001-15131 2017-02-10   56.2   8    L. Huron
## 11 2017_011   A69-9001-15133 2017-02-14   59.6  10    L. Huron
```

We can subset based on more than one criterion. For example, suppose we want to return only those fish released in Lake Huron that were also less than 10 years old.

```
# Subset out fish with `release_loc == 'L. Huron'` AND `age < 10`.
bio_data[bio_data$release_loc == 'L. Huron' & bio_data$age < 10,]

##    fish_id transmitter_code   tag_date length age release_loc
## 9 2017_009   A69-9001-15131 2017-02-10   56.2   8    L. Huron
```

Note that the logical expression within the square brackets returns an object of the same dimensions as the object being tested. To demonstrate this, we will run just the logical expression and view the output.

```
# Run just the logical expression.
bio_data$release_loc == 'L. Huron' & bio_data$age < 10

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
```

The expression returns a logical vector (TRUE and FALSE) of the same length as ***bio_data$release_loc***.

Using named list notation can be used to subset using multiple logical constraints. The code below selects all fish with 'fish_id == 1' or 'fish_id == 2'. Two things to note: first we use a double equals ('==') to specify logical comparison. A single '=' is the assignment operator in R. Second we have to repeat the 'bio_data$fish_id' bit for each comparison. This works fine for small number of logical comparison but is a bit problematic is you need to feed a large number of comparisons to R. The match function (***%in%***) provides a solution for this problem (see code below).

```
# Multiple logical comparisons.
bio_data[bio_data$fish_id == '2017_001' | bio_data$fish_id == '2017_002',]
```

```
##    fish_id transmitter_code   tag_date length age release_loc
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron
## 2 2017_002   A69-9001-15124 2017-02-10   64.4  14    L. Huron

# Example of match (%in%).
fish <- c('2017_001', '2017_002')
bio_data[bio_data$fish_id %in% fish,]

##    fish_id transmitter_code   tag_date length age release_loc
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron
## 2 2017_002   A69-9001-15124 2017-02-10   64.4  14    L. Huron
```

Aside: Missing Data - NA and NULL

R has two types of missing data, NA and NULL, which behave very differently from one another. NA is used to identify missing values in an object. For example, if we failed to record a length for one of the fish in `bio_data`, then that individual would receive an NA in the length column of `bio_data`. The term NULL is reserved for NULL objects. NULL objects are objects that have length=0 and have no attributes, i.e., they contain no data.

The following code should help to clarify the difference between NA and NULL. We will create two numeric vectors, one containing an NA and a second containing a NULL value.

```
# Create two vectors, one with an NA value and the other with a NULL value.
na_vect <- c(1, 2, 3, NA, 5)
null_vect <- c(1, 2, 3, NULL, 5)
# View the two vectors.
na_vect

## [1]  1  2  3 NA  5

null_vect

## [1] 1 2 3 5
```

In our `null_vect` vector, addition of the NULL value to the vector literally means 'add nothing', so the resulting vector is only 4 elements long. The `na_vect` vector in the other hand is 5 elements long, and retains the NA value.

Dealing with NA values in data can be tricky. For example, if you try to apply the `mean()` function to a vector of data that contains NA values, R will return a result of NA.

```
# Calculate the mean of `na_vect` vector, which contains an NA value.
na_vect

## [1]  1  2  3 NA  5

mean(na_vect)

## [1] NA
```

Fortunately, many functions allow users to strip NA values before computations proceed. This is done by passing the argument `na.rm` to the function call. The `na.rm` argument is a logical value (TRUE or FALSE) indicating whether NA values should be stripped from the data before computation.

```
# Calculate the mean of a vector containing an NA value. `na.rm=TRUE` strips
# NAs from the vector before the mean is calculated.
mean(na_vect, na.rm=TRUE)

## [1] 2.75
```

Elements that contain NAs can also be removed from objects using `na.omit()`. For example, in a vector, `na.omit()` will remove NA values, but in a data frame, `na.omit()` will remove rows that contain NA.

```
# Remove NAs from a vector and data frame using `na.omit()`.
# View `na_vect`.
na_vect

## [1]  1  2  3 NA  5

# Remove NAs from `na_vect` using `na.omit()`.
na.omit(na_vect)

## [1] 1 2 3 5
## attr(,"na.action")
## [1] 4
## attr(,"class")
## [1] "omit"

# Create a data frame, `df` containing NA values.
df <- data.frame(x=numbers[1:5], y=c(letters[1:3], NA, letters[5]))
# View `df`.
df

##   x    y
## 1 1    a
## 2 2    b
## 3 3    c
## 4 4 <NA>
## 5 5    e

# Remove rows containing NAs from df` using `na.omit()`.
na.omit(df)

##   x y
## 1 1 a
## 2 2 b
## 3 3 c
## 5 5 e
```

Another important issue with NA values is that they evaluate as TRUE when subsetting with logic statements.

For example, let's take our vector from above and subset out elements that are greater than 2.

```
# Create a numeric vector containing NA.
p <- c(1, 2 ,3 , NA, 4)
# Subset out values greater than 2.
p[p > 2]

## [1]  3 NA  4
```

Notice that the NA value appeared in the returned vector. To address this issue, we must add a second logic statement (`!is.na(p)`) to the subset call to filter out NA values. `!is.na(p)` translates to: elements in p that are not NA.

```
# Subset out values less than or equal to 2 AND not equal to NA.
p[p <= 2 & !is.na(p)]

## [1] 1 2
```

Repeated Operations

Often, we will want to perform the same operations on data more than once. Whenever you find yourself typing the same code more than once in a script, there is a good chance you should be using a loop. For the purpose of this workshop we will focus on `for` loops, but be aware that there are other less commonly used looping functions available, including: `while` and `repeat`. Please consult R's help documentation for further information on these types of loops.

The For Loop

The most common type of loop in R is the `for` loop. The basis of the `for` loop is that the user defines a variable that iterates through a supplied vector, sequentially assigning a new value to the variable each time the loop is run. The loop continues until every value in the vector has been assigned to the variable (or the loop is broken manually with the `break` command). Most commonly, the variable is an indexing variable (e.g., i or j) and the vector is a sequence of numbers that define how many times the loop will run. The index variable is used to determine when the loop breaks, but it can also be used within the loop to perform operations on data.

Here we present a simple example of a loop. We will create a vector of 10000 random numbers between 1 and 100. We will then loop through the vector and add 1 to each element. We will use the `head()` function to view the first 10 elements in our vector.

```
# Set seed so that random sampling is reproducible.
set.seed(45)

# Create a 10-element vector of random numbers between 1 and 100.
j <- sample(1:100, 10, replace=TRUE)
# Show the first 10 elements of the vector.
j

##  [1] 64 32 25 38 36 30 23 56 19  1

# Loop through the vector and add 1 to each value. `length()` returns the
# length of the vector `j` (i.e., the number of elements in `j`).
for(i in 1:length(j)){
 j[i] <- j[i] + 1
}
j

##  [1] 65 33 26 39 37 31 24 57 20  2
```

The vector that is iterated through in a loop does not have to be a vector of numbers. One can loop through character strings, or even objects.

Here is a simple example that loops through a vector of character strings and prints them in the R console.

```
# Example of looping through a vector of character strings.
for(i in c('sunny', 'cloudy', 'rainy')){
 print(i)
}

## [1] "sunny"
## [1] "cloudy"
## [1] "rainy"
```

Note that in both examples above, the variable `i` was used in the code within the loop. In each iteration of the loop the value of `i` changed. In the first example, `i` was used an index to subset a vector of numeric values. This is a very common use for the looping variable. In the second example, each time the loop ran, `i` was assigned a different character string, which was then printed to the R console.

Aside: Loops vs. Vectorization

You may recall that the process where we looped through our 10000-element vector and added 1 to each element could have been performed using vectorization (i.e., we could have added 1 to the entire column at once). Your scripts will run faster if you can vectorize your operations. We'll demonstrate this by comparing the time it takes to multiply the values in `j` by -1 using a loop to the same operation using vectorization. The `Sys.time()` function returns the current time on your computer, which is useful for timing scripts. We will start with an NA vector, `k`, of length 10000 each time, and populate `k` with the results of multiplying `j` by -1.

First, we will perform the process in a loop, keeping track of both the start and end times.

```
# Create a 10000-element vector of NAs that will be populated with our
# calculated values.
k <- rep(NA, 10000)
# Log the start time for the loop.
loop_start <- Sys.time()
for(i in 1:10000){
 k[i] <- j[i]*-1
}
# Log the end time for the loop.
loop_end <- Sys.time()
# View first 10 elements of `k`.
head(k, 10)

##  [1] -65 -33 -26 -39 -37 -31 -24 -57 -20  -2
```

Next, we'll use vectorization, and will record both the start and end times.

```
# Reset the NA vector so we have the same starting point.
k <- rep(NA, 10000)
# Log the start time of the vectorized process.
vectorize_start <- Sys.time()
k <- j*-1
# Log the end time of the vectorized process.
vectorize_end <- Sys.time()
# View first 10 elements of `k`.
head(k, 10)

##  [1] -65 -33 -26 -39 -37 -31 -24 -57 -20  -2
```

Let's now compare the amount of time it took to run each process by subtracting the start and end time of each process.

```
# Calculate the time for each process to run.
loop_end - loop_start

## Time difference of 0.009074926 secs

vectorize_end - vectorize_start

## Time difference of 0.001652956 secs
```

Notice that the vectorized process is MANY times faster than the loop. This might not seem like a big deal when you are dealing with scripts that run very quickly (seconds saved), but consider how much time could be saved when you get to the point you are writing scripts that take minutes to hours to run (common when you are dealing with large datasets or are performing complex data manipulations). The first step in programming is to create something that works, but if you find yourself waiting a long time for scripts to run, it might be worth the effort to attempt to improve the efficiency of your code through vectorization.

The apply() and sapply() Functions

If you find yourself in a situation where you are going to loop through the elements of an object and apply a function (i.e. process) that can't be vectorized, you could use the `apply` or `sapply()` functions instead.

The `apply()` function allows you to apply functions across rows or columns of a data frame or matrix. The function loops through an object by row (or column, depending on what value is passed with the `MARGIN` argument) and applies a user-defined function to the values that appear in those rows. Results are combined into an object of the same length as rows or columns in the original object.

For example, suppose we have a matrix of mean water depths, each row representing a latitude bin and each column representing a longitude bin. You can think of the above matrix as an ASCII file (commonly used format in spatial data) where the values represent water depth for each combination of latitude (i.e., rows) and longitude (i.e., columns).

```
# Set seed so that random number generation is reproducible.
set.seed(45)
# Use the `runif()` function to generate a matrix of 16 hypothetical water
# depths from a uniform distribution ranging from -30 (min) to 0 (max).
water_depths <- matrix(runif(16, -30, 0), nrow=4)
# View the matrix.
water_depths

##           [,1]      [,2]       [,3]      [,4]
## [1,] -10.99882 -19.43567 -24.463075 -20.75842
## [2,] -20.47390 -21.06724 -29.841457 -16.21340
## [3,] -22.77234 -23.16542 -18.933294 -17.07833
## [4,] -18.64758 -13.35474  -3.586065 -17.68273
```

We could use the `apply()` function to determine what the maximum water depth is across each latitude bin (i.e., by row). Within `apply()` we are defining a function (`FUN`) that has input `x` (in this case column of data - set by `MARGIN` argument), and finds the minimum value using the base function `min()` (since water depths are negative numbers).

```
# Calculate the maximum depth across latitude bins in `water_depths`.
# MARGIN=1 means apply function by row (MARGIN=2, apply function by column).
```

```
# Note that depths are negative (i.e., m below the surface), so to find
# deepest depth we need the largest negative number (i.e., the 'min' value).
max_depth_lats <- apply(water_depths, MARGIN=1, FUN=function(x) min(x))
# Print the resulting vector of depths representing the maximum water depth
# in each latitude bin.
max_depth_lats

## [1] -24.46307 -29.84146 -23.16542 -18.64758
```

Note that `apply()` returns a vector of values corresponding to the smallest value in each row of the `water_depths` matrix.

Let's repeat the process, but this time we will determine the maximum mean water depth in each longitude bin (i.e., by column).

```
# Calculate the maximum depth across longitude bins in `water_depths` matrix.
# MARGIN=2 means apply function by column. Note that depths are negative
# (i.e., m below the surface), so to find deepest depth we need the largest
# negative number (i.e., the 'min' value).
max_depth_lons <- apply(water_depths, MARGIN=2, FUN=function(x) min(x))
# Print the resulting vector of depths representing the maximum water depth
# in each longitude bin.
max_depth_lons

## [1] -22.77234 -23.16542 -29.84146 -20.75842
```

The `sapply()` function is similar to `apply()` except that it is applied to lists and vectors. Since many functions can be vectorized, `sapply()` is most useful when applied to lists, or when functions applied to vectors return a list and we wish to extract a particular value from that list (see `strsplit()` example in the next section on string manipulations).

Here we will construct a list containing three vectors of numeric data. We will then use `sapply()` to calculate the mean of each vector in the list.

```
# Create a list object containing three numeric vectors.
nums_list <- list(v1=c(1:5), v2=c(6:10), v3=c(11:15))
# View the list.
nums_list

## $v1
## [1] 1 2 3 4 5
##
## $v2
## [1]  6  7  8  9 10
##
## $v3
## [1] 11 12 13 14 15
```

Next, we will use the `sapply()` function to return the mean of each element (vector) in the list. Note that functions already loaded into the active R environment (e.g., `mean()`) can be called by supplying the function name as a string to the `FUN` argument of `apply()` and `sapply()`.

```
# Calculate the mean of vector elements in list `nums_list`.
means <- sapply(nums_list, FUN='mean')
# View results.
means
```

```
## v1 v2 v3
##  3  8 13
```

Conditional Statements

Logic statements can be used to determine whether or not to perform certain operations. This is accomplished by incorporating logic statements into conditional statements. The basis of a conditional statement is that IF some condition is met (i.e., the logic statement that describes the condition evaluates to TRUE), THEN the operation is performed. Condition statements are most often used within functions and loops.

For example, let's return to our tagging data frame, ***bio_data***. Suppose we want to be more specific about the release location. Specifically, we want to change our dataset to reflect that fish listed as being released in L. Huron were actually released in Saginaw Bay, L. Huron. So that we do not overwrite our original data we will first make a copy of ***bio_data*** called ***bio_data_2***. As we will see later, this operation could be vectorized using the ***ifelse()*** function, but for the sake of demonstrating condition statements, we will perform the operation in a loop. The code used to make ***bio_data*** and ***bio_data_2*** is before the loop in the block below.

```
# View `bio_data`.
head(bio_data)

##    fish_id transmitter_code   tag_date length age release_loc
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron
## 2 2017_002   A69-9001-15124 2017-02-10   64.4  14    L. Huron
## 3 2017_003   A69-9001-15125 2017-02-10   53.5   9 L. Michigan
## 4 2017_004   A69-9001-15126 2017-02-10   63.8  14 L. Michigan
## 5 2017_005   A69-9001-15127 2017-02-10   64.2  10    L. Huron
## 6 2017_006   A69-9001-15128 2017-02-10   55.0  10 L. Michigan

# Make a copy of the bio_data data frame.
bio_data_2 <- bio_data

# Loop through the `release_loc` column of bio_data_2 and if the value is
# 'L. Huron', we will change it to 'Saginaw Bay, L. Huron'. Each time the
# code within the loop, the value of i will change until it equals the number
# of rows in `bio_data_2`.
for(i in 1:nrow(bio_data_2)){
 if(bio_data_2$release_loc[i] == 'L. Huron'){ # Compare the i-th value of
                         # release_loc column to
                         # 'L. Huron'
  bio_data_2$release_loc[i] <- 'Saginaw Bay, L. Huron'
 }
}
# View bio_data_2.
bio_data_2

##     fish_id transmitter_code   tag_date length age           release_loc
## 1  2017_001   A69-9001-15123 2017-02-10   53.7  12 Saginaw Bay, L. Huron
## 2  2017_002   A69-9001-15124 2017-02-10   64.4  14 Saginaw Bay, L. Huron
## 3  2017_003   A69-9001-15125 2017-02-10   53.5   9           L. Michigan
## 4  2017_004   A69-9001-15126 2017-02-10   63.8  14           L. Michigan
## 5  2017_005   A69-9001-15127 2017-02-10   64.2  10 Saginaw Bay, L. Huron
## 6  2017_006   A69-9001-15128 2017-02-10   55.0  10           L. Michigan
```

```
## 7  2017_007   A69-9001-15129 2017-02-10   66.4   7           L. Michigan
## 8  2017_008   A69-9001-15130 2017-02-10   64.0  14 Saginaw Bay, L. Huron
## 9  2017_009   A69-9001-15131 2017-02-10   56.2   8 Saginaw Bay, L. Huron
## 10 2017_010   A69-9001-15132 2017-02-10   65.1  14           L. Michigan
## 11 2017_011   A69-9001-15133 2017-02-14   59.6  10 Saginaw Bay, L. Huron
```

The above code only changes the value of ***release_loc*** if it was equal (note the double **==**) to 'L. Huron'. But what if we wanted to be more specific about the release location in 'L. Michigan' also? Let's change the value 'L. Michigan' to 'Saginaw Bay, L. Huron'.

If there are only two conditions, as there is in our data frame (i.e., fish were release in L. Huron or L. Michigan), we can use ***else***. In plain English, we are saying 'If the i-th value of ***release_loc*** is 'L. Huron', change it to 'Saginaw Bay, L. Huron'. Everything else will change to 'Green Bay, L. Michigan'.

```
# Re-make bio_data_2 data frame from bio_data.
bio_data_2 <- bio_data
# Loop through the `release_loc` column of bio_data_2 and if the value is
# 'L. Huron', we will change it to 'Saginaw Bay, L. Huron'. Each time the
# code within the loop, the value of i will change until it equals the number
# of rows in `bio_data_2`.
for(i in 1:nrow(bio_data_2)){
 if(bio_data_2$release_loc[i] == 'L. Huron'){ # Compare the i-thvalue of
                         # release_loc column to
                         # 'L. Huron'
  bio_data_2$release_loc[i] <- 'Saginaw Bay, L. Huron'
 }else{ # Every i-th value of `release_loc` that is not 'L. Huron' will
     # change to 'Green Bay, L. Michigan'
  bio_data_2$release_loc[i] <- 'Green Bay, L. Michigan'
 }
}
# View bio_data_2.
head(bio_data_2)

##    fish_id transmitter_code   tag_date length age           release_loc
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12  Saginaw Bay, L. Huron
## 2 2017_002   A69-9001-15124 2017-02-10   64.4  14  Saginaw Bay, L. Huron
## 3 2017_003   A69-9001-15125 2017-02-10   53.5   9 Green Bay, L. Michigan
## 4 2017_004   A69-9001-15126 2017-02-10   63.8  14 Green Bay, L. Michigan
## 5 2017_005   A69-9001-15127 2017-02-10   64.2  10  Saginaw Bay, L. Huron
## 6 2017_006   A69-9001-15128 2017-02-10   55.0  10 Green Bay, L. Michigan
```

Recognize that this only worked for us because we had only two options. If you have more than two options, you will need to add additional condition statements. To demonstrate how this is done, we will repeat the above operation, but this time we will specify a change from 'L. Michigan' to 'Green Bay, L. Michigan', rather than saying everything that isn't 'L. Huron' will become 'Green Bay, L. Michigan'.

```
# Re-make bio_data_2 data frame from bio_data.
bio_data_2 <- bio_data
# Loop through the `release_loc` column of bio_data_2 and if the value is
# 'L. Huron', we will change it to 'Saginaw Bay, L. Huron'. Each time the
# code within the loop, the value of i will change until it equals the number
# of rows in `bio_data_2`.
for(i in 1:nrow(bio_data_2)){
 if(bio_data_2$release_loc[i] == 'L. Huron'){ # Compare the i-th value of
                         # release_loc column to
                         # 'L. Huron'
```

```
  bio_data_2$release_loc[i] <- 'Saginaw Bay, L. Huron'
 }else if(bio_data_2$release_loc[i] == 'L. Michigan'){ # Every i-th value of
                                # `release_loc` that
                                # is not 'L. Huron'
                                # will change to
                                # 'Green Bay,
                                # L. Michigan'
  bio_data_2$release_loc[i] <- 'Green Bay, L. Michigan'
 }
}
# View bio_data_2.
bio_data_2
```

```
##      fish_id transmitter_code   tag_date length age            release_loc
## 1  2017_001   A69-9001-15123 2017-02-10   53.7  12  Saginaw Bay, L. Huron
## 2  2017_002   A69-9001-15124 2017-02-10   64.4  14  Saginaw Bay, L. Huron
## 3  2017_003   A69-9001-15125 2017-02-10   53.5   9 Green Bay, L. Michigan
## 4  2017_004   A69-9001-15126 2017-02-10   63.8  14 Green Bay, L. Michigan
## 5  2017_005   A69-9001-15127 2017-02-10   64.2  10  Saginaw Bay, L. Huron
## 6  2017_006   A69-9001-15128 2017-02-10   55.0  10 Green Bay, L. Michigan
## 7  2017_007   A69-9001-15129 2017-02-10   66.4   7 Green Bay, L. Michigan
## 8  2017_008   A69-9001-15130 2017-02-10   64.0  14  Saginaw Bay, L. Huron
## 9  2017_009   A69-9001-15131 2017-02-10   56.2   8  Saginaw Bay, L. Huron
## 10 2017_010   A69-9001-15132 2017-02-10   65.1  14 Green Bay, L. Michigan
## 11 2017_011   A69-9001-15133 2017-02-14   59.6  10  Saginaw Bay, L. Huron
```

The ifelse() Function

The ***ifelse()*** function allows us to apply conditional statements to an object (i.e., vectorization), returning an object with the same dimensions. This function is particularly useful for appending columns of data generated with conditional statements to data frames (e.g., assigning fish sex based on length), and for controlling plotting options (e.g., plot points for males and females a different color).

Let's return to our tagging data frame, ***bio_data***:

```
# View first 6 rows of `bio_data`
head(bio_data)
```

```
##    fish_id transmitter_code   tag_date length age release_loc
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron
## 2 2017_002   A69-9001-15124 2017-02-10   64.4  14    L. Huron
## 3 2017_003   A69-9001-15125 2017-02-10   53.5   9 L. Michigan
## 4 2017_004   A69-9001-15126 2017-02-10   63.8  14 L. Michigan
## 5 2017_005   A69-9001-15127 2017-02-10   64.2  10    L. Huron
## 6 2017_006   A69-9001-15128 2017-02-10   55.0  10 L. Michigan
```

Suppose our species is sexually dimorphic with males typically less than 60 cm and females greater than 60 cm. We can add a new column ***sex*** to ***bio_data*** by evaluating whether ***length*** is greater than 60 cm.

```
# Add conditional column, 'sex', indicating whether the fish is male or
# female based on 'length'.
bio_data$sex <- ifelse(bio_data$length > 60, 'female', 'male')
# View first 6 rows of `bio_data`.
head(bio_data)
```

```
##    fish_id transmitter_code   tag_date length age release_loc    sex
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron   male
## 2 2017_002   A69-9001-15124 2017-02-10   64.4  14    L. Huron female
## 3 2017_003   A69-9001-15125 2017-02-10   53.5   9 L. Michigan   male
## 4 2017_004   A69-9001-15126 2017-02-10   63.8  14 L. Michigan female
## 5 2017_005   A69-9001-15127 2017-02-10   64.2  10    L. Huron female
## 6 2017_006   A69-9001-15128 2017-02-10   55.0  10 L. Michigan   male
```

String Manipulation

The paste() Function

R provides a number of useful functions for manipulating strings. A string is a sequence of characters (letters, numbers, special characters, or a mix) that are enclosed by single or double quotes. String manipulations are often used to clean up character data that are not uniform within a vector (e.g., 'TTB-1' is not equal to 'TTB-001') and to create new labels for classifying observations. They are also often used to create file names for output files. The most commonly used function in R to manipulate strings is the ***paste()*** function. This function concatenates (combines) vectors of strings. We will demonstrate the ***paste()*** function using the ***bio_data*** data frame created earlier. The code that makes ***bio_data*** is below.

```
# View `bio_data`.
head(bio_data)

##    fish_id transmitter_code   tag_date length age release_loc    sex
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron   male
## 2 2017_002   A69-9001-15124 2017-02-10   64.4  14    L. Huron female
## 3 2017_003   A69-9001-15125 2017-02-10   53.5   9 L. Michigan   male
## 4 2017_004   A69-9001-15126 2017-02-10   63.8  14 L. Michigan female
## 5 2017_005   A69-9001-15127 2017-02-10   64.2  10    L. Huron female
## 6 2017_006   A69-9001-15128 2017-02-10   55.0  10 L. Michigan   male
```

Suppose transmitters were reused among lake, so we want to create a new column, ***new_id***, in ***bio_data*** that combines ***release_loc*** and ***transmitter_code*** to create a unique identifier for each fish. We can use the ***paste()*** function to concatenate these two character strings.

```
# Create a new column in `bio_data` which is the concatenation of
# bio_data$release_loc and bio_data$animal_id.
bio_data$new_id <- paste(bio_data$release_loc, bio_data$transmitter_code)
# View `bio_data`.
head(bio_data)

##    fish_id transmitter_code   tag_date length age release_loc    sex
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron   male
## 2 2017_002   A69-9001-15124 2017-02-10   64.4  14    L. Huron female
## 3 2017_003   A69-9001-15125 2017-02-10   53.5   9 L. Michigan   male
## 4 2017_004   A69-9001-15126 2017-02-10   63.8  14 L. Michigan female
## 5 2017_005   A69-9001-15127 2017-02-10   64.2  10    L. Huron female
## 6 2017_006   A69-9001-15128 2017-02-10   55.0  10 L. Michigan   male
##                       new_id
## 1    L. Huron A69-9001-15123
## 2    L. Huron A69-9001-15124
## 3 L. Michigan A69-9001-15125
## 4 L. Michigan A69-9001-15126
```

```
## 5    L. Huron A69-9001-15127
## 6 L. Michigan A69-9001-15128
```

The ***release_loc*** and ***transmitter_code*** columns were combined and added to a new column in ***bio_data*** named ***new_id***. Notice that columns were combined using a space character (default). ***paste()*** allows strings to be combined using any sort of separator by specifying the optional ***sep*** argument. For example, if we wish to combine ***release_loc*** and ***transmitter_code*** columns with an underscore, we would do the following.

```
# Concatenate bio_data$release_loc and bio_data$animal_id, separated by an
# underscore character.
bio_data$new_id <- paste(bio_data$release_loc,
                         bio_data$transmitter_code,
                         sep='_')
# View `bio_data`.
head(bio_data)

##    fish_id transmitter_code   tag_date length age release_loc    sex
## 1 2017_001   A69-9001-15123 2017-02-10   53.7  12    L. Huron   male
## 2 2017_002   A69-9001-15124 2017-02-10   64.4  14    L. Huron female
## 3 2017_003   A69-9001-15125 2017-02-10   53.5   9 L. Michigan   male
## 4 2017_004   A69-9001-15126 2017-02-10   63.8  14 L. Michigan female
## 5 2017_005   A69-9001-15127 2017-02-10   64.2  10    L. Huron female
## 6 2017_006   A69-9001-15128 2017-02-10   55.0  10 L. Michigan   male
##                       new_id
## 1    L. Huron_A69-9001-15123
## 2    L. Huron_A69-9001-15124
## 3 L. Michigan_A69-9001-15125
## 4 L. Michigan_A69-9001-15126
## 5    L. Huron_A69-9001-15127
## 6 L. Michigan_A69-9001-15128
```

A practical application of ***paste()*** for telemetry data is combining date and time data when they are entered into different columns. You can use the paste function to combine them into single timestamp.

```
# Create a vector of time and date data.
date_time_data <- data.frame(date=c('2017-10-04', '2017-11-12', '2017-12-01'),
                  time=c('04:13:07', '12:46:14', '21:01:11'))
# Use `paste()` to create a new column, 'timestamp', containing both data and
# time. Note that we do not need to provide a `sep` character because we wish
# the strings to be separated by a space, which is the default value for
# `sep`
date_time_data$timestamp <- paste(date_time_data$date, date_time_data$time)
# View `date_time_data.
date_time_data

##         date     time           timestamp
## 1 2017-10-04 04:13:07 2017-10-04 04:13:07
## 2 2017-11-12 12:46:14 2017-11-12 12:46:14
## 3 2017-12-01 21:01:11 2017-12-01 21:01:11
```

Note: The ***paste0()*** function is a special case of ***paste()*** that combines string vectors with no separator (i.e., sep='').

The strsplit() Function

The `strsplit()` function splits a character vector into substrings based on user-defined patterns. The `strsplit()` function can be thought of as the opposite of `paste()`. For example, suppose we want to extract the tag id from the `transmitter_code` column.

```
# Split `bio_data$transmitter_code` on the '-' character.
x_split <- strsplit(bio_data$transmitter_code, '-')
# View the resulting object.
head(x_split)

## [[1]]
## [1] "A69"   "9001"  "15123"
##
## [[2]]
## [1] "A69"   "9001"  "15124"
##
## [[3]]
## [1] "A69"   "9001"  "15125"
##
## [[4]]
## [1] "A69"   "9001"  "15126"
##
## [[5]]
## [1] "A69"   "9001"  "15127"
##
## [[6]]
## [1] "A69"   "9001"  "15128"

# Determine the class of the resulting object.
class(x_split)

## [1] "list"
```

Note that `strsplit()` returns a list containing a three-element vector for each row of the data frame. If we wish to return a vector of only the third element of the string split we have two main options: 1) We could loop through the list elements, extract the 3rd vector element of each list element, and append them one by one to a vector using `c()`, 2) We could use the `sapply()` function (see above description of `sapply()` in the 'Repeated Operations' section). The `sapply()` function is a wrapper for doing essentially exactly what we would do in option 1, it is just a little more elegant because it requires fewer lines of code on our part. In the following code, we will apply the `sapply()` function to the problem. The function has two comma-separated arguments. The first is the name of our list object, `x_split`. The second is a custom function, `function(x){x[[3]]}`, which has a single argument, `x` (a list object) and will perform a double square bracket subset operation, extracting the 3rd element of list `x`. `sapply()` will return the values as a vector.

```
# Use `sapply()` to apply a double square bracket list subset to the
# individual list elements in `x_split`.
sapply(x_split, function(x){x[[3]]})

##  [1] "15123" "15124" "15125" "15126" "15127" "15128" "15129" "15130"
##  [9] "15131" "15132" "15133"
```

The substr() Function

The `substr()` function is used to extract or replace substrings in a character vector. Substrings are selected based on their position in the string. It can be used in place of `strsplit()` in cases where the number of characters and position of the substring to be extracted or replaced are known.

For example, let's return to our example of extracting tag id from the `transmitter_code` column of `bio_data`. Because the format of the `transmitter_code` character string is consistent across fish, we could use `substr()` instead of `strsplit()` to extract the tag ids.

```
substr(bio_data$transmitter_code, start=10, stop=14)

##  [1] "15123" "15124" "15125" "15126" "15127" "15128" "15129" "15130"
##  [9] "15131" "15132" "15133"
```

The above code was more straight forward than using `strsplit()`, which required us to also use `sapply()` to extract the tag ids from the vectors of transmitter codes. However, this only worked because the number and position of characters we wanted to extract was the same for all rows in the data frame. If, for example, tag id codes differed in length among fish, we could not have used this method.

Replacing a substring using `substr()` is similar to extracting a substring, except we include an assignment operator to assign a new set of characters to that location in the character string (conceptually similar to how we changed the column headers in a data frame using the `names()` function). Suppose we wish to replace 'A69' in our `transmitter_code` strings with the character string '69k'. We will first assign `bio_data$transmitter_code` to a new vector named `test_string` so that we do not overwrite the data in `bio_data`, which we will use again in subsequent sections of this guide.

```
# Assign `bio_data$transmitter_id` to a new vector named `test_string`.
test_string <- bio_data$transmitter_code

# Replace 'A69' in the characters strings with '69kHz'.
substr(test_string, start=1, stop=3) <- '69k'
test_string

##  [1] "69k-9001-15123" "69k-9001-15124" "69k-9001-15125" "69k-9001-15126"
##  [5] "69k-9001-15127" "69k-9001-15128" "69k-9001-15129" "69k-9001-15130"
##  [9] "69k-9001-15131" "69k-9001-15132" "69k-9001-15133"
```

The gsub() Function

Another way to substitute one character or character string for another is to use the `gsub()` function. The `gsub()` function matches a user defined pattern and replaces each occurrence of the pattern in a character vector with another string. In the next example, we will use `gsub()` to change the format of the `tag_date` column of `bio_data`. We will replace the character '-' that separates each field in the date to a '/' character.

```
# Display the original tag_date format.
bio_data$tag_date

##  [1] "2017-02-10" "2017-02-10" "2017-02-10" "2017-02-10" "2017-02-10"
##  [6] "2017-02-10" "2017-02-10" "2017-02-10" "2017-02-10" "2017-02-10"
## [11] "2017-02-14"

# Match and replace '-' in `bio_data$tag_date` with '/'.
bio_data$tag_date <- gsub('-', '/', bio_data$tag_date)
```

```
# Display new tag_date format.
bio_data$tag_date

##  [1] "2017/02/10" "2017/02/10" "2017/02/10" "2017/02/10" "2017/02/10"
##  [6] "2017/02/10" "2017/02/10" "2017/02/10" "2017/02/10" "2017/02/10"
## [11] "2017/02/14"
```

This function is probably most commonly used for replacing special characters (e.g., '-', '_','/','#') and for removing unwanted spaces within a character string. In the next example, we will remove the space in ***bio_data$release_loc*** by replacing the space (' ') with an empty character string ('').

```
# View original `bio_data$release_loc`.
bio_data$release_loc

##  [1] "L. Huron"    "L. Huron"    "L. Michigan" "L. Michigan" "L. Huron"
##  [6] "L. Michigan" "L. Michigan" "L. Huron"    "L. Huron"    "L. Michigan"
## [11] "L. Huron"

# Remove the space (' ') from elements in `bio_data$release_loc`.
gsub(' ', '', bio_data$release_loc)

##  [1] "L.Huron"    "L.Huron"    "L.Michigan" "L.Michigan" "L.Huron"
##  [6] "L.Michigan" "L.Michigan" "L.Huron"    "L.Huron"    "L.Michigan"
## [11] "L.Huron"
```

More functions for manipulating strings can be found by searching for ***?grep***.

Plotting Data

Plots are an important tool for visualizing data and provide an efficient way to communicate results. Many R users use the ***ggplot2*** package to make figures. We encourage you to explore this package if you are interested, but we will focus on base plotting functions (i.e., the base ***graphics*** package) that come with the standard R download in this manual.

Scatter and Line Plots: The plot() Function

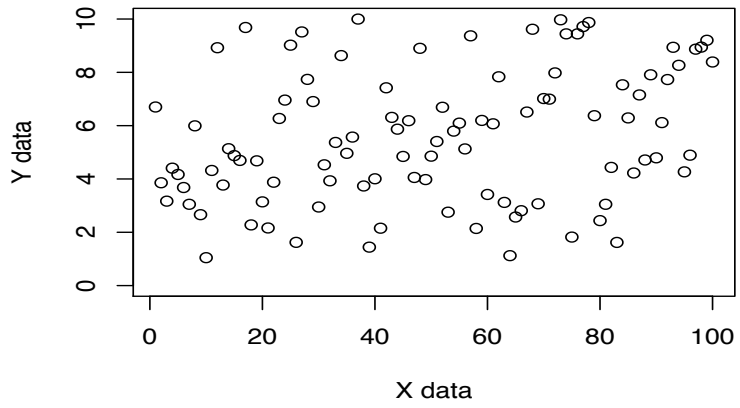
We will start by plotting some simple data so that we can gain experience modifying some of the base plot features.

First, we will create some random data to plot. We will use the ***set.seed()*** function to ensure that our random number generation is reproducible. See ***?set.seed*** for more information on this function. The ***runif()*** function returns values from a uniform distribution. If we had wanted to draw random numbers from a normal or binomial distribution, we could have used ***rnorm()*** and ***rbinom()***, respectively.

```
# Set seed so that random number generation is reproducible.
set.seed(45)
# Create two vectors, `x` and `y`, each of length 100.
x_data <- 1:100
y_data <- runif(100, 1, 10) # Return 100 values between 1 and 10.
```

Next, we will plot the data as a scatter plot. In the call to ***plot()***, the ***x*** argument refers to data to be plotted on the x-axis, and the ***y*** argument refers to data to be plotted on the y-axis. The ***xlab*** and ***ylab*** arguments provide the axes titles for the x and y axes, respectively. The ***ylim*** argument specifies the extent of values that will be displayed on the y-axis.

```
# Create a scatter plot of our `x` and `y` vectors.
plot(x_data, y_data, xlab='X data', ylab='Y data', ylim=c(0, 10))
```

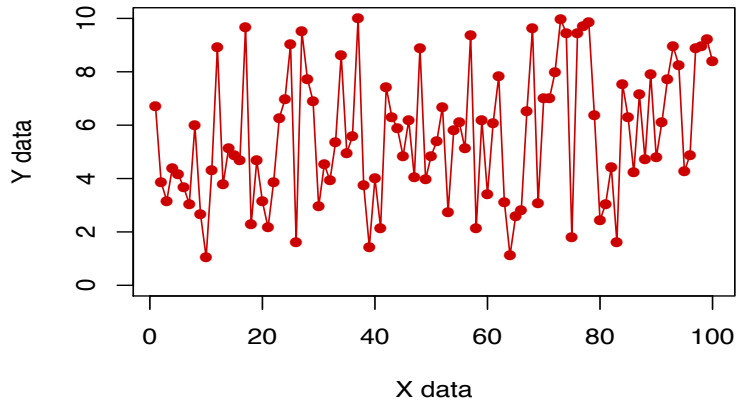


In R, we have an extensive ability to control the appearance of the plot by adding optional arguments to the `plot()` call. A good description of the different graphical parameters that can be used to modify the plot appearance can be found at http://www.statmethods.net/advgraphs/parameters.html.

Let's make some aesthetic modifications to the plot:

1. Add lines connecting the points (`type`), change to a different point style (`pch`), and make the plot red (`col`). Note that a chart of colors that can be called by name in R is supplied in Appendix D. The list of names can also be viewed by running the command `colors()` in the R Console.

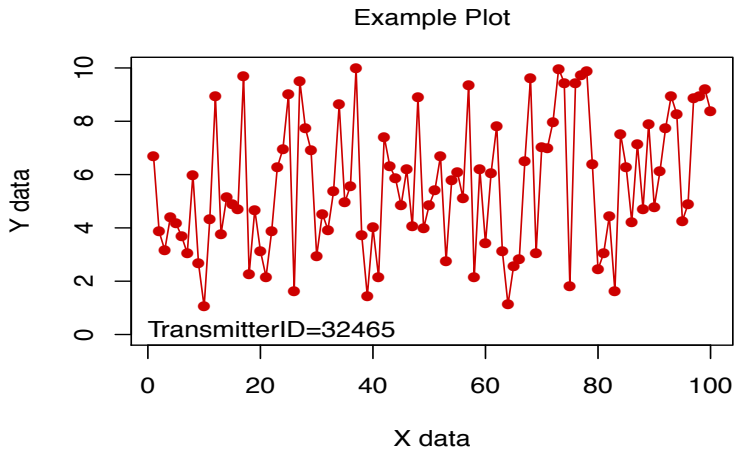
```
# Replot `x_data` and `y_data`, this time adding lines between the points,
# changing the point style, and making the plot red.
plot(x_data, y_data, xlab='X data', ylab='Y data', ylim=c(0, 10), type='o',
   pch=16, col='red3')
```



2. Add text to the top of the plot and within the plot using the `mtext()` and `text()` functions. The `mtext()` function adds text to the margin of a plot (i.e., outside the plot). In the call to `mtext()`, the `side` argument indicates on which side of the plot to add the text (labeled 1 to 4, counterclockwise from the x-axis); the `line` argument specifies the number of lines to move the text off the axis. Note that a title can also be added to the top of a figure by providing a `main` argument within the call to

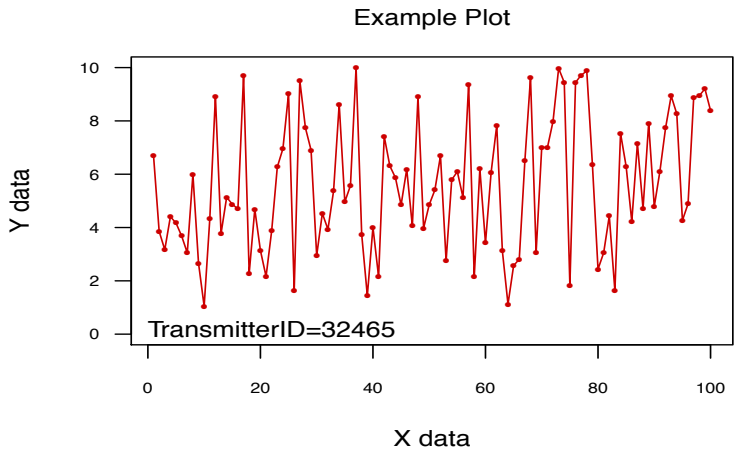
`plot()` itself, but you will have less freedom to format it. The `text()` function is used to place text within the plot. The text is placed at the coordinates specified by the `x` and `y` arguments. The `adj` argument in the call to `text()` specifies the justification of the text (0 = left justified).

```
# Replot the data.
plot(x_data, y_data, xlab='X data', ylab='Y data', ylim=c(0, 10), type='o',
   pch=16, col='red3')
# Use `mtext()` to add a title at the top of the plot.
mtext(side=3, line=1, 'Example Plot')
# Add text within the plot.
text(x=0, y=0.2, 'TransmitterID=32465', adj=0)
```



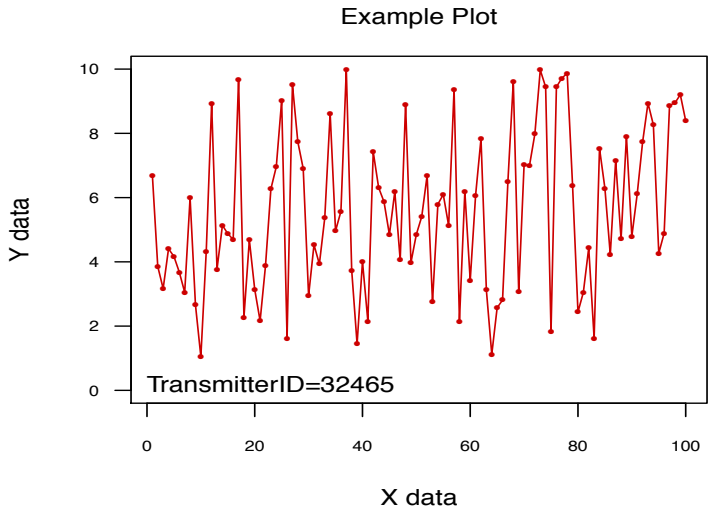
3. Increase the size of the points (`cex`), decrease the size of the axes labels (`cex.axis`), and change the orientation of the y-axis labels (`las`) in the `plot()` function. Default for both `cex` and `cex.axis` is 1; therefore, smaller < 1, larger >1.

```
# Replot the data, changing the size of the points (`cex`), size of the axes
# labels (`cex.axis`), and orientation of the y-axis labels (`las`).
plot(x_data, y_data, xlab='X data', ylab='Y data', ylim=c(0, 10), type='o',
   pch=16, col='red3', cex=0.5, cex.axis=0.7, las=1)
# Use `mtext()` to add a title at the top of the plot.
mtext(side=3, line=1, 'Example Plot')
# Add text within the plot.
text(0, 0.2, 'TransmitterID=32465', adj=0)
```



4. Change the margin size. This is accomplished with a call to ***par()***, which is used to set the global graphing parameters.

```
# Change the size of the margins. Margin widths are provided as a vector
# c(bottom, left, top, right).
par(mar=c(5, 5, 2, 2))
# Replot the data.
plot(x_data, y_data, xlab='X data', ylab='Y data', ylim=c(0, 10), type='o',
   pch=16, col='red3', cex=0.5, cex.axis=0.7, las=1)
# Use `mtext()` to add a title at the top of the plot.
mtext(side=3, line=1, 'Example Plot')
# Add text within the plot.
text(0, 0.2, 'TransmitterID=32465', adj=0)
```

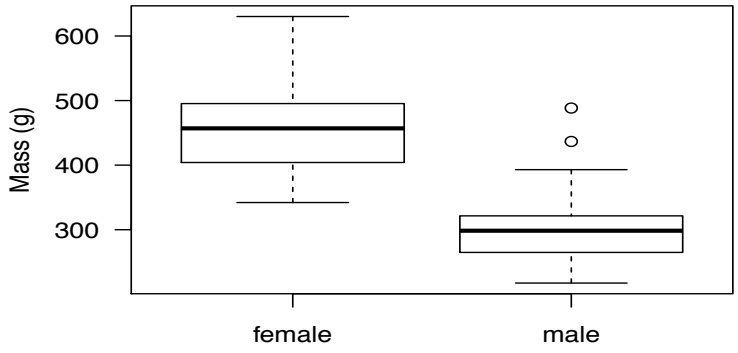


Box Plots: The boxplot() Function

Box plots provide a standardized way of displaying the distribution of data. From bottom to top, the lines represent the minimum, first quartile, median, third quartile, and maximum. Box plots are created using the ***boxplot()*** function. Outliers are displayed as individual points by default, but the plot can be changed so that the whiskers of the plot encompass the entire range of the dataset by setting ***range=0*** in the ***boxplot()*** function. In the following code, we will create a data frame for weights for 100 male and female lake trout.

The data frame will contain an ***id*** column, a ***sex*** column, and a ***mass*** column. We will then use the ***boxplot()*** function to create a box plot comparing the weights of males and females. Note that the input data are entered differently in this example than in the previous example using ***plot()***. This time we enter the input data as a formula of the form, y~x, which translates to 'y as a function of x'. In this case, we are plotting mass as a function of sex.

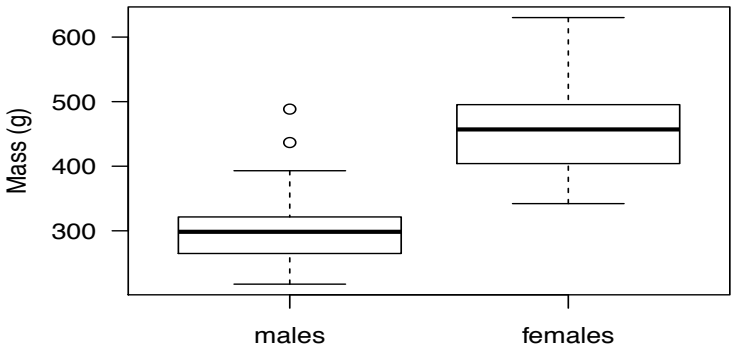
```
# Set seed so that random number generation is reproducible.
set.seed(45)
# Create a data frame of weight data for male and female fish.
weight_data <- data.frame(id=1:100,
             sex=c(rep('male', 50), rep('female', 50)),
             mass=c(rnorm(50, 300, 50), rnorm(50, 450, 50)),
             stringsAsFactors=FALSE)
# View the first 20 rows of `weight_data`.
head(weight_data)

##   id  sex     mass
## 1  1 male 317.0400
## 2  2 male 264.8330
## 3  3 male 281.0231
## 4  4 male 262.6976
## 5  5 male 255.0946
## 6  6 male 283.2603

# Make a box plot that compares the weights of males and females.
boxplot(weight_data$mass ~ weight_data$sex, ylab='Mass (g)', las=1)
```



By default, R plots the categorical labels in alphabetical order. If you want to change the order of plotting along the x-axis (e.g., plot males and then females) you can add a column to the data frame containing a numeric value indicating the order of plotting (e.g., rows with sex='male'are assigned a value of 1, rows with sex='female' are assigned a value of 2; can be created using the ***ifelse()*** function described above). You would then use the new numeric column as the ***x*** argument in the ***boxplot()*** call and provide the x-axis labels using the ***names*** argument in the ***boxplot()*** call.

```
# Add a new column indicating plot order along the x-axis.
weight_data$plot_order <- ifelse(weight_data$sex == 'male', 1, 2)
# Plot the box plot in reverse order, with males appearing first on the
# x-axis.
```

```
boxplot(weight_data$mass ~ weight_data$plot_order, ylab='Mass (g)',
    las=1, names=c('males', 'females'))
```



Bar Plots: The barplot() Function

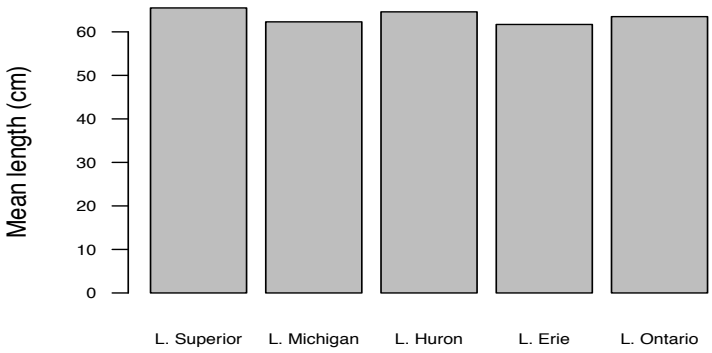
Bar plots are typically used to graph numeric data with categorical independent variables. Typically, bar plots display summary data like means, or counts.

Suppose we want to compare visually the mean length of adult walleye captured across the five Great Lakes. We have already calculated the means and want to plot them using the ***barplot()*** function. First, we will create a data frame with our summary data. Then we will plot the data using the ***barplot()*** function.

```
# Create a data frame containing mean length data for walleye across the five
# Great Lakes.
walleye_summary <- data.frame(lake=c('L. Superior', 'L. Michigan',
                    'L. Huron', 'L. Erie', 'L. Ontario'),
                mean_length=c(65.5, 62.3, 64.6, 61.7, 63.5),
                stringsAsFactors=FALSE)
# View the `walleye_summary` data frame.
walleye_summary

##          lake mean_length
## 1 L. Superior        65.5
## 2 L. Michigan        62.3
## 3    L. Huron        64.6
## 4     L. Erie        61.7
## 5  L. Ontario        63.5
```

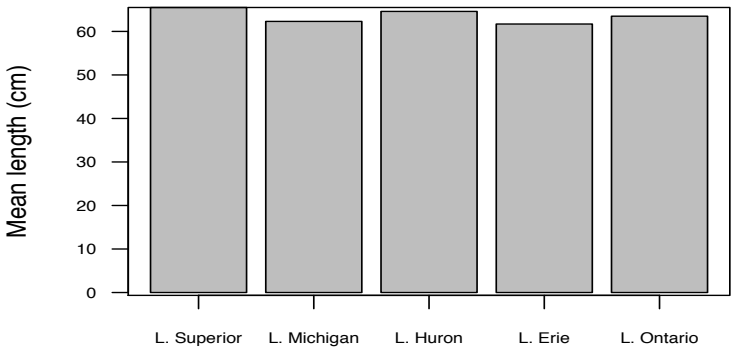
Now that we have the summary data, we can create the bar plot using the ***barplot()*** function.

```
barplot(walleye_summary$mean_length, names.arg=walleye_summary$lake,
    cex.axis=0.7, cex.names=0.7, las=1, ylab='Mean length (cm)')
```



The default settings for `barplot()` produce a plot with no x-axis or box around the plot. To make this plot more suitable for presentations and publication we need to add the x-axis (using the `axis.lty` argument) and a box around the plot (using the `box()` function).

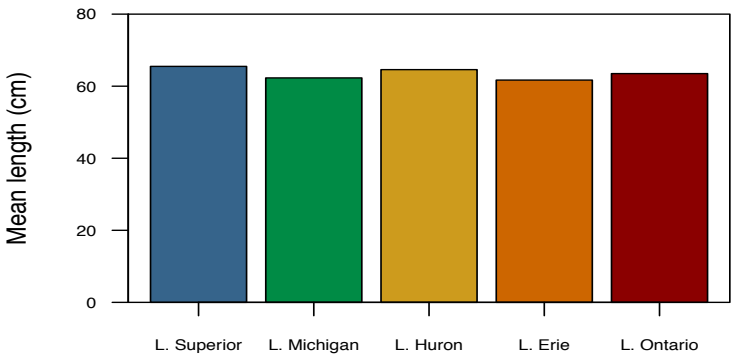
```
# Replot the bar plot, adding the x-axis to the plot using the `axis.lty`
# argument.
barplot(walleye_summary$mean_length, names.arg=walleye_summary$lake,
    axis.lty=1, cex.names=0.7, cex.axis=0.7, las=1,
    ylab='Mean length (cm)')
# Add a box around the plot using the `box()` function.
box(lwd=1)
```



We now have a box around the plot, but we need to expand the y axis (`ylim`) so that there is space between the bars and the top edge of the bordering box. We'd also like the bar for each lake to be represented by a different color. We can do this by supplying a vector of colors to the `col` argument of `barplot()`. Note that if the length of the supplied color vector (`col`) is less than the number of bars in the plot, the color vector is recycled. Also, note that by setting the minimum `ylim` value to 0, we get rid of the blank space between the bars and the x-axis.

```
barplot(walleye_summary$mean_length, names.arg=walleye_summary$lake,
    ylim=c(0, 80), axis.lty=1, cex.names=0.7, cex.axis=0.7, las=1,
    ylab='Mean length (cm)', col=c('steelblue4', 'springgreen4',
```

```
                    'goldenrod3', 'darkorange3',
                    'darkred'))
box(lwd=1)
```



Our plot is looking pretty good now, but it's missing one important feature, error bars. Surprisingly, R doesn't have a built-in function for plotting error bars but can be added to graphs in a few different ways using various workarounds. One way that we have found works well is to use the ***arrows()*** function. The ***arrows()*** function adds arrows to a plot by specifying a start (***x0*** and ***y0***) and end point (***x1*** and ***x2***) for the arrow. The angle of the arrow head can be set so that it is a flat line using the ***angle*** argument.

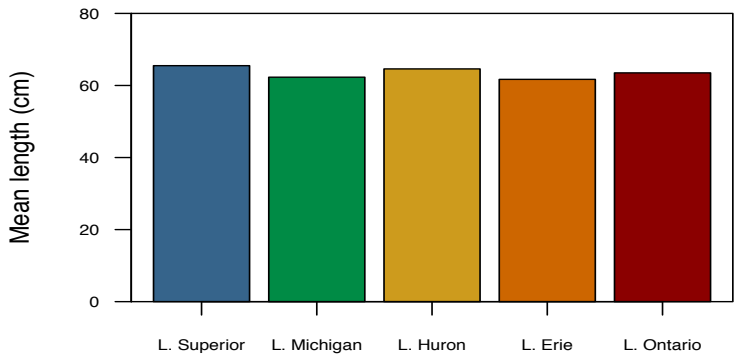
Let's add error bars to our plot. First, we will add a vector of error data to our data frame.

```
# Add an error column to `walleye_summary`.
walleye_summary$error <- c(10.4, 8.5, 11.0, 9.3, 7.0)
# View `walleye_summary`.
walleye_summary

##          lake mean_length error
## 1 L. Superior        65.5  10.4
## 2 L. Michigan        62.3   8.5
## 3    L. Huron        64.6  11.0
## 4     L. Erie        61.7   9.3
## 5  L. Ontario        63.5   7.0
```

We will now add the error bars using the ***arrows()*** function. One difficulty we face in doing this is that ***barplot()*** plots the bars at locations along the x-axis that are not obvious by looking at the plot, so we cannot simply say that the bars are located at x=1,2,3,4,5. We can get the plotting location for the plot by assigning the plot to an object. The object stores the location of each bar on the x-axis in a matrix. In the next section of code, we will re-plot the data, this time also assigning the plot to an object names ***plot_locs***.

```
# Replot the `walleye_summary` data, this time also assigning the plot to an
# object named `plot_locs`.
plot_locs <- barplot(walleye_summary$mean_length,
           names.arg=walleye_summary$lake,
           ylim=c(0, 80),
           axis.lty=1,
           cex.names=0.7,
           cex.axis=0.7,
           las=1,
```

```
            ylab='Mean length (cm)',
            col=c('steelblue4','springgreen4','goldenrod3',
               'darkorange3','darkred'))
box(lwd=1)
```



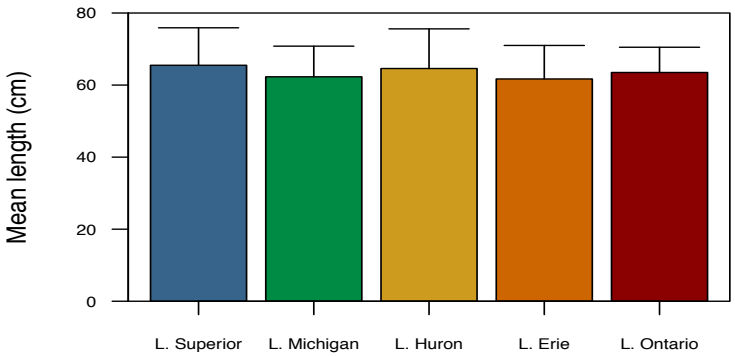

```
# View `plot_locs`.
plot_locs
```

```
##      [,1]
## [1,]  0.7
## [2,]  1.9
## [3,]  3.1
## [4,]  4.3
## [5,]  5.5
```

We now have a one-column matrix indicating the center location of each bar on the x-axis. We can use this matrix and square bracket subsetting to properly locate our arrows (i.e., error bars). Note that when subsetting `plot_locs` we must use two-dimensional square bracket subsetting notation. This is because `plot_locs` is technically a matrix, even though it only has a single column. Our arrows start at the mean values that determined the heights of the bars, and they end at the mean values plus the error values. The x values for each arrow do not change from start to end. We only added upper error bars to this plot, but lower error bars could be added the exact same way, only with the arrow end points being the means minus the errors. To make the arrow head flat, we set `angle=90`.

```
# Replot the mean data.
barplot(walleye_summary$mean_length,
    names.arg=walleye_summary$lake,
    ylim=c(0, 80),
    axis.lty=1,
    cex.names=0.7,
    cex.axis=0.7,
    las=1,
    ylab='Mean length (cm)',
    col=c('steelblue4', 'springgreen4', 'goldenrod3', 'darkorange3',
       'darkred'))
box(lwd=1)
# Plot error bars, x0 and x1 are the locations stored in `plot_locs`. y0 is
# the mean values (`walleye_summary$mean_length`), and y1 is the mean length
# of the fish plus the error (`walleye_summary$mean_length` +
```

```
# `walleye_summary$error`). Angle=90 makes the arrow head flat (90 degrees
# from the shaft of the arrow).
arrows(x0=plot_locs[,1],
    y0=walleye_summary$mean_length,
    x1=plot_locs[,1],
    y1=walleye_summary$mean_length + walleye_summary$error,
    angle=90)
```



Grouped Bar plots

Often, we will want to create grouped bar plots. For example, we may want to plot the mean length of males and females separately across lakes. To accomplish this, we will first make a new data frame containing the mean length of males and females in each lake.

```
# Create a data frame containing the mean length of male and female walleye
# across the five Great Lakes.
walleye_summary_2 <- data.frame(lake=c('Lake Superior', 'Lake Michigan',
                     'Lake Huron', 'Lake Erie',
                     'Lake Ontario', 'Lake Superior',
                     'Lake Michigan', 'Lake Huron',
                     'Lake Erie', 'Lake Ontario'),
                  sex=c(rep('Males', 5), rep('Females', 5)),
                  mean_length=c(61.5, 59.3, 62.6, 58.7, 60.5,
                         67.5, 64.3, 68.6, 63.7, 67.5),
                  stringsAsFactors=FALSE)
# View the `walleye_summary_2` data frame.
walleye_summary_2

##             lake     sex mean_length
## 1  Lake Superior   Males        61.5
## 2  Lake Michigan   Males        59.3
## 3     Lake Huron   Males        62.6
## 4      Lake Erie   Males        58.7
## 5   Lake Ontario   Males        60.5
## 6  Lake Superior Females        67.5
## 7  Lake Michigan Females        64.3
## 8     Lake Huron Females        68.6
```

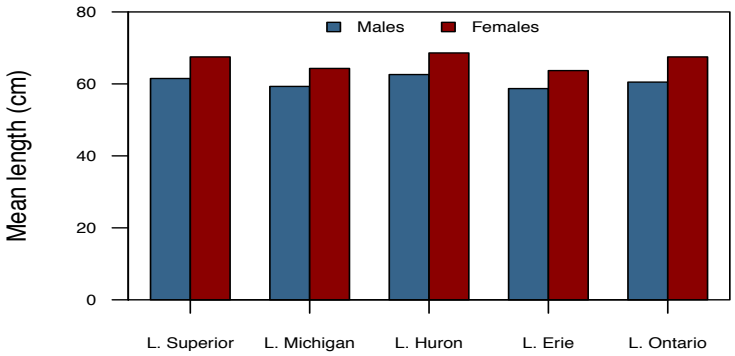
```
## 9      Lake Erie Females        63.7
## 10  Lake Ontario Females        67.5
```

To plot grouped bar graphs, we need to supply a matrix as the ***height*** argument to the ***barplot()*** function. In our example, the columns in our matrix correspond to the five lakes, and the rows correspond to the sexes. In other words, columns are the between comparisons and rows are the within comparisons.

```
# Convert mean male and female walleye lengths to a matrix for passing to
# barplot().
lengths <- matrix(walleye_summary_2$mean_length, nrow=2, byrow=TRUE)
# View the `lengths` matrix.
lengths

##      [,1] [,2] [,3] [,4] [,5]
## [1,] 61.5 59.3 62.6 58.7 60.5
## [2,] 67.5 64.3 68.6 63.7 67.5
```

Well now pass our matrix to ***barplot()*** to create the grouped bar plot. The ***legend()*** function allows us to add a legend to the figure. Consult the help page for ***legend()*** to view the plotting options.

```
# Plot the grouped bar plot.
group_plot <- barplot(height=lengths,
           names.arg=walleye_summary$lake,
           ylim=c(0, 80),
           axis.lty=1,
           cex.names=0.7,
           cex.axis=0.7,
           las=1,
           ylab='Mean length (cm)',
           col=c('steelblue4', 'darkred'),
           beside=TRUE)
box(lwd=1)
# Add a legend to the plot.
legend(mean(group_plot[,3]), 81,
    legend=c('Males', 'Females'),
    fill=c('steelblue4', 'darkred'),
    horiz=TRUE,
    cex=0.7,
    bty='n',
    xjust=0.5)
```



In this example, we displayed the bars adjacent to one another to compare males and females. However, you may wish to plot stacked bars instead (for example, if you wanted to plot species composition for each lake). We can plot stacked bar plots by running `barplot()` function with the argument `beside=FALSE`.

Multi-Panel Plots: par(mfrow) and layout() Function

It is relatively easy to make multi-panel plots in R. There are several ways to do this using base R functions. The various ways to build multi-panel plots in R are discussed in a document by Sean Anderson, available at http://seananderson.ca/courses/11-multipanel/multipanel.pdf. We will present two different methods here: `par(mfrow)`, and the `layout()` function.

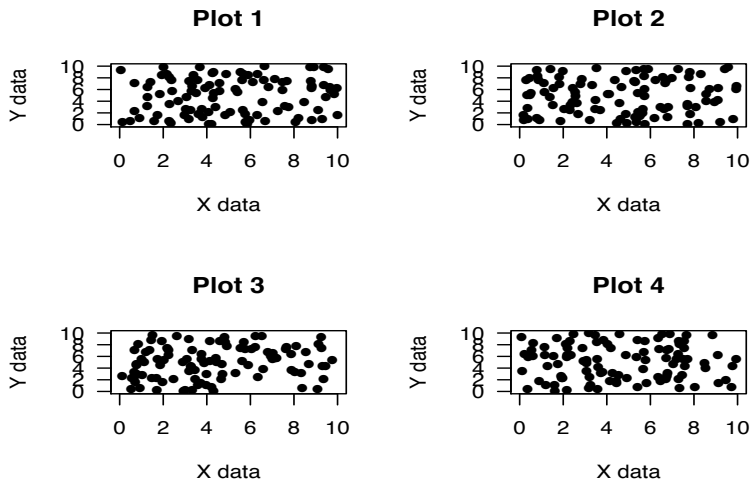
Using par(mfrow)

For basic grid layouts, you can use `par(mfrow=)`. `par(mfrow)` accepts a two-element vector indicating the number of rows and columns, respectively, in a multi-panel grid layout. Individual plots are added to the multi-panel plot in the order they are called. With `par(mfrow)`, plots are added to the grid row by row. If you wish to add plots column by column, you can use `par(mfcol)`.

In the following example, we will plot four sets of randomly-generated data in a single multi-panel plot. We will use a `for` loop to do this, as this is a common way to create multi-panel plots. We will use the index variable, `i`, during each iteration of the `for` loop to sequentially label our plots 1 through 4 using the `paste()` function. Again, this is something you will likely do fairly frequently.

```
# Set seed so that random number generation is reproducible.
set.seed(45)

# Set mfrow=c(2, 2). This will create a multi-panel plot with 2 equally-sized
# rows and 2 equally-sized columns.
par(mfrow=c(2,2))
# Use a `for` loop to build a four-panel plot.
for(i in 1:4){
 plot(runif(100, 0, 10),
   runif(100, 0, 10),
   xlim=c(0, 10),
   ylim=c(0, 10),
   ylab='Y data',
   xlab='X data',
```

```
   main=paste('Plot',i),
   pch=16,
   las=1)
}
```

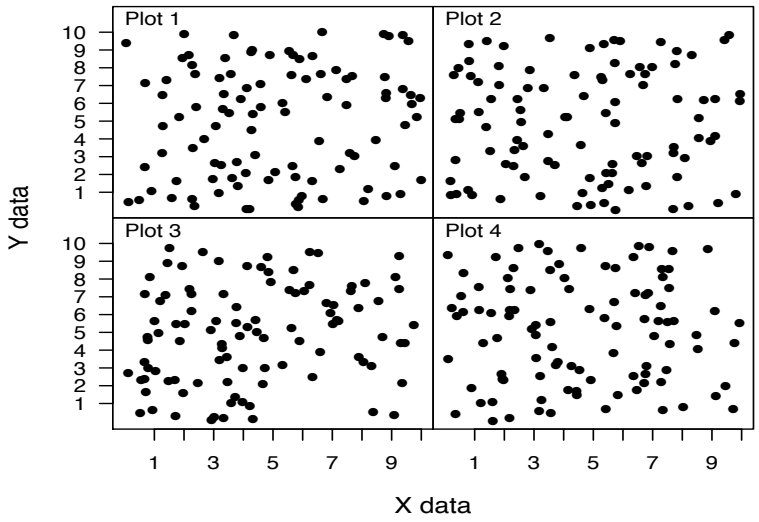


Notice that our plots all have the same axes. We can reduce redundancy and maximize the viewable space of the plots by adjusting the margin sizes. The commands `par(mar)` and `par(oma)` are used to specify the size of inner and outer margin, respectively. To make a 'prettier' plot, we will eliminate the space between plots (`par(mar)`), suppress the x and y axes (`xaxt='n'` and `yaxt='n'`) and add custom axes (`axis()`) and axes titles (`mtext()`). Lastly, we will add plot labels to the top left hand corner of each plot, rather than on top.

There is a lot going on in this code. Because we don't want x and y-axes on every plot, we have added conditional statements (i.e., `if`) so that we add custom x axes only when plotting plots 3 and 4, and add custom y-axes only when plotting plots 1 and 3. We use the `text()` function to add a label in the top right-hand corner of every plot within the `for` loop, but wait until after the loop finishes to add our custom axes titles to the x and y margin using the `mtext()` function.

```
# Set seed so that random number generation is reproducible.
set.seed(45)

# Set mfrow=c(2, 2), make inner margins=c(0, 0, 0, 0) and outer
# margins=c(4, 5, 1, 1).
par(mfrow=c(2,2), mar=c(0, 0, 0, 0), oma=c(4, 5, 1, 1))
# Use a `for` loop to build a four-panel plot.
for(i in 1:4){
 plot(runif(100, 0, 10),
    runif(100, 0, 10),
    xlim=c(0, 10),
    ylim=c(0, 11),
    ylab='',
    xlab='',
    xaxt='n',
    yaxt='n',
    pch=16)
 # Add custom y-axes to plots 1 and 3.
 if(i %in% c(1, 3)){
  axis(2, at=1:10, labels=1:10, las=1)
```

```
 }
 # Add custom x-axes to plots 3 and 4.
 if(i %in% c(3, 4)){
  axis(1, at=1:10, labels=1:10, las=1)
 }
 # Add plot labels to the four individual plots.
 text(0, 10.75, paste('Plot', i), adj=0)
}
# Add custom x and y axis title to the `outer` margin of the plot. This is
# done outside the loop because we want a single x and y axis title for each
# axis of the multi-panel plot.
mtext('Y data', side=2, line=3, outer=TRUE)
mtext('X data', side=1, line=3, outer=TRUE)
```



Using layout()

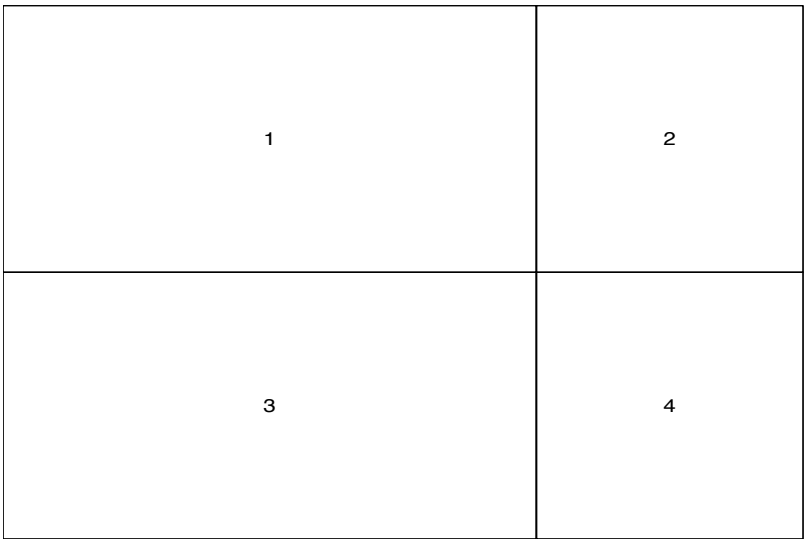
Using `par(mfrow)` we created a grid layout, with each plot occupying an equal amount of area, but suppose instead that we wanted plots 2 and 4 to be half the width of plots 3 and 4. We cannot accomplish this with `par(mfrow)` because `par(mfrow)` splits the plot into a perfect grid. To create a multi-panel plot with differently-sized grid cells, we will use the `layout()` function. The `layout()` function creates a plot layout based on a matrix representing the relative size and location of each individual plot. With `layout()` you are limited only by your ability to create a matrix to represent the plot layout. If you assign `layout()` to an object, you can view the layout before plotting using the `layout.show()` function. In the following examples, we will create and view different plot layouts, but because we have already had lots of practice plotting data, will not worry about creating actual plots to fill them.

Plotting using the `layout()` function is similar to `par(mfrow)` in that the plot is filled in the order in which plots are called, except in this case, they are filled in numerical order, rather than by row or column, with the location of each plot depending on the placement of the number representing that plot in the layout matrix. We will use the `layout()` function to create a number of more advanced multi-panel plots. The key is defining the layout matrix.

1. Create a 2 x 2 plot with column 2 half the width of column 1. Within the `layout()` function, we can specify the row and column widths by supplying optional vectors containing the relative width and height of the plots.

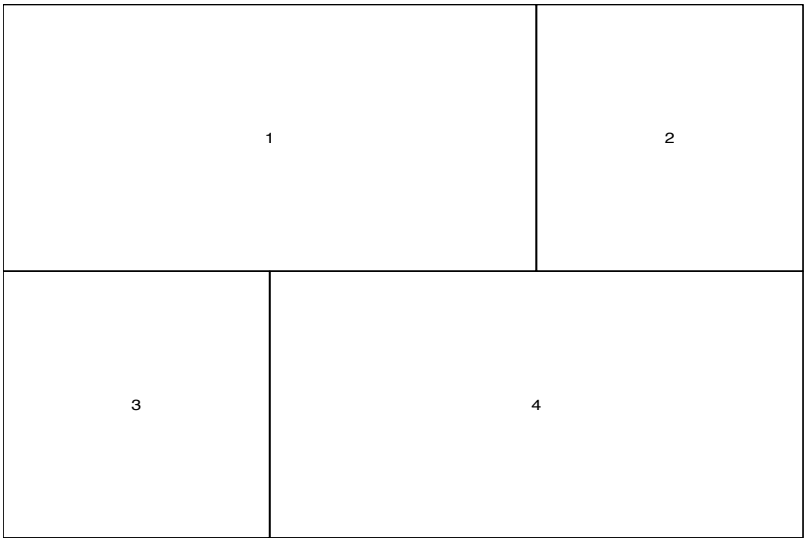
```
# Create a layout matrix to pass to the `layout()` function.
layout_matrix <- matrix(c(1:4), ncol=2, byrow=TRUE)
```

```
# View the layout matrix.
layout_matrix

##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4

# View the layout created using the `layout()` function. We specify that the
# width of column 2 should be half that of column one using the `widths`
# argument.
layout.show(layout(layout_matrix, heights=c(1, 1), widths=c(1, 0.5)))
```



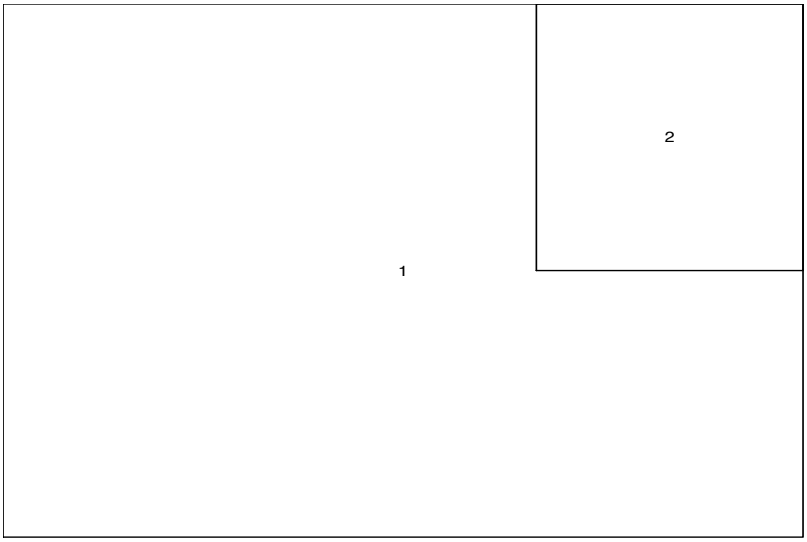
2. Create a 2 x 2 plot with plots 2 and 3 half the width of plots 1 and 4. Note that in this case we cannot use the heights and widths arguments because we are not manipulating the entire row or column in the same way. We must create the layout matrix manually!

```
# Create the layout matrix.
layout_matrix <- matrix(c(1, 1, 2, 3, 4, 4), ncol=3, byrow=TRUE)
# View the layout matrix.
layout_matrix

##      [,1] [,2] [,3]
## [1,]    1    1    2
## [2,]    3    4    4

# View the plot layout.
layout.show(layout(layout_matrix))
```



3. Create a plot with a second, smaller inset plot in the top right hand corner. The size of the inset plot could be changed by adding rows and columns to the layout matrix, or by supplying optional ***heights*** and ***widths*** arguments in the call to ***layout()***.

```
# Create the layout matrix.
layout_matrix <- matrix(c(1, 1, 2, 1, 1, 1), ncol=3, byrow=TRUE)
# View the layout matrix.
layout_matrix

##      [,1] [,2] [,3]
## [1,]    1    1    2
## [2,]    1    1    1

# View the plot layout.
layout.show(layout(layout_matrix))
```



Importing and Exporting Data

Importing Data into R: Read Functions

There are a number of different ways to import data into the R environment. For tabular data, the most straight-forward way is to read the data in from csv files. R has built-in functions for doing this (e.g., `read.csv()` and `read.table()`). These functions coerce the data into a data frame object. R can read directly from Microsoft Excel workbooks (`xlsx` package, for example), but in most cases, we've found it more reliable to simply save individual worksheets to csv file first and then read the data in using base R functions like `read.csv()`. Non-tabular text files can by imported into the R environment using the base R function `readLines()`.

Below, we will read in sample detection data using the `read.csv()` function. There are a number of arguments that can be passed to `read.csv()`, but the most common are `read.csv(file='filepath', header=TRUE, as.is=TRUE)`. The `header=TRUE` argument indicates whether the first line of data in the csv file contains column headers and the `as.is=TRUE` argument prevents R from converting strings to factors. In most cases, you will not want your character strings converted to factors. It is a good practice to check the structure of objects when they are created. This can be done using the base R function `str()`. The output of `str()` indicates the object type, number of elements, and the type of data within the object. Other functions that are useful for checking objects include: `head()` (displays first 6 elements of an object), `tail()` (displays last 6 elements of an object), and `class()` (displays the class of an object).

If a full file path (e.g., 'C:/Users/JSmith/Desktop') is not supplied for the `file` argument in `read.csv()` and `read.table()`, R will assume the `file` string is a relational file path starting at the current working directory. You can determine your current working directory using the `getwd()` function. Similarly, you can set or change your working directory using `setwd()`, providing the file path for the new working directory as a string in the parentheses. RStudio offers a shortcut for both `getwd()` and `setwd()` under the 'More' drop-down menu on the 'File' tab. Setting the working directory also allows you to specify where your output files are saved.

In the code below we will read in a csv and store it as a data frame object named `det_csv`. We will the use the `str()` function to check the structure of the data frame. First, we must set the path to the 'walleye_detections.csv' example dataset in the `glatos` package.

```
# Set path to walleye_detections.csv example dataset
det_csv <- system.file('extdata', 'walleye_detections.csv',
                       package='glatos')
# Read in a csv and store it in a data frame named `det_csv`.
detections_csv <- read.csv(file=det_csv, header=TRUE, as.is=TRUE)
# You can ceck the structure of the data frame using `str(detections_csv)`.
```

The `read.csv()` function reads timestamps in as character strings, so we will use the `as.POSIXct()` function discussed in an earlier section to convert the timestamps in our detection data to POSIXct (i.e., date time format). Note that GLATOS detections are in UTC ('GMT') time.

```
# Check class of `detections_csv$detection_timestamp_utc` `using class()`
# function. Note it is a character string.
class(detections_csv$detection_timestamp_utc)

## [1] "character"
```

```
# Convert columns with datetime stamps to POSIXct.
detections_csv$detection_timestamp_utc <-
 as.POSIXct(detections_csv$detection_timestamp_utc, tz='GMT')
detections_csv$utc_release_date_time <-
 as.POSIXct(detections_csv$utc_release_date_time, tz='GMT')
# Confirm the timestamp columns are now POSIXct objects using the class()
# function.
class(detections_csv$detection_timestamp_utc)

## [1] "POSIXct" "POSIXt"

class(detections_csv$utc_release_date_time)

## [1] "POSIXct" "POSIXt"
```

The `detections_csv` data frame contains a number of columns that users may not use in their day-to-day analyses. In order to save space, we will create a new data frame containing only the columns: 'animal_id', 'detection_timestamp_utc', 'glatos_array', 'transmitter_id', 'deploy_lat', and 'deploy_long'.

```
# Reduce the size of the dataset by subsetting out on those columns we deem
# necessary for analysis.
detections_csv_reduced <- detections_csv[,c('animal_id',
                     'detection_timestamp_utc',
                     'glatos_array',
                     'transmitter_id',
                     'deploy_lat',
                     'deploy_long')]
# View detections_csv_reduced.
head(detections_csv_reduced)

##   animal_id detection_timestamp_utc glatos_array transmitter_id deploy_lat
## 1       153     2012-04-29 01:48:37          TTB          32054   43.39165
## 2       153     2012-04-29 01:52:55          TTB          32054   43.39165
## 3       153     2012-04-29 01:55:12          TTB          32054   43.39165
## 4       153     2012-04-29 01:56:42          TTB          32054   43.39165
## 5       153     2012-04-29 01:58:37          TTB          32054   43.39165
## 6       153     2012-04-29 02:01:22          TTB          32054   43.39165
##   deploy_long
## 1   -83.99264
## 2   -83.99264
## 3   -83.99264
## 4   -83.99264
## 5   -83.99264
## 6   -83.99264
```

Exporting Data to File

Now that we have filtered out the unwanted columns from our original detection file, we may want to write out a new file so that we can use it for subsequent analyses, rather than having to perform the filter separately each time we do an analysis. There are a few different options available to us. The two we will present here are: 1) writing the data to csv, and 2) saving the object as an rds file.

Write to CSV: The write.csv() Function

The `write.csv()` function is the reverse of the `read.csv()`. It creates a csv text file. By default, the resulting csv file will include the R-assigned row names (i.e., row numbers) in the csv file. You can override this behavior using the `row.names=FALSE` argument.

```
# Write detections_csv_reduced to csv.
write.csv(detections_csv_reduced, 'detections_reduced.csv', row.names=FALSE)
```

When using `write.csv()`, the object properties (e.g., column data types) are not retained. To demonstrate this, we will read the new 'detections_reduced.csv' file back into R and examine its structure using the `str()` function. Notice that `dtf4$detection_timestamp_utc`, which we previously converted to POSIXct, is now back to being a character string.

```
# Read the newly created csv file back into R, assigning it to a data frame
# object named `detections_reduced`.
detections_reduced <- read.csv('detections_reduced.csv', as.is=TRUE)
# Check the structure of `detections_reduced`. Note that the timestamps are
# back to being character strings.
str(detections_reduced)

## 'data.frame':    7180 obs. of  6 variables:
##  $ animal_id              : int  153 153 153 153 153 153 153 153 153 153 ...
##  $ detection_timestamp_utc: chr  "2012-04-29 01:48:37" "2012-04-29 01:52:55"
"2012-04-29 01:55:12" "2012-04-29 01:56:42" ...
##  $ glatos_array           : chr  "TTB" "TTB" "TTB" "TTB" ...
##  $ transmitter_id         : int  32054 32054 32054 32054 32054 32054 32054
32054 32054 32054 ...
##  $ deploy_lat             : num  43.4 43.4 43.4 43.4 43.4 ...
##  $ deploy_long            : num  -84 -84 -84 -84 -84 ...
```

Write to RDS: The saveRDS() Function

A second option for writing data to file is to use the base R `saveRDS()` function. There are two main advantage of this method over `write.csv()`. First, the resulting rds file retains the object attributes (e.g., data types). Second, the compressed rds file will be a much smaller file than the csv file, which means it will load faster.

We'll now save the `detections_csv_reduced` data frame to rds file.

```
# Save `detections_csv_reduced` to rds.
saveRDS(detections_csv_reduced, 'detections_reduced.rds')
```

Now let's read the rds file back into R and examine its properties. Note that `detections_reduced$detections_timestamp_utc` is now a POSIXct object.

```
# Read in the newly created rds file and assigning it to a data frame object
# named `detections_reduced`.
detections_reduced <- readRDS('detections_reduced.rds')
# View the structure of `detections_reduced`. Note that the timestamps are
# still POSIXct objects.
str(detections_reduced)

## 'data.frame':    7180 obs. of  6 variables:
##  $ animal_id              : int  153 153 153 153 153 153 153 153 153 153 ...
##  $ detection_timestamp_utc: POSIXct, format: "2012-04-29 01:48:37" "2012-04-
```

```
29 01:52:55" ...
##  $ glatos_array          : chr  "TTB" "TTB" "TTB" "TTB" ...
##  $ transmitter_id        : int  32054 32054 32054 32054 32054 32054 32054
32054 32054 32054 ...
##  $ deploy_lat            : num  43.4 43.4 43.4 43.4 43.4 ...
##  $ deploy_long           : num  -84 -84 -84 -84 -84 ...
```

Now we will compare the file sizes using the base R `file.info()` function.

```
# Assign the file size attributes of the csv and rds file to objects `a` and
# `b`, respectively.
a <- file.info('detections_reduced.csv')$size
b <- file.info('detections_reduced.rds')$size
# Print a statement comparing the size of the files.
print(paste0('The size of the detections_reduced.csv file is ', a,
       ' bytes, while the size of the detections_reduced.rds file is ',
       b, ' bytes. The detections_reduced.csv file is ', round(a/b, 1),
       ' times larger than the FilteredData_3600s.rds file.'))

## [1] "The size of the detections_reduced.csv file is 389390 bytes, while the
size of the detections_reduced.rds file is 28320 bytes. The
detections_reduced.csv file is 13.7 times larger than the
FilteredData_3600s.rds file."
```

If you wish to save multiple objects to file simultaneously, you can use the `save()` function and save a file as a `.RData` file, which can then be loaded into R using the `load()` function. We will not go into `RData` files in detail here (for more details enter `?save` into the R Console), but there are two major differences between `RData` files and `Rds` files: 1) `RData` files can store more than one object, `Rds` files cannot; 2) When reading in `Rds` files, the object contained within is assigned a name upon loading. In contrast, objects loaded into R from `RData` files retain their original names.

Summarizing Data: The table(), aggregate(), and plyr::ddply() Functions

The ability to create data summaries is an important component of analyzing telemetry data. There are many built-in functions to help summarize data including: `summary()`, `fivenum()`, `mean()`, `sd()`, `median()`, `sum()`, and `range()`. However, often researchers are interested in summarizing data based on some grouping variable like sex, origin, or age. While one could calculate summaries for each level of the variable using data frame subsetting, there are a number of R functions available that simplify the process.

Using the table() Function

Suppose that we want to determine how many detections occurred for each transmitter in `detections_reduced`. One way to do this is using the base `table()` function. The `table()` treats the input element like a factor and counts the number of times each level of the factor appears in the data element.

```
# Summarize the number of times each unique value (level) of
# `detections_reduced$transmitter_id` occurs in the
# `detections_reduced$transmitter_id` vector.
table(detections_reduced$transmitter_id)
```

```
##
## 16173 16190 32054
##  2807  1327  3046
```

The output of **`table()`** can be awkward to work with, but it can be coerced into a data frame using the **`as.data.frame()`** function. Remember that character strings are converted to factor by default, so you may want to suppress that behavior using the **`stringsAsFactors=FALSE argument`**.

```
# Coerce the output table from above into a data frame.
as.data.frame(table(detections_reduced$transmitter_id),
       stringsAsFactors=FALSE)

##    Var1 Freq
## 1 16173 2807
## 2 16190 1327
## 3 32054 3046
```

By default, **`table()`** only includes values present in the input data in the outputted results. However, you can add additional values to be counted by manually converting the input data object to a factor and supplying all levels of that factor that you wish to appear in the output. A practical example of when you might do this is if you were counting the number of detections across GLATOS arrays, but your fish weren't detected on some of them. By default, **`table()`** will only list arrays the fish were detected on (i.e., will not include arrays with zero detections), but it may be just as important to know where the fish weren't detected.

Below is an example of how **`factor()`** can be used to add zero counts to **`table()`** results. We will create a vector of **`glatos_array`** values. Each occurrence of a **`glatos_array`** represents a detection on that array. We are interested in knowing how many times each of our fish was detected on the following arrays: c('SGR', 'THB', 'MAU', 'RAR', 'DRU', 'DRF'). To make the data easier to read, we will coerce the table into a data frame using the **`as.data.frame()`** function. Note that our dataset contains detections that do not appear in our specified levels. They do not appear in the table output.

```
# Create a list of glatos arrays of interest to be used as levels.
ga <- c('SGR', 'THB', 'MAU', 'RAR', 'DRU', 'DRF')

as.data.frame(table(detections_reduced$transmitter_id,
          factor(detections_reduced$glatos_array, levels=ga)),
       stringsAsFactors=FALSE)

##     Var1 Var2 Freq
## 1  16173  SGR    0
## 2  16190  SGR    0
## 3  32054  SGR  171
## 4  16173  THB    0
## 5  16190  THB    0
## 6  32054  THB  362
## 7  16173  MAU  821
## 8  16190  MAU  813
## 9  32054  MAU    0
## 10 16173  RAR 1765
## 11 16190  RAR    0
## 12 32054  RAR    0
## 13 16173  DRU    0
## 14 16190  DRU  169
## 15 32054  DRU    0
## 16 16173  DRF    0
```

```
## 17 16190   DRF   62
## 18 32054   DRF    0
```

Using aggregate()

The table function produces contingency tables of count data, but what if we wanted to create a summary containing a list of all GLATOS arrays that each tag was detected on? We could use the ***aggregate()*** function from the built-in ***stats*** package. The ***aggregate()*** function splits the data into subsets and applies one or more function to each subset. Results are combined and returned as a data frame.

The ***aggregate()*** function coerces the input data object to a data frame. As we learned earlier, the default behavior for data frame methods is to convert character strings to factors. This means that if we were to simply pass ***detections_reduced$glatos_array*** (a vector) to the function, aggregate would coerce it to a data frame and, in the process, convert the character strings in that vector to factors. To prevent this, we can coerce the vector to a data frame ourselves using the ***as.data.frame()*** function. As we have done previously, we will prevent conversion of our data to factors using the ***stringsAsFactors=FALSE*** argument.

```
# Create a data frame containing a list of all the GLATOS arrays each
# transmitter was detected on.
array_list <- aggregate(as.data.frame(detections_reduced$glatos_array,
                                      stringsAsFactors=FALSE),
                        by=list(detections_reduced$transmitter_id),
                        FUN=function(x) paste(unique(x), collapse=', '))
# View the summary data frame.
array_list

##   Group.1                  detections_reduced$glatos_array
## 1   16173                                   MAU, RAR, TSR
## 2   16190                      MAU, DRL, DRF, DRU, SCL, SCM
## 3   32054 TTB, SGR, SBI, SBO, OSC, THB, PRS, FMP, STG, SHR
```

Using ddply()

A similar function to ***aggregate()*** is the ***ddply()*** function from the ***plyr*** package. Like ***aggregate()***, ***ddply()*** subsets data frames and applies one or more function to each subset. Results are combined and returned as a data frame. An advantage of ***ddply()*** over ***aggregate()*** is the ability to easily apply more than one function to the data frame at once (i.e., appends more than one column to the data frame at a time). This is also possible with ***aggregate()***, but it is not as intuitive, and ***aggregate()*** returns a data frame with a matrix element when multiple functions are applied in a single call (as opposed to ***ddply()***, which simply appends more columns to the data frame).

Let's make the same summary as above using ***ddply()***, but this time we will also add a column indicating the number of unique glatos arrays each fish was detected on.

```
# Load the plyr package
require(plyr)

## Loading required package: plyr

# Create the summary using `ddply()`. `unique()` returns a vector of unique
# GLATOS arrays each fish was detected on, so we must paste the values within
# the vector together to make a single character string (we cannot put a
```

```
# vector object in a data frame element).
array_list_2 <- plyr::ddply(detections_reduced, plyr::.(transmitter_id),
                            summarise,
                            num_glatos_arrays=length(unique(glatos_array)),
                            glatos_arrays=paste0(unique(glatos_array),
                                                 collapse=', '))
# View the summary data frame.
array_list_2
```

```
##   transmitter_id num_glatos_arrays
## 1          16173                 3
## 2          16190                 6
## 3          32054                10
##                                      glatos_arrays
## 1                                    MAU, RAR, TSR
## 2                      MAU, DRL, DRF, DRU, SCL, SCM
## 3 TTB, SGR, SBI, SBO, OSC, THB, PRS, FMP, STG, SHR
```

We'll now extract the GLATOS arrays data column from `array_list_2` using standard data frame named list notation.

```
array_list_2$glatos_arrays
```

```
## [1] "MAU, RAR, TSR"
## [2] "MAU, DRL, DRF, DRU, SCL, SCM"
## [3] "TTB, SGR, SBI, SBO, OSC, THB, PRS, FMP, STG, SHR"
```

There are a number of other functions in `plyr` that are useful for manipulating data frames. We recommend reading through the package documentation to cater data manipulation to your specific needs.

GLATOS Package for R

glatos: An R package for the Great Lakes Acoustic Telemetry Observation System

glatos is an R package with functions useful to members of the Great Lakes Acoustic Telemetry Observation System (http://glatos.glos.us). Functions may be generally useful for processing, analyzing, simulating, and visualizing acoustic telemetry data, but are not strictly limited to acoustic telemetry applications.

Package status

This package is in early development and its content is evolving. To access the package or contribute code, join the project at (https://gitlab.oceantrack.org/GreatLakes/glatos). If you encounter problems or have questions or suggestions, please post a new issue or email cholbrook@usgs.gov (maintainer: Chris Holbrook).

Installation

Installation instructions can be found at (https://gitlab.oceantrack.org/GreatLakes/glatos/wikis/installation-instructions)

Contents

Data loading and processing

1. read_glatos_detections and read_otn_detections provide fast data loading from standard GLATOS and OTN data files to a single structure that is compatible with other glatos functions.

2. read_glatos_receivers reads receiver location histories from standard GLATOS data files to a single structure that is compatible with other glatos functions.

3. read_glatos_workbook reads project-specific receiver history and fish taggging and release data from a standard glatos workbook file.

4. read_vemco_tag_specs reads transmitter (tag) specifications and operating schedule.

5. real_sensor_values converts 'raw' transmitter sensor (e.g., depth, temperature) to 'real'-scale values (e.g., depth in meters) using transmitter specification data (e.g., from read_vemco_tag_specs).

Filtering and summarizing

1. min_lag facilitates identification and removal of false positive detections by calculating the minimum time interval (min_lag) between successive detections.

2. detection_filter removes potential false positive detections using "short interval" criteria (see *min_lag*).

3. detection_events distills detection data down to a much smaller number of discrete detection events, defined as a change in location or time gap that exceeds a threshold.

4. summarize_detections calculates number of fish detected, number of detections, first and last detection timestamps, and/or mean location of receivers or groups, depending on specific type of summary requested.

Simulation functions for system design and evaluation

1. calc_collision_prob estimates the probability of collisions for pulse-position-modulation type co-located telemetry transmitters. This is useful for determining the number of fish to release or tag specifications (e.g., delay).

2. receiver_line_det_sim simulates detection of acoustic-tagged fish crossing a receiver line (or single receiver). This is useful for determining optimal spacing of receviers in a line and tag specifications (e.g., delay).

3. crw_in_polygon, transmit_along_path, and detect_transmissions individually simulate random fish movement paths within a water body (*crw_in_polygon*: a random walk in a polygon), tag signal transmissions along those paths (*transmit_along_path*: time series and locations of transmissions based on tag specs), and detection of those transmittions by receivers in a user-defined receiver network (*detect_transmissions*: time series and locations of detections based on detection range curve). Collectively, these functions can be used to explore, compare, and contrast theoretical performance of a wide range of transmitter and receiver network designs.

Visualization and data exploration

1. abacus_plot is useful for exploring movement patterns of individual tagged animals through time.

2. detection_bubble_plot is useful for exploring distribution of tagged individuals among receivers.

3. interpolate_path, make_frames, and make_video Interpolate spatio-temporal movements, between detections, create video frames, and stitch frames together to create animated video file using *FFmpeg* software.

4. adjust_playback_time modify playback speed of videos and optionally convert between video file formats. Requires *FFmpeg*

Functions for Loading GLATOS Data into R

The `glatos` package contains custom functions to ensure that importing GLATOS data into R is done efficiently and consistently. Functions within the `glatos` package have been designed to process data in the format returned by the load functions, so using the `glatos` load functions will generally ensure that the resulting data conform to the requirements of other functions in the package. If `glatos` load functions are not used, then users will need to ensure that their data meet the requirements of each function used. Currently there are three functions, one for each: 1) GLATOS detection export file (.csv), 2) GLATOS receiver locations file (.csv), and 3) GLATOS project workbook (.xlsx). The purpose of these functions is two-fold. First, they facilitate faster data loading and formatting than base R functions like `read.csv()` and `as.POSIXct()` because they use efficient functions from the `data.table` and `fasttime` packages. Second, they check the format of incoming data to ensure they conform to the requirements of functions contained within the `glatos` package and assign a class to the object. All of the functions within the `glatos` package are designed to work directly from the GLATOS files (i.e., they use the column names from the GLATOS files).

Aside: The `data.table` package contains many fast and efficient functions (e.g., `fread()` for importing csv files) that can be used in place of base R functions, but the syntax in `data.table` is different than in base R. Note that data table objects created with functions from the `data.table` package are also of class 'data.frame', so they can be manipulated in the same way as data frames. We do not cover `data.table` in this manual, but encourage interested users to investigate the `data.table` package on their own. If you are interested in learning more about `data.table`, visit the package home page (https://github.com/Rdatatable/data.table/wiki) or, after installing the function in R, view the package vignettes (`browseVignettes('data.table')`).

In the sections below, we will demonstrate each of `glatos` data loading functions by using them to read in example GLATOS files that are contained within the `glatos` package. Also note that additional examples of the `glatos` functions may be found in the package help by running `?glatos`.

Detections: The read_glatos_detections() Function

The `read_glatos_detections()` function reads in detection data from standard detection exports (.csv files) provided by the GLATOS database and compares the structure to schema that are defined in the package to ensure the data conforms with requirements for functions within the `glatos` package. In addition to reading the data in, the function converts timestamps to class 'POSIXct' and dates to class 'Date'.

A significant advantage of using the `read_glatos_detections()` function to import GLATOS detection data into R is speed. To demonstrate just how much faster the `read_glatos_detections()` function is than the traditional workflow (i.e., `read.csv()`, `as.POSIXct()`, `as.Date()`), we will use both techniques to load in example GLATOS detection data and time both processes using the `Sys.time()` function. The `Sys.time()` function returns the current time on the computer clock. Therefore, by calling the function before and after a section of code runs, you can subtract the two values to determine the time it took to run the code.

First, we will use ***system.file()*** to get the path to the walleye_detections.csv file in the ***glatos*** package.

```
# Set path to walleye_detections.csv example dataset.
det_file <- system.file('extdata', 'walleye_detections.csv', package='glatos')
```

We will now read in the walleye_detections.csv file using ***read.csv()*** and convert timestamps and dates to the appropriate format using ***as.POSIXct()*** and ***as.Date()***, respectively (i.e., the traditional workflow). The ***na.strings*** argument in ***read.csv()*** tells the computer to convert both NAs and blank characters (i.e., '') to NA. ***stringsAsFactors=FALSE*** prevents R from converting character strings to factors. We can view the structure of the resulting data frame using the ***str()*** function.

```
a <- Sys.time()
# Read in the walleye_detections.csv file.
walleye_detections <- read.csv(det_file, na.strings=c(NA, ''),
                stringsAsFactors=FALSE)
# Convert columns `detection_timestamp_utc` and `utc_release_date_time` to
# POSIXct.
walleye_detections$detection_timestamp_utc <-
 as.POSIXct(walleye_detections$detection_timestamp_utc, tz='GMT')
walleye_detections$utc_release_date_time <-
 as.POSIXct(walleye_detections$utc_release_date_time, tz='GMT')
# Convert column `glatos_caught_date` to Date.
walleye_detections$glatos_caught_date <-
 as.Date(walleye_detections$glatos_caught_date)
Sys.time()-a

## Time difference of 0.453912 secs

# View the structure of walleye_detections using `str(walleye_detections)`.
class(walleye_detections)

## [1] "data.frame"
```

Now we will load the walleye_detection.csv file using the ***read_glatos_detections()*** function and view the structure of the resulting data frame.

```
# Attach glatos package to global environment.
library(glatos)
# Get start time from computer clock.
a <- Sys.time()
# Read in the walleye_detections.csv file using `read_glatos_detections()`.
walleye_detections <- read_glatos_detections(det_file)
# Report elapsed time.
Sys.time()-a

## Time difference of 0.04646277 secs

# View the structure of walleye_detections using `str(walleye_detections)`.
class(walleye_detections)

## [1] "glatos_detections" "data.frame"
```

In both cases, the result is a data frame object, and the structure of the data frames created using the two method is the same, except that the ***read_glatos_detections()*** data frame has two classes: 'glatos_detections' and 'data.frame'. The 'glatos_detections' label means that the data conform to the data requirements of the package and therefore should work with all of the ***glatos*** functions that use detection

data. Also note that `read_glatos_detections()` was substantially (i.e., > 10 times) faster than `read.csv()` combined with `as.POSIXct()` and `as.Date()`. The example dataset has just over 7000 records in it, so the time savings may seem minimal in this example. However, imagine the time savings if you were reading in 10+ million records, as many GLATOS members do.

Receiver Locations: The read_glatos_receivers() Function

Exports from the GLATOS database include a 'GLATOS_receiverLocations_XXXXXXXX.csv' file that contains deployment, retrieval, and metadata for all receiver deployments included in the database. This information is used in the database to assign latitude and longitude values to raw detection data exported from receivers, and among other things, can be used to create receiver history plots for summarizing the status of the GLATOS receiver array during a period of interest.

The `read_glatos_receivers()` function imports a GLATOS_receiverLocations_XXXXXXXX.csv file into R, creating a data frame with classes 'data.frame' and 'glatos_receivers'.

There is a sample receiver locations dataset in the `glatos` package called 'sample_receivers.csv'. We will use `read_glatos_receivers()` to read the csv into R. We will first get the path to the example csv file.

```
# Get path to sample_receivers.csv example dataset.
rec_file <- system.file('extdata', 'sample_receivers.csv', package='glatos')
```

Now, we will load the receiver locations csv file into R using `read_glatos_receivers()` and then view the structure of the resulting data frame.

```
# Load example receiver location data into R using `read_glatos_receivers()`.
receiver_locations <- read_glatos_receivers(rec_file)
# View the structure of receiver_locations data frame.
str(receiver_locations)

## Classes 'glatos_receivers' and 'data.frame': 898 obs. of  23 variables:
##  $ station             : chr  "WHT-009" "FDT-001" "FDT-004" "FDT-003" ...
##  $ glatos_array        : chr  "WHT" "FDT" "FDT" "FDT" ...
##  $ station_no          : chr  "9" "1" "4" "3" ...
##  $ consecutive_deploy_no: int  1 2 2 2 2 2 2 2 2 1 ...
##  $ intend_lat          : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ intend_long         : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ deploy_lat          : num  43.7 45.9 45.9 45.9 45.9 ...
##  $ deploy_long         : num  -82.5 -83.5 -83.5 -83.5 -83.5 ...
##  $ recover_lat         : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ recover_long        : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ deploy_date_time    : POSIXct, format: "2010-09-22 18:05:00" "2010-11-12
15:07:00" ...
##  $ recover_date_time   : POSIXct, format: "2012-08-15 16:52:00" "2012-05-15
13:25:00" ...
##  $ bottom_depth        : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ riser_length        : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ instrument_depth    : num  NA NA NA NA NA NA NA NA NA NA ...
##  $ ins_model_no        : chr  "VR2W" "VR3" "VR3" "VR3" ...
##  $ glatos_ins_frequency : int  69 69 69 69 69 69 69 69 69 69 ...
##  $ ins_serial_no       : chr  "109450" "442" "441" "444" ...
##  $ deployed_by         : chr  "" "" "" "" ...
##  $ comments            : chr  "" "" "" "" ...
##  $ glatos_seasonal     : chr  "NO" "No" "No" "No" ...
```

```
##  $ glatos_project       : chr  "HECWL" "DRMLT" "DRMLT" "DRMLT" ...
##  $ glatos_vps           : chr  "NO" "No" "No" "No" ...
```

As with ***read_glatos_detections()***, the output data frame has two classes: 'glatos_receivers' and 'data.frame'.

The GLATOS_receiverLocations_XXXXXXXX.csv exported with the detection query is a record of every receiver deployment since the inception of GLATOS (note: our example dataset is a reduced version of the export to save on storage space). Often, a researcher will want to work with just a subset of the receiver locations data. For example, suppose you wanted to pull out only those receiver deployments that are associated with the GLATOS project 'HECWL'. Because ***receiver_locations*** is a data frame (in addition to being a 'glatos_receivers' object), subsetting is accomplished as would be done for any other data frame object.

For example, we can subset out deployments associated with the GLATOS project 'HECWL'.

```
# Subset out receiver deployments associated with the GLATOS project HECWL.
receiver_subset <-
 receiver_locations[receiver_locations$glatos_project == 'HECWL',]
# View first 2 rows of `receiver_subset`.
head(receiver_subset, 2)

##    station glatos_array station_no consecutive_deploy_no intend_lat
## 1  WHT-009          WHT          9                     1         NA
## 10 LVD-001          LVD          1                     1         NA
##    intend_long deploy_lat deploy_long recover_lat recover_long
## 1           NA   43.74216   -82.50791          NA           NA
## 10          NA   42.12930   -83.12518          NA           NA
##       deploy_date_time   recover_date_time bottom_depth riser_length
## 1  2010-09-22 18:05:00 2012-08-15 16:52:00           NA           NA
## 10 2011-04-21 14:51:00 2012-06-02 03:59:00           NA           NA
##    instrument_depth ins_model_no glatos_ins_frequency ins_serial_no
## 1                NA         VR2W                   69        109450
## 10               NA         VR2W                   69        109937
##    deployed_by comments glatos_seasonal glatos_project glatos_vps
## 1                                    NO          HECWL         NO
## 10                                   NO          HECWL         NO
```

Another common processing step that researchers might want to perform on the ***receiver_locations*** data frame is to remove deployments without associated recovery times, which signifies that no data are currently available in the database for those receiver deployments. Missing recovery times in ***receiver_locations*** are coded as NA, so they can be subsetted out using ***!is.na()***, which translates to 'is not NA'.

```
# Determine how many receiver deployments in `receiver_locations` do not have
# recovery times.
sum(is.na(receiver_locations$recover_date_time))

## [1] 2

# Note that this is equivalent to, but more compact than the following.
nrow(receiver_locations[is.na(receiver_locations$recover_date_time),])

## [1] 2

# Remove rows without recover times.
receiver_locations_with_recoveries <-
```

```
 receiver_locations[!is.na(receiver_locations$recover_date_time),]
# Verify that there are zero rows without recovery times in the new data
# frame.
sum(is.na(receiver_locations_with_recoveries$recover_date_time))

## [1] 0
```

Project Workbook: The read_glatos_workbook() Function

Those who have contributed data to the GLATOS database will be familiar with the GLATOS project workbook, an MS Excel (.xlsm) file containing project, tagging, and receiver metadata for a project. There are a number of third-party packages available for loading data into R from MS Excel files (e.g., `XLConnect`, `xlsx`); but we have included a function in the glatos package, `read_glatos_workbook()`, that specifically reads in data from the standard GLATOS project workbook, the 'locations', 'deployment', and 'recovery' worksheets into a single `receivers` data frame), and removes a few redundant columns. The output from `read_glatos_workbook()` is a list object containing three data frames (`metadata`, `animals`, and `receivers`). By default, only the standard GLATOS workbook columns are imported, but any custom (project-specific) columns added to the standard workbook worksheets can be imported along with the standard columns using the argument `read_all=TRUE`.

We will first get the file path to the example GLATOS workbook that has been added to the `glatos` package.

```
# Get path to receiver_locations_2011.csv example dataset.
wb_file <- system.file('extdata', 'walleye_workbook.xlsm', package='glatos')
```

Now we will use `read_glatos_workbook()` to import the example workbook. As mentioned above, the imported object is a list containing three data frames. We will use the `names()` function to view the objects contained within the `workbook` list object.

```
workbook <- read_glatos_workbook(wb_file)
names(workbook)

## [1] "metadata"  "animals"   "receivers"
```

The individual data frames within `workbook` are accessed using square bracket subsetting. Remember that we must use double square brackets to return the list elements as data frames (single square brackets return elements as lists).

A very common application for importing data from within the GLATOS project workbook into R is to append animal bio data (e.g., age, length, sex) to detection data. In the following example, we will append columns from the `animals` data frame in `workbook` to the `walleye_detectons` data frame using the `merge()` function. The `merge()` function behaves like a relational database query, combining row data by column data (i.e., relational keys) that appears in both data frames. Rather than appending all columns of `animals` (some already appear in the GLATOS detections export), we will only append the 'glatos_external_tag_id1', 'glatos_external_tag_id2', and 'age' columns.

We will start by creating a new data frame called `bio` containing the information we will append to `walleye_detections`. Note that we also included 'animal_id' in the subset. This column is required to link the rows in `bio` to those in `walleye_detections` (i.e., it is the relational key). Other columns like 'transmitter_id' could be used, but they must be unique to an individual fish (i.e., tags cannot have been reused).

```
# Pull `animals` out of `workbook` and subset out columns necessary for the
# merge with `walleye_detections`.
bio <- workbook[['animals']][,c('animal_id', 'glatos_external_tag_id1',
                'glatos_external_tag_id2', 'age')]
# View first 6 rows of `bio`.
head(bio)

##   animal_id glatos_external_tag_id1 glatos_external_tag_id2 age
## 1        28                    5550                    5549   7
## 2        29                    6385                    6387   3
## 3        30                    5501                    5502   7
## 4        31                    5547                    5546   8
## 5        32                    5503                    5504   6
## 6        33                    5541                    5540   7
```

Next, we will merge the ***walleye_detections*** and ***bio*** data frames using the ***merge()*** function. The ***by*** argument tells R which columns the data frames should be merged on (i.e., which columns are the relational keys).

```
# Merge the `walleye_detections` and `bio` data frames.
# Note that both a `by.x` and `by.y` argument were supplied to `merge()`,
# indicating the merge columns in the first and second data frames,
# respectively. Because the merge column in both data frames is the same, a
# single argument, `by='animal_id'`, could have been supplied.
walleye_detections <- merge(walleye_detections,
              bio,
              by.x='animal_id',
              by.y='animal_id')
# View first column names of `walleye_detections` to ensure the columns were
# appended. You can use `str()` or `head()` to view the data itself.
names(walleye_detections)

##  [1] "animal_id"                 "detection_timestamp_utc"
##  [3] "glatos_array"              "station_no"
##  [5] "transmitter_codespace"     "transmitter_id"
##  [7] "sensor_value"              "sensor_unit"
##  [9] "deploy_lat"                "deploy_long"
## [11] "receiver_sn"               "tag_type"
## [13] "tag_model"                 "tag_serial_number"
## [15] "common_name_e"             "capture_location"
## [17] "length"                    "weight"
## [19] "sex"                       "release_group"
## [21] "release_location"          "release_latitude"
## [23] "release_longitude"         "utc_release_date_time"
## [25] "glatos_project_transmitter" "glatos_project_receiver"
## [27] "glatos_tag_recovered"      "glatos_caught_date"
## [29] "station"                   "min_lag"
## [31] "glatos_external_tag_id1"   "glatos_external_tag_id2"
## [33] "age"
```

Functions for Filtering GLATOS Data

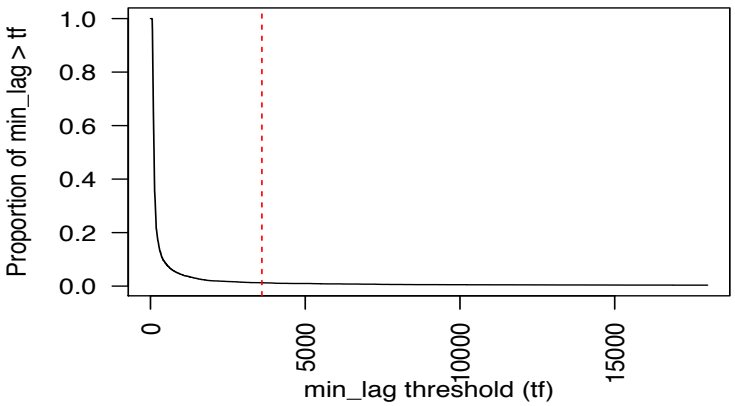
Remove Possible False Detections: The false_detections() Function

When two or more transmitters are located within detection range of a receiver, the transmitted codes can combine to produce a transmitter id that was not present at that time (i.e., a false detection). The ***false_detections()*** function flags possible false detections based loosely on the 'short-interval' criteria described by Pincock (2012; http://vemco.com/pdf/false_detections.pdf). The filter is based on the principle that the probability of two or more transmitters with random delays overlapping to produce two false detections of the same ID on a single receiver is very low during a short time interval. For example, consider a case where tag ID 12345 was detected only twice on a receiver with 12 days between those detections and a similar case where the detections were 72 seconds apart. We might assume, based on Pincock's (2012) analysis, that the latter detections (72-sec) are more likely to be real than the former (12-day), but the important question is if the detections in either case are isolated enough to consider them potentially false. Thus, a threshold time interval (***tf*** in ***false_detections()***) is often identified by which this determination is made. A common rule of thumb is to omit data when a detection is separated from others of the same code on a single receiver by 30 times the nominal delay of the transmitter (e.g., 3600 seconds for transmitters with a 120 second nominal delay). The ***false_detections()*** function facilitates false detection identification and removal by identifying detections where the time between that detection and the next closest detection of that transmitter on the same receiver (called 'min_lag' in GLATOS data) exceeds the threshold time (***tf***). The standard GLATOS detections export includes the 'min_lag' column, but the ***false_detections()*** function will also calculate 'min_lag' if absent. The ***false_detections()*** function appends a column, ***passed_filter***, to the GLATOS detection data that indicates whether the detection passed ('passed_filter'=1) or did not pass ('passed_filter'=0) the filter. When a transmitter is detected only once on a receiver, then that detection is assigned a value of NA for 'min_lag' and those records will not pass the filter. After ***false_detections()*** runs, it also prints a statement indicating how many and what proportion of detections identified as potentially false. An optional plot can be displayed showing the portion of detections that exceeded the threshold min_lag (using the ***show_plot=TRUE*** argument). This plot is useful for assessing sensitivity of the proportion of detections removed to the choice of ***tf***.

Possible false detections can be filtered out by subsetting on the 'passed_filter' column to retain only those detections that passed the filter (i.e., ***passed_filter == 1***). Note that false detection filtering comes at a cost; namely that the detections removed are likely a mixture of true and false detections. We generally recommend auxiliary analyses to asses the sensitivity of results and conclusions to the choice of time threshold (***tf***). Such analyses are beyond the scope of this document, but the optional sensitivity plot may be a useful first step.

We'll now apply the ***false_detections()*** function to our GLATOS detection data to flag possible false detections. We will then subset out detections that did not pass the filter, retaining only those detections we can be confident are true positive detections.

```
# Pass `walleye_detections` through the `false_detections()` function to
# flag possible false detections.
walleye_detections_filtered <- false_detections(det=walleye_detections,
                        tf=3600,
                        show_plot=TRUE)

## The filter identified 93 (1.3%) of 7180 detections as potentially false.
```




```
# Subset on the `passed_filter` column to retain only those detections that
# passed the filter.
walleye_detections_filtered <-
 walleye_detections_filtered[walleye_detections_filtered$passed_filter == 1,]
```

Functions for Summarizing GLATOS Data

Summarizing Detection Data by Animal and Location: The summarize_detections() Function

The ***glatos*** package contains a ***summarize_detections()*** function that summarizes detection data grouped by: 1) an animal identifier ('animal_id'), 2) a location identifier (***location_col***), or 3) both and 'animal_id' and location identifier (***location_col***). For many researchers, these types of summaries represent a good first step for analyzing detection data because they can be produced quickly, and they provide a good overview of the spatial distribution of detection data. Which summary is produced is set by the ***summ_type*** argument, where ***summ_type='animal'*** summarizes the detection data by the 'animal_id' column, ***summ_type='location'*** summarizes the detection data by a unique location identifier column (specified using the ***location_col*** argument), and ***summ_type='both*** summarizes the detection data by both 'animal_id' column and ***location_col***.

Because the summaries are produced from the detection dataset, animal IDs and locations that do not appear in the dataset are not included in the default summary. Zeros are often important, however, so ***summarize_detections()*** allows the user to specify the values of 'animal_id' and ***location_col*** that will appear in the summary. This is done using the optional arguments ***animal*** and ***receiver_locs***, respectively. ***animals*** is a vector of unique animal_ids to include in the summary and can include animal IDs that are not in the detections data frame. For example, one could pass the ***animal_id*** column from their GLATOS project workbook into the function to include all fish in their project in the summary. ***receiver_locs*** is a data frame containing at least three columns: 'deploy_lat', 'deploy_long', and a column matching the column name specified by ***location_col***.

We will use our filtered example dataset ***walleye_detections_filtered*** to demonstrate the ***summarize_detections()*** function. First, we will summarize the detection data by animal. Because this is the default for ***summarize_detections()***, we need only specify the detections dataframe in the call to the function.

```
# Summarize `walleye_detections_filtered` data frame by 'animal_id'.
animal_summary <- summarize_detections(det=walleye_detections_filtered)
# View summary.
animal_summary

##   animal_id num_locs num_dets           first_det            last_det
## 1       153        9     3014 2012-04-29 01:48:37 2013-05-09 15:10:33
## 2        22        3     2752 2012-03-27 13:05:27 2013-05-01 14:24:32
## 3        23        6     1321 2012-03-27 17:12:31 2012-05-31 02:11:16
##                           locations
## 1 FMP OSC PRS SBI SBO SGR STG THB TTB
## 2                         MAU RAR TSR
## 3             DRF DRL DRU MAU SCL SCM
```

The output from `summarize_detections()` when `summ_type=animal` is a data frame indicating the number of locations, time of first detection, time of last detection, and the unique locations at which each animal_id was detected.

Now we will summarize `walleye_detections_filtered` by location using the `summ_type=location` argument. We will summarize by 'glatos_array', which is the default value for the `location_col`, but we could supply a custom column to the to the function via the `location_col` argument.

```
# Summarize `walleye_detections_filtered` data frame by 'location'.
locations_summary <- summarize_detections(det=walleye_detections_filtered,
                                          summ_type='location')
# View summary.
head(locations_summary)

##   glatos_array num_fish num_dets           first_det            last_det
## 1          DRF        1       62 2012-05-26 15:12:15 2012-05-26 19:00:20
## 2          DRL        1      186 2012-05-25 10:10:54 2012-05-25 22:48:07
## 3          DRU        1      168 2012-05-27 07:33:37 2012-05-27 21:04:50
## 4          FMP        1      848 2012-08-23 02:31:57 2012-09-26 21:33:37
## 5          MAU        2     1625 2012-03-27 13:05:27 2012-04-09 18:33:54
## 6          OSC        1       47 2012-05-25 04:52:21 2013-05-09 15:10:33
##   mean_lat  mean_lon
## 1 42.24937 -83.11824
## 2 42.09788 -83.11929
## 3 42.34051 -82.97551
## 4 45.50114 -83.90478
## 5 41.60809 -83.57177
## 6 44.45161 -83.30285
```

The output from `summarize_detections()` when `summ_type=location` is a data frame indicating the number unique fish, number of detections, time of first detection, time of last detection, and mean latitude and longitude for each location.

We could also summarize by both animal_id and location using the `summ_type='both'` argument. In this case, `summarize_detections()` creates a summary for combinations of animal_id and locations.

```
# Summarize `walleye_detections_filtered` data frame by both animal_id and
# location.
animal_locations_summary <- summarize_detections(
 det=walleye_detections_filtered, summ_type='both')
# View firs 10 rows of summary.
head(animal_locations_summary, 10)
```

```
##    animal_id glatos_array num_dets           first_det            last_det
## 1        153          DRF        0                <NA>                <NA>
## 2        153          DRL        0                <NA>                <NA>
## 3        153          DRU        0                <NA>                <NA>
## 4        153          FMP      848 2012-08-23 02:31:57 2012-09-26 21:33:37
## 5        153          MAU        0                <NA>                <NA>
## 6        153          OSC       47 2012-05-25 04:52:21 2013-05-09 15:10:33
## 7        153          PRS      658 2012-07-26 10:23:32 2012-10-20 10:13:29
## 8        153          RAR        0                <NA>                <NA>
## 9        153          SBI      445 2012-05-23 01:24:51 2013-05-07 18:03:32
## 10       153          SBO      423 2012-05-24 08:01:20 2013-05-09 00:00:15
##    mean_lat  mean_lon
## 1  42.24937 -83.11824
## 2  42.09788 -83.11929
## 3  42.34051 -82.97551
## 4  45.50114 -83.90478
## 5  41.60809 -83.57177
## 6  44.45161 -83.30285
## 7  45.34238 -83.44450
## 8  41.63703 -82.97409
## 9  44.13322 -83.43909
## 10 44.23105 -83.41086
```

The output from ***summarize_detections()*** when ***summ_type=both*** is a data frame indicating the number of detections, time of first detection, time of last detection, and mean latitude and longitude for each animal and location combination.

In the above examples, the summaries only included levels of 'animal_id' and ***location_col*** that appeared in the detection dataset. To include animals that were not detected then we can supply a vector of animal IDs to ***summarize_detections()*** using the ***animals*** argument. Similarly, we can use the ***receiver_locs*** argument to supply receiver location data, including 'deploy_lat' and 'deploy_lon' for those locations that we want included in the summary.

Another good reason to supply ***summarize_detections()*** via ***receiver_locs*** is that when ***receiver_locs=NULL*** (the default), then mean latitude and longitude of each location ('mean_lat' and 'mean_lon' in output data frame) will be calculated only from data in the detections. Therefore, mean locations in the output summary may not represent the mean among all receiver stations in a particular group if detections did not occur on all receivers in each group. However, when actual receiver locations are specified by ***receiver_locs***, then ***mean_lat*** and ***mean_lon*** will be calculated from ***receiver_locs***.

We will now summarize the filtered walleye detections using the full list of receivers for the walleye project. Recall that ***workbook*** is a three-element list object containing three date frames: ***metadata***, ***animals***, and ***receivers***. We pass the receiver data ***summarize_detections()*** using the ***receiver_locs*** argument to create a summary of detections across all glatos arrays in the walleye example workbook. Notice that this is summarized by 'location'.

```
# Summarize detections for all glatos arrays in the example walleye workbook.
all_gatos_arrays_summary <- summarize_detections(
 det=walleye_detections_filtered, summ_type='location',
receiver_locs=workbook$receivers)
# View first 6 rows of summary.
head(all_gatos_arrays_summary)

##   glatos_array num_fish num_dets first_det last_det mean_lat  mean_lon
## 1          AGR        0        0      <NA>     <NA> 44.02782 -83.68105
```

```
## 2          ASR         0         0      <NA>     <NA> 44.41930 -83.33897
## 3          BAD         0         0      <NA>     <NA> 43.30373 -84.11176
## 4          BBI         0         0      <NA>     <NA> 45.69494 -84.41926
## 5          BBW         0         0      <NA>     <NA> 45.77278 -84.61658
## 6          BEI         0         0      <NA>     <NA> 42.33021 -83.02144
```

We now pass the animal IDs to ***summarize_detections()*** using the ***animals*** argument to ensure that all fish are included in the summary. Note that we extracted the list of unique animal IDs from the workbook and assigned that to an object named ***all_fish***, which we then passed to ***animals***. Notice that this is summarized by 'animal' and includes animals with no detections.

```
# Create a vector containing all fish.
all_fish <- workbook$animals$animal_id
# Summarize detections for all glatos arrays in the example walleye workbook.
all_gatos_arrays_summary2 <- summarize_detections(
 det=walleye_detections_filtered, summ_type='animal', animals=all_fish)
# View first six rows of summary.
head(all_gatos_arrays_summary2)

##   animal_id num_locs num_dets first_det last_det locations
## 1         1        0        0      <NA>     <NA>      <NA>
## 2        10        0        0      <NA>     <NA>      <NA>
## 3       100        0        0      <NA>     <NA>      <NA>
## 4       101        0        0      <NA>     <NA>      <NA>
## 5       102        0        0      <NA>     <NA>      <NA>
## 6       103        0        0      <NA>     <NA>      <NA>
```

Reduce Detection Data to a Series of Discrete Detection Events: The detection_events() Function

Some researchers find it useful to distill raw detection data down to a series of discrete detection events. This is most useful when fish are continuously detected in one area (i.e., accumulation of numerous sequential detections on the same receiver/array) for a period of time before being detected somewhere else. A detection event is marked by a start and end time. Usually this will be the time of the first and last of a series of sequential detections that occurred on the same receiver before the fish was detected on a different receiver. However, frequently, a long period of time will pass between sequential detections on the same receiver. This often means that the fish moved to an area that did not have receiver coverage for some period of time and then returned to the same area. We may wish to flag these instances and start a new detection event when the fish returns to the original receiver. This is accomplished by setting a threshold time that must occur between sequential detections on the same receiver for the return detection to be considered the start of a new detection event (***time_sep***). Selection of this time threshold is subjective, and will depend largely on the specific analysis being performed.

The ***detection_events()*** function applies an event filter to detection datasets. In the next example, we will apply the event filter to the dataset filtered for false detections in the previous section (i.e., ***walleye_detections_filtered***). There are two options (set using the ***condense*** argument) for how results are returned: 1) a data frame condensed so that all summary information for each detection event (i.e., 'mean_latitude', 'mean_longitude', 'first_detection', 'last_detection', 'num_detections', 'res_time_sec') are contained in a single row (default); 2) a data frame matching the original data frame with 4 columns appended ('time_diff', 'arrive', 'depart', 'event'). The appended columns contain 0 or 1 and identify the first detection of each event ('arrive'), last detection of each event ('depart'), and a numeric event identifier ('event'). In the added 'arrive' and 'depart' columns, 1 = TRUE and 0 = FALSE.

We will first produce a data frame of condensed detection events using the ***detection_events()*** function. Many researchers have found that the condensed filtered output data is easier to deal with than the raw detection data, especially if they are using programs like Microsoft Excel or Microsoft Access to manage and analyze their data. This is due to the large decrease in the number of elements (i.e., rows) in the dataset.

```
# Apply the `detection_events()` function to `walleye_detections_filtered`.
# Use threshold time of 172800 sec (48 hours) for `time_sep`.
det_events <- detection_events(walleye_detections_filtered, time_sep=172800)

## The event filter distilled 7087 detections down to 167 distinct detection
events.

# View the first 6 rows of the results.
head(det_events)

##   event Individual location mean_latitude mean_longitude
## 1     1        153      TTB      43.38991      -83.99063
## 2     2        153      SGR      43.61098      -83.87383
## 3     3        153      SBI      44.17820      -83.54592
## 4     4        153      SBO      44.24085      -83.43306
## 5     5        153      OSC      44.45198      -83.31861
## 6     6        153      THB      44.95323      -83.29662
##       first_detection      last_detection num_detections res_time_sec
## 1 2012-04-29 01:48:37 2012-04-29 02:26:07             21         2250
## 2 2012-04-30 04:46:40 2012-04-30 09:50:21             93        18221
## 3 2012-05-23 01:24:51 2012-05-23 06:57:33             50        19962
## 4 2012-05-24 08:01:20 2012-05-24 16:01:50            118        28830
## 5 2012-05-25 04:52:21 2012-05-25 05:15:33             11         1392
## 6 2012-07-15 11:15:40 2012-07-17 06:06:44            335       154264
```

Users that prefer to retain the information in the original data frame can set ***condense=FALSE***, which will simply append the detection event information to the original data frame. This approach is useful when subsequent analyses will rely on only first or last detections of the events.

In the following code, we will apply the ***detection_events()*** function to the ***walleye_detections_filtered*** data frame, but will conserve the original data.

```
# Apply the `detection_events()` function to `walleye_detections_filtered`.
# Use threshold time of 172800 sec (48 hours) for `time_sep`. Do not condense
# results.
det_events_append <- detection_events(walleye_detections_filtered,
                     time_sep=172800,
                     condense=FALSE)

## The event filter identified 167 distinct events in 7087 detections.

# View first 3 rows of `det_events_append`.
head(det_events_append, 3)

##   animal_id detection_timestamp_utc glatos_array station_no
## 1       153     2012-04-29 01:48:37          TTB          2
## 2       153     2012-04-29 01:52:55          TTB          2
## 3       153     2012-04-29 01:55:12          TTB          2
##   transmitter_codespace transmitter_id sensor_value sensor_unit deploy_lat
## 1             A69-9001          32054           NA        <NA>   43.39165
## 2             A69-9001          32054           NA        <NA>   43.39165
## 3             A69-9001          32054           NA        <NA>   43.39165
```

```
##   deploy_long receiver_sn tag_type tag_model tag_serial_number
## 1   -83.99264      113213     <NA>      <NA>              <NA>
## 2   -83.99264      113213     <NA>      <NA>              <NA>
## 3   -83.99264      113213     <NA>      <NA>              <NA>
##   common_name_e    capture_location length weight sex release_group
## 1       walleye Tittabawassee River  0.565     NA   F          <NA>
## 2       walleye Tittabawassee River  0.565     NA   F          <NA>
## 3       walleye Tittabawassee River  0.565     NA   F          <NA>
##   release_location release_latitude release_longitude
## 1    Tittabawassee               NA                NA
## 2    Tittabawassee               NA                NA
## 3    Tittabawassee               NA                NA
##   utc_release_date_time glatos_project_transmitter glatos_project_receiver
## 1   2012-03-20 20:00:00                      HECWL                   HECWL
## 2   2012-03-20 20:00:00                      HECWL                   HECWL
## 3   2012-03-20 20:00:00                      HECWL                   HECWL
##   glatos_tag_recovered glatos_caught_date station min_lag
## 1                   NO               <NA> TTB-002     258
## 2                   NO               <NA> TTB-002     137
## 3                   NO               <NA> TTB-002      90
##   glatos_external_tag_id1 glatos_external_tag_id2 age passed_filter
## 1                    6321                    6320   7             1
## 2                    6321                    6320   7             1
## 3                    6321                    6320   7             1
##   time_diff arrive depart event
## 1        NA      1      0     1
## 2       258      0      0     1
## 3       137      0      0     1
```

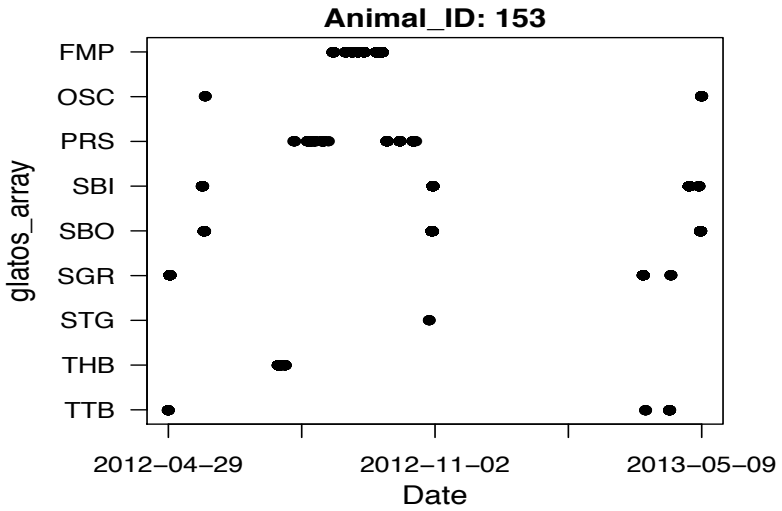
Functions for Visualizing GLATOS Data

Visualizing Detection Patterns Over Time: The abacus_plot() Function

A plot that many researchers use for visualizing telemetry data is the so-called 'abacus plot'. The abacus plot is the type of plot that is used to visualize detection data in VUE. These plots are very popular among telemetry researchers, often appearing in presentations and manuscripts, and are a great way to view detection over space and time. We have written a function called ***`abacus_plot()`*** that can be used to make these plots. Abacus plots tend to work best when they display data for a single fish at a time unless the detection history for multiple fish is very sparse.

We will now use the ***`abacus_plot()`*** function to make a plot for a single fish in our ***`walleye_detections_filtered`*** data frame (***`animal_id == 153`***).

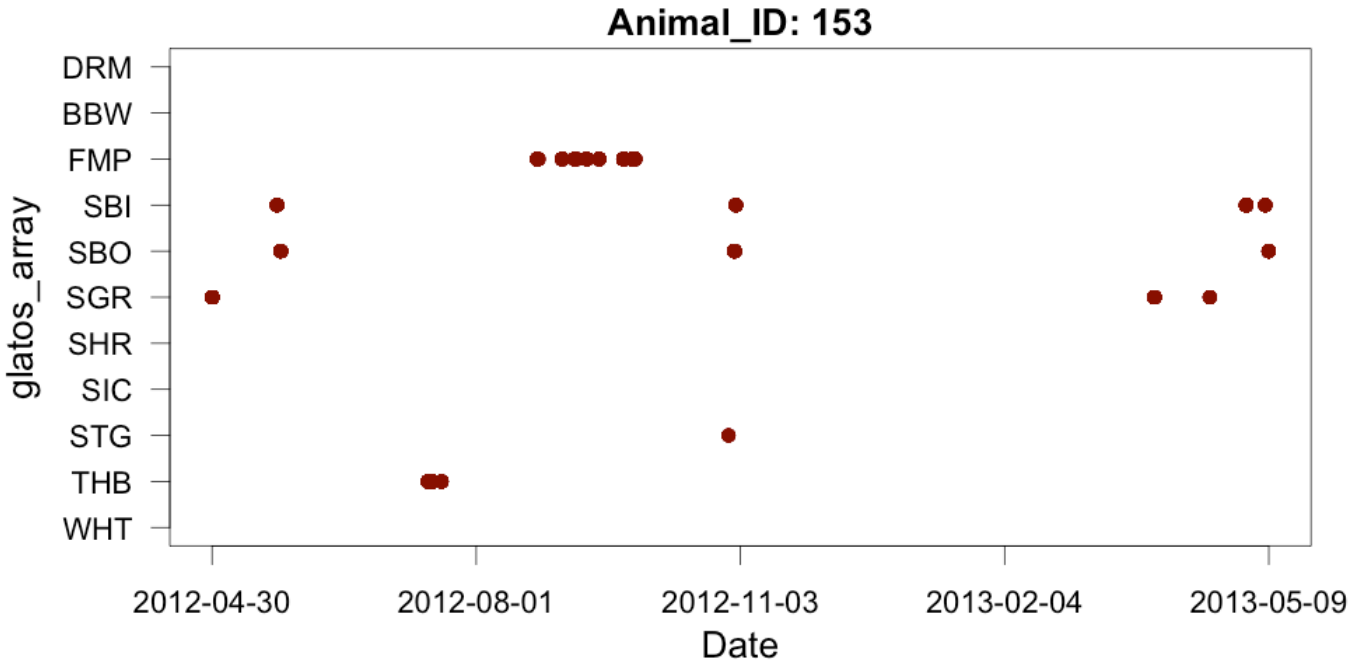
```
# Create an abacus plot for that fish and add a main title.
abacus_plot(walleye_detections_filtered[
 walleye_detections_filtered$animal_id == 153,],
 location_col='glatos_array', main='Animal_ID: 153')
```



By default, the ***abacus_plot()*** function plots only those locations that appear in the supplied ***det*** data frame (in alphabetical order from top to bottom); however, you can control what locations appear on the plot, and in what order, by supplying an optional ***locations*** vector. The ***locations*** vector should contain the locations to be plotted in the order (from bottom to top) that they should be plotted. Values for locations that do not appear in the detections data frame ***det*** are allowed (i.e., can plot locations the fish were not detected).

We will re-plot the above, but this time we will supply a ***locations*** vector so that we can control which locations appear on the plot. We will also change the color of the points to 'darkred' by supplying an optional ***col*** argument to the ***abacus_plot()*** function. By default the plot is displayed in the plot window, but we can write it to file by supplying an ***outFile*** argument, corresponding to the file name and extension for the output image. This time we will write the output to a file called 'abacus_plot.png'

```
# Create a vector of unique 'glatos_array' values to by plotted against.

locs <- c('WHT', 'THB', 'STG', 'SIC', 'SHR', 'SGR', 'SBO', 'SBI', 'FMP',
     'BBW', 'DRM')

# Create an abacus plot using a supplied control table of locations to appear
# on the y-axis.
abacus_plot(walleye_detections_filtered,
      location_col='glatos_array',
      locations=locs,
      main='Animal_ID: 153',
      col='darkred',
      outFile='TablesAndFigures/abacus_plot.png')

## Output file is located in the following directory:
## TablesAndFigures
```



Visualizing Spatial Distribution of Detections: The bubble_plot() Function

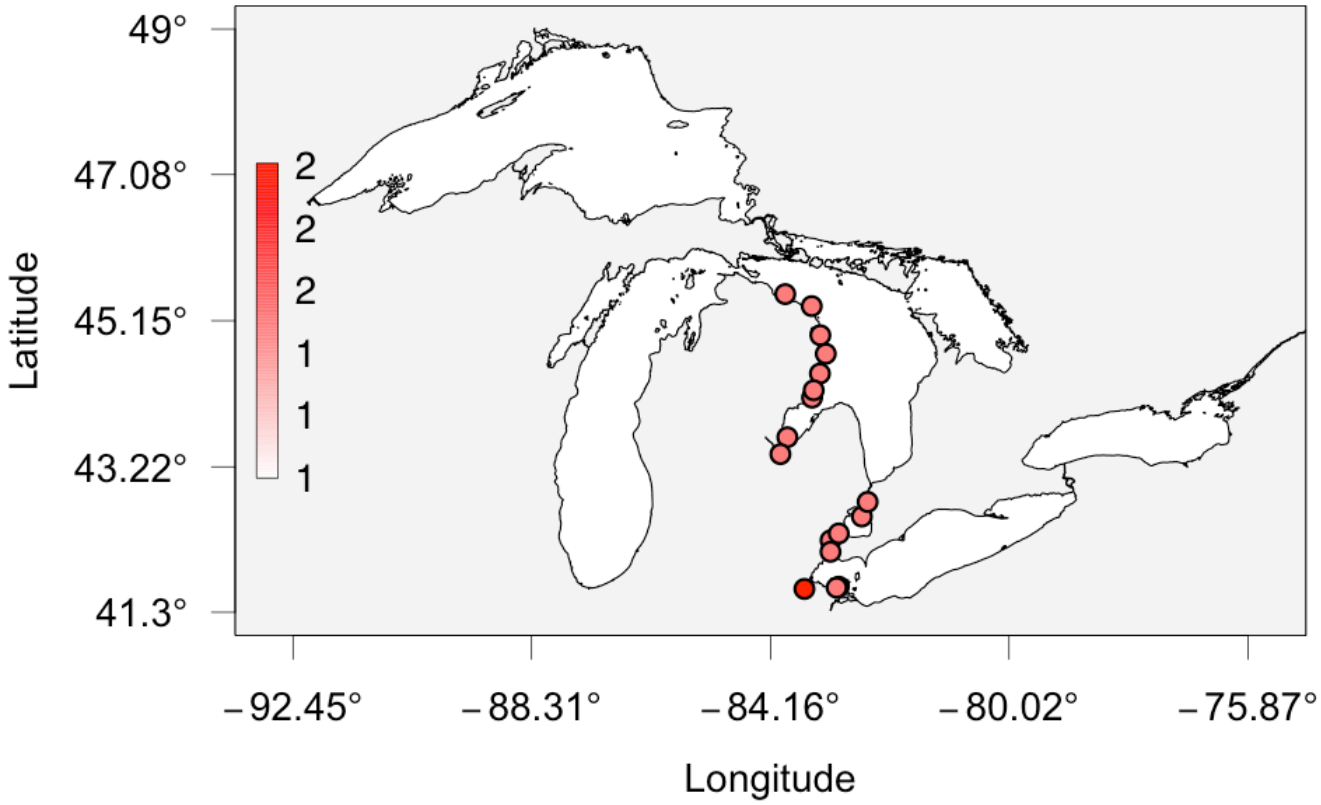
A bubble plot is useful for visualizing the spatial distribution of fish detections but provides limited information about timing of detections. The ***detection_bubble_plot()*** function plots circles (bubbles) on a map background at user-inputted locations, coloring each circle based on the number of fish detected at each location.

Like the ***abacus_plot()*** function, the ***detection_bubble_plot()*** function can 1) display only those locations where fish were detected, or 2) display a custom set of locations to be plotted independent of whether fish were deteted there or not. ***detection_bubble_plot()*** is wrapper for the ***glatos*** function ***summarize_detections()*** (with ***summ_type='locations'***) that also displays a plot. The grouping location for detections is set by the ***location_col*** argument, which by default is 'glatos_array' and plots only those locations that had detections (i.e., only locations contained within the ***det*** data frame). If the user desires a plot that includes locations the fish were not detected, they must supply an optional ***receiver_locs*** data frame containing three columns: 1) the name passed to ***location_col***, ***deploy_lat***, and ***deploy_long***.

The ***detection_bubble_plot()*** function returns a summary data frame containing the data used to create the plot (technically, the output from ***summarize_detections()***). By default, a plot is drawn to the active plot device, but the user may specify an output file using the ***file_out*** argument, which results in the plot being written to an image file (e.g., png, jpeg, or tiff).

We will now use the ***detection_bubble_plot()*** function to plot the spatial distribution of detections in our ***walleye_detections_filtered*** data frame. The default plot uses a shapefile of the Great Lakes coastline (zoomed out to its fullest extent) that is included with the package as the background.

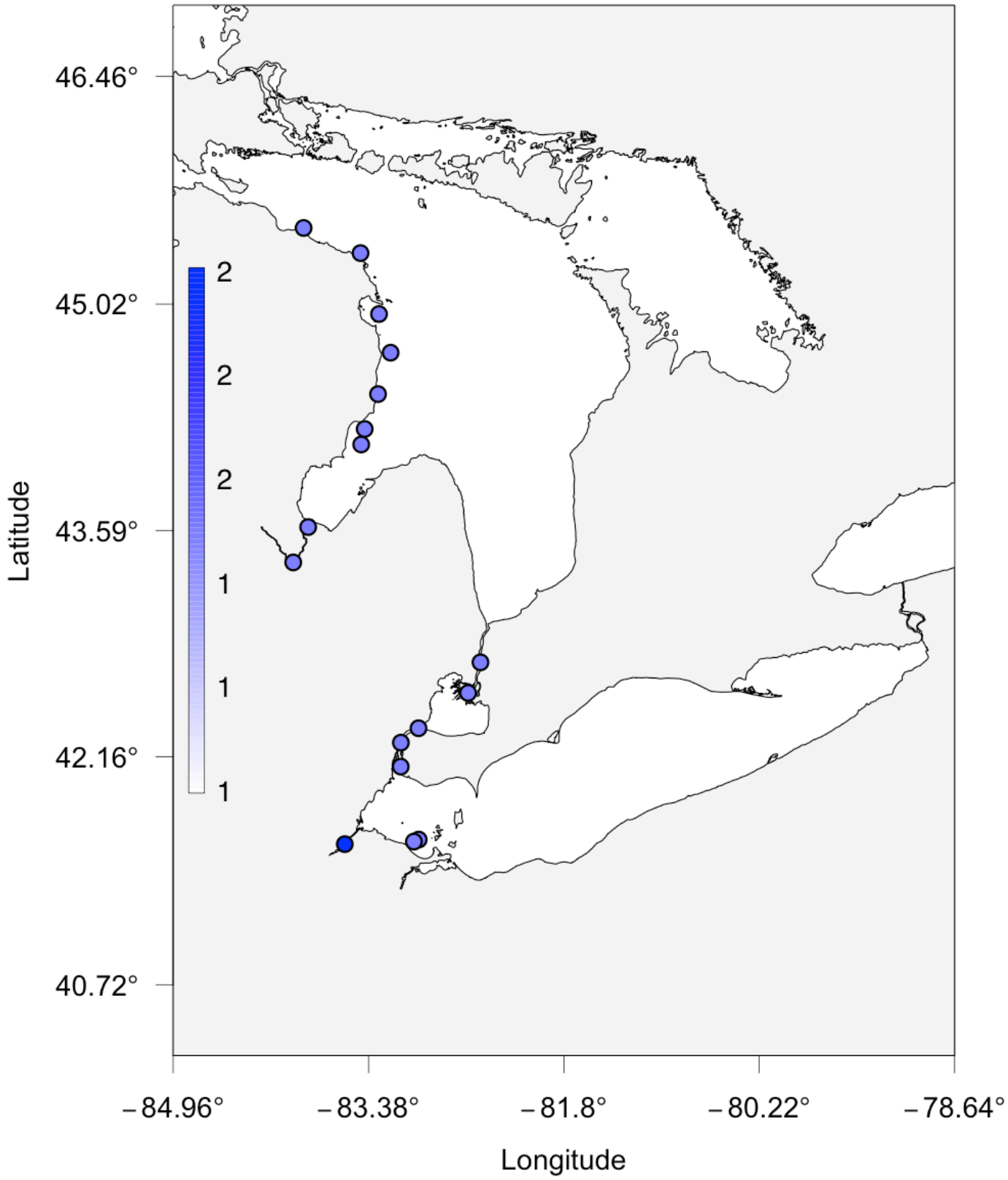
```
# Run `detection_bubble_plot()` on the `walleye_detections_filtered` data
# frame.
detection_bubble_plot(walleye_detections_filtered)
```



We can limit the viewable area of the plot by supplying the option arguments ***background_xlim*** and ***background_ylim***.

For example, let's limit the viewing area of the image to Lakes Huron and Erie. We'll also change the color gradient to blue instead of red.

```
# Run `detection_bubble_plot()` on the `walleye_detections_filtered` data
# frame, but limit the background to Lakes Huron and Erie.
detection_bubble_plot(walleye_detections_filtered,
                      background_xlim=c(-84.96, -78.64),
                      background_ylim=c(40.72, 46.46),
                      col_grad=c('white', 'blue'))
```



In the two previous examples, bubbles were plotted only at locations where fish were detected. Let's now plot bubbles everywhere that receivers were present during that period, whether the fish were detected in them or not.

To do this, we will use the receiver location data ***receiver_locations*** we loaded into R earlier using the ***read_glatos_receivers()*** function. Recall that these data are exported with detection data from the GLATOS database and contain deployment records for every GLATOS receiver deployment since the inception of the program. We will, therefore, want to filter this dataset to include only those deployments that were in the water during the period of detection we are analyzing.

Let's take a quick look at the ***receiver_locations*** data frame.

```
# View first 2 lines of `receiver_locations`.
head(receiver_locations)

##   station glatos_array station_no consecutive_deploy_no intend_lat
## 1 WHT-009          WHT          9                     1         NA
## 2 FDT-001          FDT          1                     2         NA
## 3 FDT-004          FDT          4                     2         NA
## 4 FDT-003          FDT          3                     2         NA
## 5 FDT-002          FDT          2                     2         NA
## 6 DTR-001          DTR          1                     2         NA
##   intend_long deploy_lat deploy_long recover_lat recover_long
## 1          NA   43.74216   -82.50791          NA           NA
## 2          NA   45.93014   -83.50204          NA           NA
## 3          NA   45.94764   -83.48847          NA           NA
## 4          NA   45.93794   -83.46884          NA           NA
## 5          NA   45.92377   -83.48483          NA           NA
## 6          NA   45.97745   -83.89740          NA           NA
##      deploy_date_time   recover_date_time bottom_depth riser_length
## 1 2010-09-22 18:05:00 2012-08-15 16:52:00           NA           NA
## 2 2010-11-12 15:07:00 2012-05-15 13:25:00           NA           NA
## 3 2010-11-12 15:36:00 2012-05-15 14:15:00           NA           NA
## 4 2010-11-12 15:56:00 2012-05-15 14:40:00           NA           NA
## 5 2010-11-12 16:26:00 2012-05-15 16:10:00           NA           NA
## 6 2010-11-12 19:43:00 2012-05-10 15:49:00           NA           NA
##   instrument_depth ins_model_no glatos_ins_frequency ins_serial_no
## 1               NA         VR2W                   69        109450
## 2               NA          VR3                   69           442
## 3               NA          VR3                   69           441
## 4               NA          VR3                   69           444
## 5               NA          VR3                   69           447
## 6               NA          VR3                   69           439
##   deployed_by comments glatos_seasonal glatos_project glatos_vps
## 1                                   NO          HECWL         NO
## 2                                   No          DRMLT         No
## 3                                   No          DRMLT         No
## 4                                   No          DRMLT         No
## 5                                   No          DRMLT         No
## 6                                   No          DRMLT         No
```

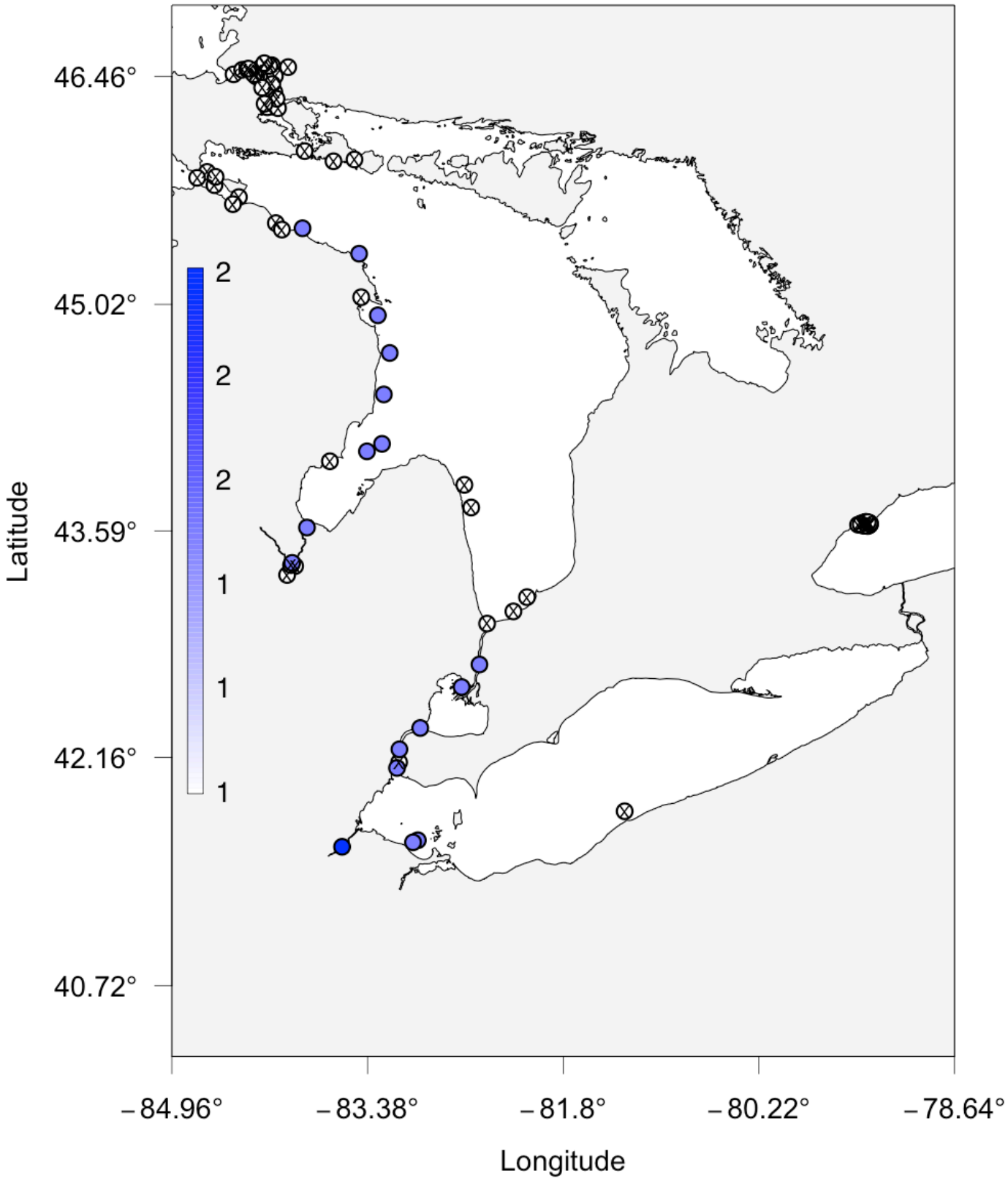
Let's now filter out deployments that were not in the water between the period of the first detection in ***walleye_detections_filtered*** and the last detections in ***walleye_detections_filtered***. First, will determine when the first and last detection in ***walleye_detections_filtered*** occurred. Next, we will filter out all records from ***receiver_locations*** that were not in the water during that period. In order for a receiver to have been in the water during that period, it must have a 'deploy_date_time' less than the last detection AND a 'recover_date_time' greater than the first detection. Receiver deployments that have not yet been recovered have an NA in the 'recover_date_time' column and must be filtered out manually, otherwise the will be retained in the dataset (recall that NA evaluates to true when subsetting!).

```
# Determine time of first and last detection in the
# `walleye_detections_filtered` data frame.
first <- min(walleye_detections_filtered$detection_timestamp_utc)
last <- max(walleye_detections_filtered$detection_timestamp_utc)

# Filter out deployments that occurred outside the period between the first
# and last detection in our `walleye_detections_filtered` data frame. Don't
# forget to filter out the NAs for the 'recover_date_time' column also.
plot_receiver_locations <-
  receiver_locations[receiver_locations$deploy_date_time < last &
                       receiver_locations$recover_date_time > first &
                       !is.na(receiver_locations$recover_date_time),]
```

We can now pass ***plot_receiver_locations*** directly to ***detection_bubble_plot()*** so that all receiver locations will appear in the bubble plot and summary. Like ***summarize_detections()***, ***detection_bubble_plot()*** if ***receiver_locs*** are not supplied, then mean latitude and longitude of each location (***mean_lat*** and ***mean_lon*** in output data frame) will be calculated from detection data only. Therefore, mean locations in the output summary may not represent the mean among all receiver stations in a particular group if detections did not occur on all receivers in each group. However, when actual receiver locations are specified by ***receiver_locs***, then mean_lat and mean_lon will be calculated from ***receiver_locs***.

We will re-plot our data using our filtered receiver locations.

```
# Run `detection_bubble_plot()` on the `walleye_detections_filtered` data
# frame, but limit the background to Lakes Huron and Erie.
detection_bubble_plot(walleye_detections_filtered,
                      location_col='glatos_array',
                      receiver_locs=plot_receiver_locations,
                      background_xlim=c(-84.96, -78.64),
                      background_ylim=c(40.72, 46.46),
                      col_grad=c('white', 'blue'))
```


46.46°
45.02°
43.59°
42.16°
40.72°
Latitude
2
2
2
1
1
1
−84.96°
−83.38°
−81.8°
−80.22°
−78.64°
Longitude

Creating Animations: The interpolate_path(), make_frames(), make_video, and adjust_playback_time() Functions

Visualizing fish movement using animations is a powerful tool for exploring movements of tagged fish and an effective strategy for quickly conveying complex movement patterns to others. Development of an animation using R is not trivial from a programming perspective and can require substantial computer processing resources. The examples of animations that we present in this guide use a 'flipbook' approach where individual images (frames) showing the geographic position of all fish during a discrete time bin are written to your computer. After images are created, the glatos package relies on ***FFmpeg*** software to stitch frames together into video animations. ***FFmpeg*** is an extremely powerful open source cross-platform video editing software (https://ffmpeg.org/download.html) that is executed via system terminal commands. Installation of ***FFmpeg*** varies by operating system and must be completed prior to calling the animation functions in the glatos package. Detailed installation instructions are outside the scope of this manual but may be found on the internet. see: https://www.wikihow.com/Install-FFmpeg-on-Windows. (windows) or http://www.idiotinside.com/2016/05/01/ffmpeg-mac-os-x/ (mac). Alternatively, all animation functions in the glatos may be used to create frames that can be stitched together using other software (i.e., Windows Movie Maker, iMovie, etc).

Creating animations is a multi-step process using the ***glatos*** package. First, detection data is interpolated using the ***interpolate_path()*** function. This function interpolates coordinates for each fish at equal time intervals (user defined) using a linear 'connect the dots' approach or a non-linear approach that calculates the shortest path between subsequent detections that avoid impossible overland movements. The second step of the workflow consists of creating individual frames, or images, for each equally spaced time bin. This step is handled by the ***make_frames()*** function. The last step is to stitch individual frames together into a animation movie using ***FFmpeg***. This step is handled by ***make_frames()*** or alternatively, ***make_video()*** if additional customization to the video is needed. In addition to these main functions, other functions are included in ***glatos*** to aid customization. The ***adjust_playback_time()*** function uses ***FFmpeg*** to speed up or slow down the playback of a video. As well, this function enables the user to convert video formats (e.g., convert a .mp4 animation to .wmv). The ***interpolate_path()*** function uses the ***shortestPath()*** function in the ***gdistance*** package to calculate the shortest path that avoids overland movements for two geographic coordinates using a raster transition layer (see ***gdistance*** package for more information). Included in the ***glatos*** package is a transition layer that distinguishes between water (cell value = 1) and land (cell value = 0) for the entire Great Lakes Basin. This dataset is basin wide and of moderate resolution such that tributaries and fine detail are not included. The ***make_transition()*** function in glatos creates a raster transition layer from a polygon shapefile, necessary for creating a customized animation. In the next sections, we will explore the ***glatos*** animation functions by creating and customizing several animations.

In the first example, we will interpolate positions for walleye detected in Lake Huron during 2012-2013.

In the code below, we load the walleye detection data and apply linear interpolation to calculate positions for all fish at daily intervals. The ***int_time_stamp*** argument controls the interpolation time step (seconds) and ***trans=NULL*** (default) specifies linear interpolation.

```
library(glatos)
dtc <- system.file('extdata', 'walleye_detections.csv', package='glatos')
dtc <- read_glatos_detections(dtc)

# Linear interpolation.
pos <- interpolate_path(dtc, trans=NULL, int_time_stamp=86400)

head(pos)
```

```
##   animal_id       bin_timestamp latitude longitude  record_type
## 1       153 2012-04-28 13:05:27 43.39165 -83.99264    detection
## 2       153 2012-04-28 13:05:27 43.38937 -83.99000    detection
## 3       153 2012-04-28 13:05:27 43.38709 -83.98737    detection
## 4       153 2012-04-29 13:05:27 43.60963 -83.88658    detection
## 5       153 2012-04-29 13:05:27 43.61235 -83.86080    detection
## 6       153 2012-04-30 13:05:27 43.61570 -83.85883 interpolated
```

The ***pos*** object contains 5 columns: 'animal_id', 'bin_timestamp', 'latitude', 'longitude', and 'record_type'. The 'animal_id' column contains the fish identification codes, 'bin_timestamp' is the equally spaced time bins for coordinates ('latitude', 'longitude') and 'record_type' displays whether the coordinates are actual detections or interpolated.

To use non-linear interpolation, assign a transition layer object to the ***trans*** argument. In this example, we use the ***greatLakesTrLayer*** included with the ***glatos*** package.

```
# Load default transition layer.
data(greatLakesTrLayer)

# Non-linear interpolation.
pos <- interpolate_path(dtc, trans=greatLakesTrLayer,
            int_time_stamp=86400,
            lnl_thresh=0.9)

# View first 6 rows.
head(pos)

##   animal_id       bin_timestamp latitude longitude  record_type
## 1       153 2012-04-28 13:05:27 43.39165 -83.99264    detection
## 2       153 2012-04-28 13:05:27 43.38709 -83.98737    detection
## 3       153 2012-04-29 13:05:27 43.60963 -83.88658    detection
## 4       153 2012-04-29 13:05:27 43.61235 -83.86080    detection
## 5       153 2012-04-30 13:05:27 43.61570 -83.85883 interpolated
## 6       153 2012-05-01 13:05:27 43.64044 -83.84426 interpolated
```

The ***interpolate_path()*** function calculates the ratio of the linear distance (possibly over land) and the non-linear (shortest point to avoid land) for each movement between geographic positions. If the distance between geographic points is substantially larger for the pathway that avoids land than the straight line distance, then non-linear interpolation is likely needed. However, if the opposite is true, we can use linear interpolation. The ***lnl_thresh*** (default = 0.9) argument allows the user to control the threshold between linear and non-linear interpolation. Calculation of non-linear solution is extremely computationally intensive compared to a linear solution and we decrease execution time by only using non-linear interpolation in situations where the fish would cross over land. In our testing, lnl_thresh values of about 0.9 seem to prevent most overland movements. If you find fish moving overland in your animations, increase ***lnl_thresh*** and if you need shorter execution times, decrease ***lnl_thresh***.

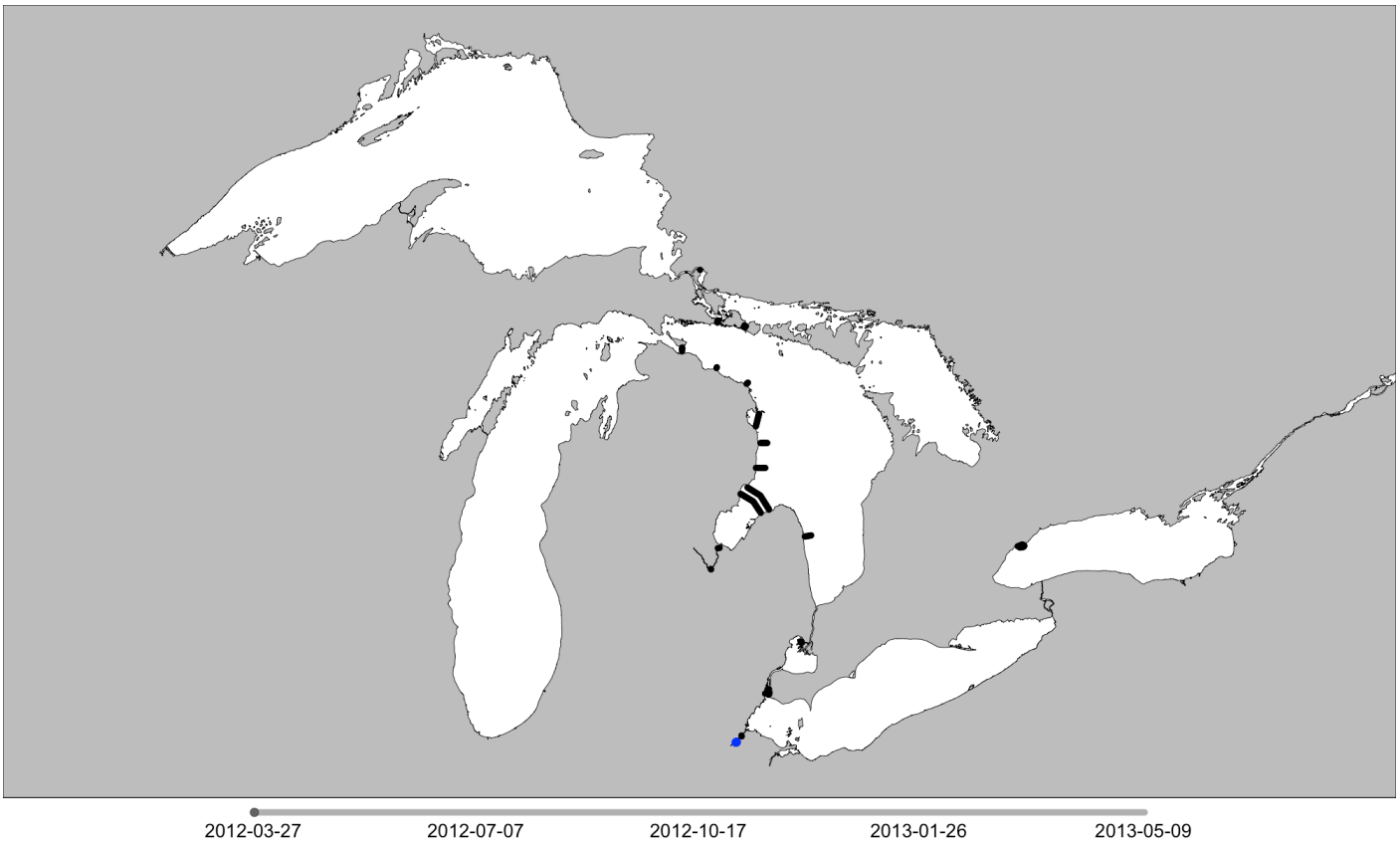
When we interpolate fish positions, we are 'making up' geographic positions that likely do not reflect the actual path (and complex behavior) of the organism. While these functions make it quick and easy to interpolate positions based on the shortest possible path within detection data, we encourage you to consider if and when this is appropriate. Be sure to carefully define and clearly distinguish interpolated and observed detections in any presentations, reports, or other information products.

Let's make some frames for a video using the interpolated data. The ***make_frames()*** function takes as an input the object created by ***interpolate_path()***. Additional arguments are available that allow the user

to control the content and output of the function. Numerous examples of ***make_frames()*** are included in the help documentation.

For the examples in this manual, we will only produce the first frame of animations by specifying the ***preview=TRUE*** argument. Likewise, we will not create the animation by setting the ***animate=FALSE***. If frames and an animation are desired, set ***preview=FALSE*** and ***animate=TRUE***.

```
# Load receiver location data.
rec_file <- system.file('extdata', 'sample_receivers.csv', package='glatos')
recs <- read_glatos_receivers(rec_file)

# Create output folder.
myDir <- paste0(getwd(),'/frames1')

# Create first frame of animation.
make_frames(pos, recs=recs, out_dir=myDir, animate=FALSE, preview=TRUE)
```



A quick note about the ***ffmpeg*** argument in ***make_frames()***:

Installation of ***FFmpeg*** on windows requires modifying the path variable on your computer to enable a system call from any directory to launch the ***FFmpeg*** program. If you are unable to change the path variable on your computer due to security constraints, you can still use ***FFmpeg*** by downloading and extracting the software on your computer and specifying a file path to the executable file on your machine. For Windows, the executable is 'ffmpeg.exe' and is usually located within the 'bin' folder of the ***FFmpeg*** download. On mac, the executable is 'FFmpeg' within the bin folder.

If you wish to fine-tune or need additional control over the creation of the animation using ***FFmpeg***, the ***make_video()*** function in ***glatos*** converts a folder of sequentially numbered images to a movie. See the help documentation for ***make_video()*** to find many examples. Notably, this function will create

animations in many formats (*.wmv, *.mp4), scale output video to a specified resolution, tweak input and output frame rates, and enable the user to specify start and end frames.

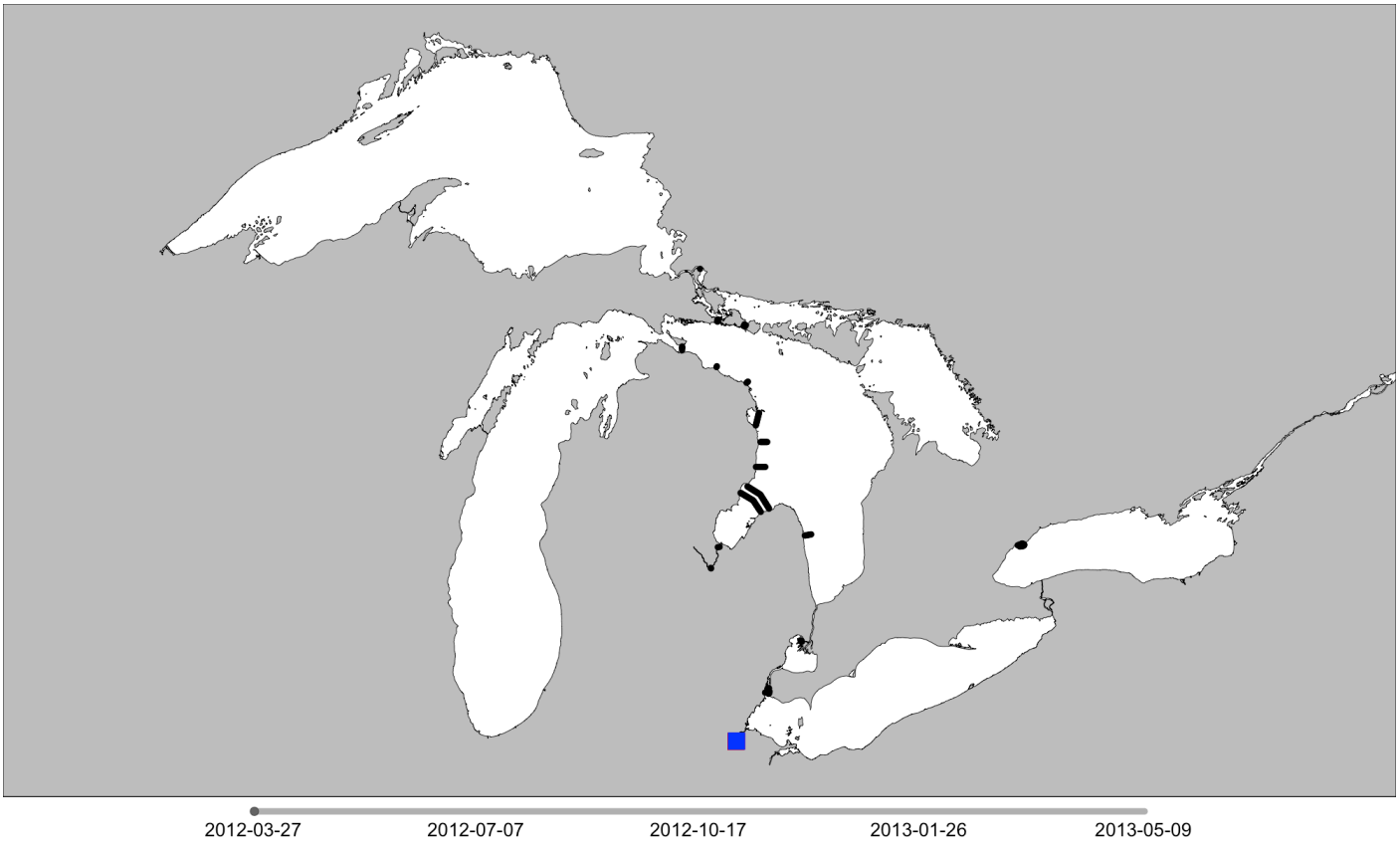
Once you have created an animation, the `adjust_playback_time()` function can be used to speed up or slow down playback of the original animation and will output a new video. This is especially useful for modifying existing animations for specific time slots in talks without having to remake frames. In addition to adjusting playback rates, `adjust_playback_time()` can change formats the format of the new animation. Check out help documentation for many examples of `adjust_playback_time()` and `make_video()`.

Customizing Animations: The make_frames() Function

`make_frames()` allows some customization of the frame content. For example, we can change the shape, size, and color of markers for fish individually. This is done by adding columns that specify plotting parameters to the output of `interpolate_path()` and then passing these columns to make_frames. In addition, you can adjust the extents of the background layer in order to zoom in on a particular location. We will demonstrate some of these customizations in the next code block.

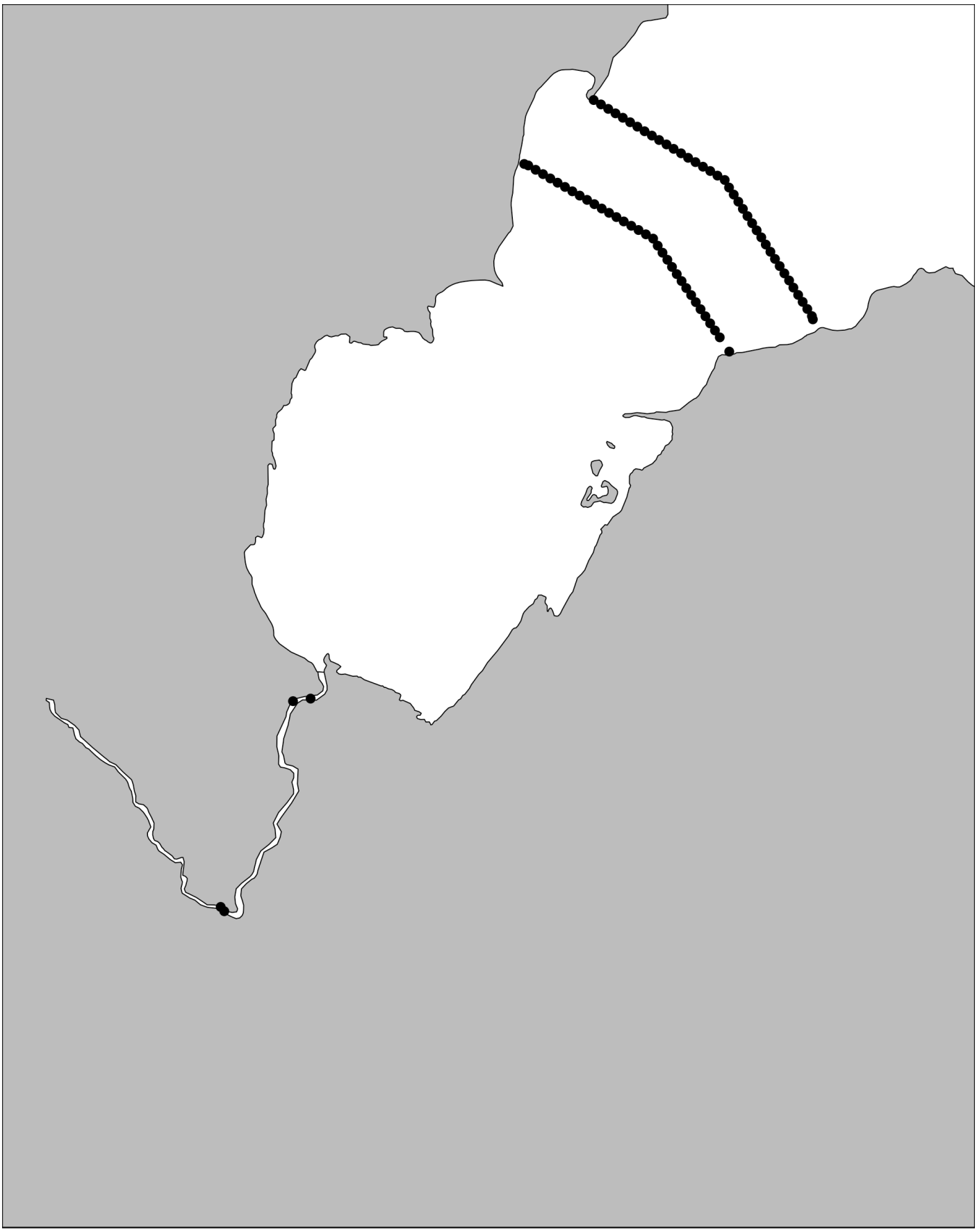
Let's change all interpolated positions to blue, actual detections to red and make all markers large squares.

```
# Add column of colors- blue for detections and red for interpolated
# detections.
pos$color <- ifelse(pos$record_type %in% 'detection', 'red', 'blue')
pos$marker_size <- 4
pos$marker <- 15

# Create output folder.
myDir <- paste0(getwd(),'/frames2')

make_frames(pos, recs=recs,
      out_dir=myDir,
      animate=FALSE,
      preview=TRUE,
      col=pos$color,
      cex=pos$marker_size,
      pch=pos$marker)
```



Specifying custom plot extents will allow us to zoom in on a particular area. Let's zoom in on Saginaw Bay and the Tittabawassee River.

```
myDir <- paste0(getwd(),'/frames3')
make_frames(pos, recs=recs, out_dir=myDir, background_ylim=c(43.1,44.3),
      background_xlim=c(-84.2,-83),
      animate=FALSE, preview=TRUE,
      col=pos$color, cex=pos$marker_size, pch=pos$marker)
```



2012-03-27 2012-07-07 2012-10-17 2013-01-26 2013-05-09

Create Custom Transition Layer: The make_transition() Function

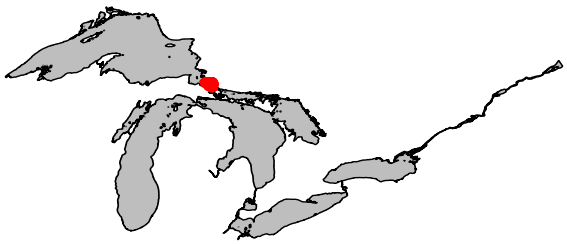
The ***make_frames()*** function only allows customization of fish plotting and the background extents. In the next example, we will interpolate positions for fish in a river system. The transition layer that we used above is sufficient for interpolating movements over large spatial distances (i.e. 100's of km) but lacks sufficient resolution to interpolate movements at smaller spatial scales (~10's km). In order to use the animation functions in ***glatos*** for these jobs, we will need to create our own transition layer. In the next example, we will use ***make_transition()*** to create a transition layer with sufficient resolution to interpolate and animate sea lamprey movements in the St. Marys river. The ***make_transition()*** function accepts a polygon shapefile and returns a transition layer where cell values of 0 are land and cell values of 1 are water. If a raster file exists for your study area, then a transition layer can be made by simply coding cells that represent land = 0 and cells representing water = 1. This is easily done using package ***raster*** (see raster package vignettes).

Before we create the transition layer, we will extract the St. Marys River system from the polygon shapefile that is included with the glatos package. Creating a raster layer of the entire Great Lakes basin at sufficient resolution to interpolate fish movements in the St. Marys River would result in massive file size and excessive computation times. Extracting the St. Marys River from the basin-wide shapefile results in reasonable file size and computation times.

The next code loads the sea lamprey detection data and polygon shapefile of the Great Lakes Basin and creates a plot of all locations where sea lamprey were detected in the St. Marys.

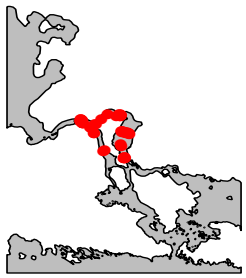
```
# Load in polygon shapefile of Great Lakes.
library(raster)

## Loading required package: sp

library(rgdal)

## rgdal: version: 1.2-16, (SVN revision 701)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 2.1.3, released 2017/20/01
##  Path to GDAL shared files:
/Library/Frameworks/R.framework/Versions/3.4/Resources/library/rgdal/gdal
##  GDAL binary built with GEOS: FALSE
##  Loaded PROJ.4 runtime: Rel. 4.9.3, 15 August 2016, [PJ_VERSION: 493]
##  Path to PROJ.4 shared files:
/Library/Frameworks/R.framework/Versions/3.4/Resources/library/rgdal/proj
##  Linking to sp version: 1.2-5

poly <- system.file('extdata', 'shoreline.zip', package='glatos')
poly <- unzip(poly, exdir=tempdir())
gl <- readOGR(poly[grepl('*.shp', poly)])

## OGR data source with driver: ESRI Shapefile
## Source:
"/var/folders/lb/cdwlg4dj5097yg9vfc7hbvl40000gn/T//RtmpYHHtqX/shoreline.shp",
layer: "shoreline"
## with 4 features
## It has 8 fields

# Load sea lamprey example data.
det_file <- system.file('extdata', 'lamprey_detections.csv',
            package='glatos')
det <- read_glatos_detections(det_file)
```

```
# Extract only unique positions to get an idea where fish were detected.
fish <- unique(det[, c('deploy_lat', 'deploy_long')])

# Plot fish locations.
plot(gl, col='grey')
points(x=fish$deploy_long, fish$deploy_lat, col='red', pch=16)
```



As you can see, we need to zoom in on the St. Marys in order to visualize the fish movements. The next code block uses spatial extents obtained from Google Earth to crop out the St. Marys River.

```
# Use raster::crop to create new shapefile of the St. Marys.
# Extents determined using google earth: extent(xmin, xmax, ymin, ymax)
st_mary <- crop(gl, extent(-84.7, -83.57, 45.90, 47))

## Loading required namespace: rgeos

# Take a look - much better!
plot(st_mary, col='grey')
points(x=fish$deploy_long, fish$deploy_lat, col='red', pch=16)
```

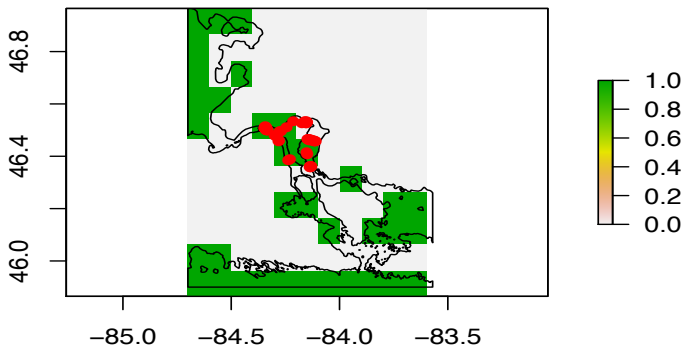


```
# We need to write out the cropped shapefile for the next step.
# This line writes the .shp in a folder named 'st_mary' with shapefile
```

```
# named 'st_mary' in the working directory.
writeOGR(obj=st_mary, dsn='st_mary', layer='st_mary',
     driver='ESRI Shapefile', overwrite_layer=TRUE)
```

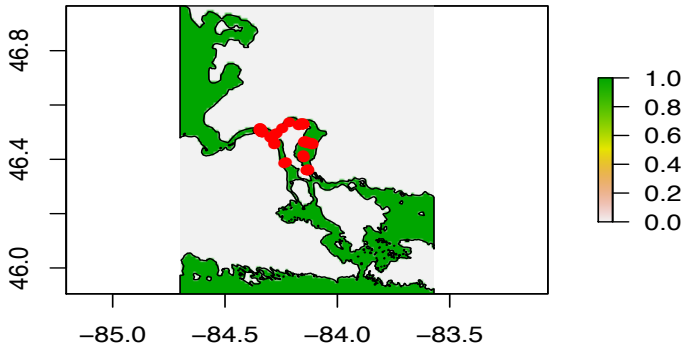
Next we use ***make_transition()*** to create a transition layer for non-linear interpolation. This function reads the shapefile that we extracted in the previous step and outputs a geotiff raster layer and a list object that contains the raster (named 'rast') and a transition layer (named 'transition'). The first attempt uses a pixel size of 0.1 degrees (x) by 0.1 degrees (y).

```
# Make transition layer and geotif from shapefile.
tran1 <- make_transition(in_file='st_mary/st_mary.shp',
             output='st_mary.tif',
             output_dir='st_mary',
             res=c(0.1, 0.1))

# Time to check by plotting detections, shoreline, and raster layer.
plot(tran1$rast)
plot(st_mary, add=TRUE)
points(x=fish$deploy_long, fish$deploy_lat, col='red', pch=16)
```



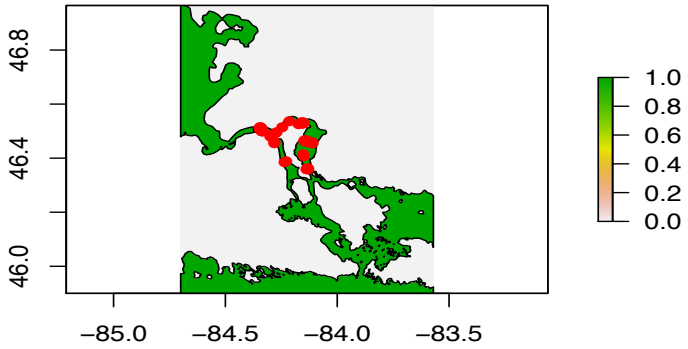
The figure shows that the pixel size is way too large to capture fish movements in the St. Marys. Notice many holes in coverage and receivers that aren't in water (green = water). We will increase resolution by an order of magnitude (0.01 x 0.01) and remake transition layer in the next code block.

```
# Remake transition layer at higher resolution.
tran2 <- make_transition(in_file='st_mary/st_mary.shp',
             output= 'st_mary1.tif',
             output_dir='st_mary',
             res=c(0.01, 0.01))

# Check layer.
plot(tran2$rast)
plot(st_mary, add=TRUE)
points(x=fish$deploy_long, fish$deploy_lat, col='red', pch=16)
```



The resolution of the second attempt definitely captures more of the spatial features of the St. Marys River and probably would work fine for interpolating. However, let's increase resolution one more time to 0.001 x 0.001 degrees. Notice how computation time increases with resolution.

```
# Remake transition layer at 0.001 x 0.001 degrees (cell size).
tran3 <- make_transition(in_file='st_mary/st_mary.shp',
             output= 'st_mary2.tif',
             output_dir='st_mary',
             res=c(0.001, 0.001))

plot(tran3$rast)
plot(st_mary, add=TRUE)
points(x=fish$deploy_long, fish$deploy_lat, col='red', pch=16)
```



The third attempt looks really good. Water = 1, land = 0 and there aren't a lot of jagged edges or holes. Determining the resolution necessary for a particular interpolation task depends on the study and requires a bit of trial and error. We will use the third transition layer to perform non-linear interpolation of the sea lamprey data.

Now we are ready to interpolate the sea lamprey movements. First, we will reload the detection file and run `interpolate_path()`. Notice we are using a `int_time_stamp` of two hours.

```
# Reload sea lamprey example data.
det_file <- system.file('extdata', 'lamprey_detections.csv',
            package='glatos')
det <- read_glatos_detections(det_file)
# Call the 'transition matrix' that we created in the previous step.
# Run non-linear interpolation on sea lamprey example data.
# Interpolating positions at 2-hour intervals.
pos <- interpolate_path(det, trans=tran3$transition, int_time_stamp=3600*2)

## starting linear interpolation

## finished linear interpolation

## starting non-linear interpolation

## finished non-linear interpolation
```

Now we are ready to load receivers, make frames, and animate. If you do not have `FFmpeg` setup on your computer, set `animate=FALSE` or supply a path to `FFmpeg` using the `ffmpeg` argument. We have created the full animation in this step and included the file in the supporting documents of this manual.

```
# Make animation.
# Re-load receiver location data.
rec_file <- system.file('extdata', 'sample_receivers.csv', package='glatos')
recs <- read_glatos_receivers(rec_file)
# Make frames.
myDir <- paste0(getwd(),'/frames4')
# Set background limits to same extents as used in 'crop' above.
make_frames(pos, recs=recs, background_ylim=c(46.2, 46.6),
      background_xlim=c(-84.57, -83.57),
      out_dir=myDir,
      animate=TRUE,
      preview=FALSE,
      ffmpeg=NA,
      tail_dur=0,
      pch=16,
      cex=3)
```

Last, lets try out `adjust_playback_time()` by slowing down the animation that we created in the previous step by a factor of two and changing the output format to '.wmv'. This function uses `FFmpeg`, so you will need to supply a path to `FFmpeg` if it is not installed on your system path.

```
# Slow the playback down.
adjust_playback_time(2,
                     input=file.path(getwd(), 'frames4',
                                     'animation.mp4'),
                     output_dir=file.path(getwd(), 'frames4'),
                     output='new.wmv',
                     ffmpeg=NA)
```

Foray into Simulation: Predicting Receiver Line Performance

The topics we've covered to this point all involve methods to summarize and visualize observed detection data. However, some of the most critical processes that affect telemetry systems, including decisions we make about system design or interpretation of data, are not observed. Simulations can be useful for exploring

the relative performance of system characteristics (e.g., receiver spacing, transmitter power, transmitter delay) or the potential consequences of unanticipated outcomes (e.g., lost receivers). In our final example, we will estimate, by simulation, the probability of detecting an acoustic-tagged fish on a receiver line, given constant fish velocity (ground speed), receiver spacing, number of receivers, and detection range curve.

An important consideration when analyzing telemetry data or developing a new telemetry project is to estimate the probability that a fish is detected when within range of a receiver. Successful detection of an acoustic tag signal by a receiver is determined by complex interactions between both abiotic and biotic variables in the environment. Detection range curves defining the probability of detecting a tag transmission at different tag-receiver distances are typically estimated using mobile or static range detection studies and are used to estimate how fluctuations in detection range (resulting from changes in the acoustic environment) may influence detection of fish. Although range detection studies are important for estimating the probability of detecting a tag transmission, they do not estimate the probability of detecting a fish that is moving through a line of multiple receivers.

Here, we will use the ***receiver_line_det_sim()*** function in the ***glatos*** package to determine the smallest spacing between receivers in a new receiver line that will ensure that all fish are detected by that line. The function ***receiver_line_det_sim()*** swims a virtual tagged fish perpendicularly through a virtual receiver line to estimate the probability of detecting the fish on that receiver line, given constant fish movement rate, receiver spacing and number of receivers for a specified detection range curve. Detection of each transmission on all receivers is simulated by draws from a Bernoulli distribution using the distance between the tag and each receiver in the line based on the user-provided range detection curve. A fish is only considered detected on the line if detected more than once on a given receiver in the simulation, as single detections would not pass a false detection filter. The ***receiver_line_det_sim()*** requires a function that specifies a range detection curve for the ***rngFun*** argument. The logistic curve is what we will use here but any other function that describes the shape of the detection range curve can be used as long as the function supplies a numeric vector of detection probabilities for an input vector of distances. Other examples of detection range curves are available in the help document for ***receiver_line_det_sim()***. The remaining arguments for ***receiver_line_det_sim()*** are: ***vel***, constant fish velocity; ***delayRng***, a numeric vector with min and max delay for the tag; ***burstDur***, duration of each coded burst; ***recSpc***, a numeric vector with distances in meters between receivers; ***maxDist***, max distance between fish and receiver; ***outerLim***, maximum distance simulated fish are allowed to pass to the left and right of the receiver line; ***nsim***, number of fish to simulate; ***showPlot***, a switch specifying whether a plot of fish paths is to be returned.

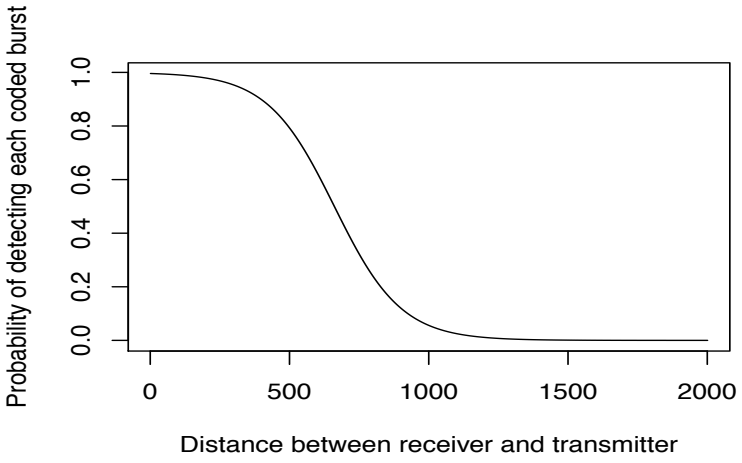
Let's assume that we have detection range data (for ***rngFun***) from a range test we conducted at the new receiver line location, fish swim speed (***vel***) from the literature, tag delay we want to use based on consideration of collision rates and tag life (for ***delayRng***), and the burst duration of the tag from the manufacturer (for ***burstDur***). We can use these parameters to estimate the probability that a fish will be detected on the receiver line, given those parameters, using the ***receiver_line_det_sim()*** function.

Now, let's setup and run a simulation. First, we need to define ***rngFun***. We will use a logistic regression to define the detection range curve in this example. ***dm*** is the distance in meters and ***b*** is the intercept and slope for the logistic curve.

```
# Define detection range function (to pass as rngFun) that returns detection
# probability for given distance assume logistic form of detection range
# curve where dm=distance in meters b=intercept and slope.
pdrf <- function(dm, b=c(5.5, -1/120)){
 p <- 1/(1+exp(-(b[1]+b[2]*dm)))
 return(p)
}
```

Before we run the simulation, let's preview the range curve created by ***pdrf***. If we pass a vector of distances from 0 to 2000 meters, the function returns detection probabilities for a logistic detection range curve.

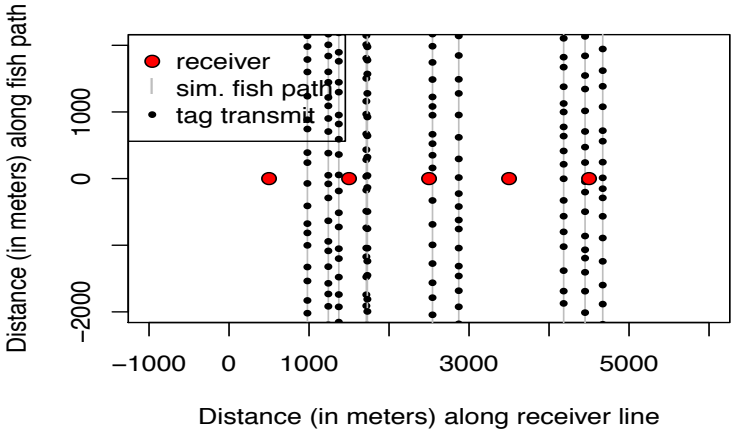
```
# Preview detection range curve.
plot(pdrf(0:2000),type='l',
  ylab='Probability of detecting each coded burst',
  xlab='Distance between receiver and transmitter')
```



Now that the range detection curve is set up, we can set up the simulation. ***pdrf*** is supplied to ***receiver_line_det_sim()*** as the ***rngFun*** argument. Now let's assume that we have five receivers for our new line, intended to span 5 km between an island and mainland shoreline.

Now, let's run the simulation with our five receivers spaced at 1000 m (***recSpc***) and the first and last receivers 500 m from shore (***outerLim***) so that fish are allowed to pass left and right of virtual line up to 500 m. First, we will run the function with ***showPlot=T*** and just ten virtual fish so we can see what it's doing. Then we will run it with 10000 fish (with ***showPlot=F***) to get a more robust result.

```
# Five receivers and allow fish to pass to left and right of line.
dp <- receiver_line_det_sim(rngFun=pdrf, recSpc=rep(1000,4),
    outerLim=c(500, 500), nsim=10, showPlot=T)
```



```
# The `receiver_line_det_sim()` estimates line detection probability for the
# specified combination of receiver line layout and fish transmitter
# specifications. The value returned is the portion of simulated fish that
# were detected at least twice passing the receiver line (i.e., they would
# pass a false detection filter).
dp

## [1] 1
```

It looks like all ten fish were detected. Now, let's repeat it with 10000 fish, but without the plot.

```
# Again but with 10000 fish and no plot.
dp <- receiver_line_det_sim(rngFun=pdrf, recSpc=rep(1000,4),
    outerLim=c(500, 500), nsim=10000)
dp

## [1] 0.997
```

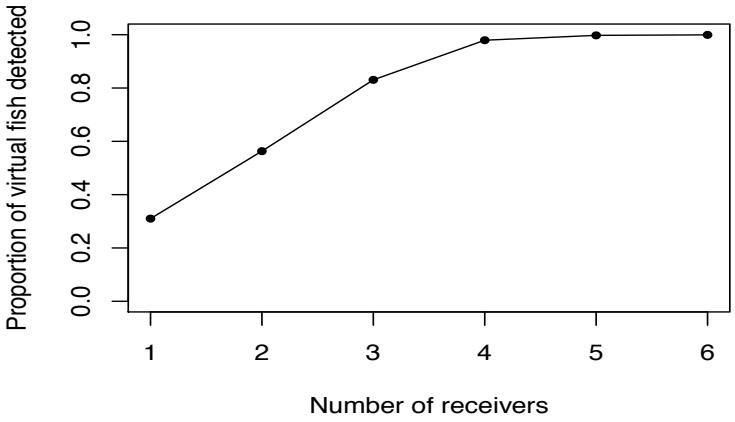
Results suggest that nearly all fish should be detected (99.7% of fish) under the conditions simulated. However, we might want to use fewer receivers if we could still detect all fish. So let's use the simulation to identify the smallest receiver spacing that would still detect nearly every fish. We'll simply loop through a vector of receiver spacing, get the detection probability estimate for each, and then use that to guide our decision.

```
# Preallocate a vector to store estimates.
dp <- rep(NA, 6)

# One receiver with 2500 m on each side.
dp[1] <- receiver_line_det_sim(rngFun=pdrf, recSpc=0,
 outerLim=c(2500, 2500), nsim=10000)
dp[1]

## [1] 0.3101

# Two receivers with 2500 m between and 1250 m on each end.
dp[2] <- receiver_line_det_sim(rngFun=pdrf, recSpc=2500,
 outerLim=c(1250, 1250), nsim=10000)
dp[2]

## [1] 0.5632

# Three receivers with 1667 m in between and 833 m on each end.
dp[3] <- receiver_line_det_sim(rngFun=pdrf, recSpc=rep(1667,2),
 outerLim=c(833, 833), nsim=10000)
dp[3]

## [1] 0.8307

# Four receivers with 1250 m in between and 625 m on each end.
dp[4] <- receiver_line_det_sim(rngFun=pdrf, recSpc=rep(1250,3),
 outerLim=c(625, 625), nsim=10000)
dp[4]

## [1] 0.9793

# Five receivers with 1000 m in between and 500 m on each end.
dp[5] <- receiver_line_det_sim(rngFun=pdrf, recSpc=rep(1000,4),
```

```
 outerLim=c(500, 500), nsim=10000)
dp[5]

## [1] 0.9976

# Six receivers with 833 m in between and 417 m on each end.
dp[6] <- receiver_line_det_sim(rngFun=pdrf, recSpc=rep(833,5),
 outerLim=c(433, 433), nsim=10000)
dp[6]

## [1] 0.9994
```

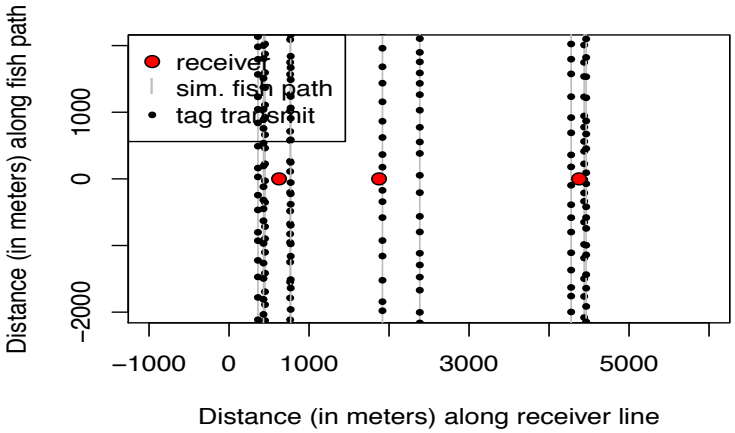
We've seen the results, but let's plot them too.

```
# Plot number of receivers vs proportion of virtual fish detected.
plot(1:6, dp, type='o',ylim=c(0,1),
 xlab='Number of receivers',
 ylab='Proportion of virtual fish detected',
 pch=20)
```



These results suggest that we could still detect 98% of the fish with only four receivers instead of five, so we might choose to deploy only four receivers spaced 1250 m. But before we make our final decision, let's consider what might happen if we went with the four-receiver spacing but lost a receiver during our study, so that our line ended up having a hole in it. Again, we will simulate just ten fish with a plot to see what this looks like and then we'll run the simulation with 10000 virtual fish.

```
# Three receivers now with a gap of 2500 m between two and 625 m on ends.
# 10 fish with plot.
dph <- receiver_line_det_sim(rngFun=pdrf, recSpc=c(1250,2500),
 outerLim=c(625, 625), nsim=10, showPlot=T)
```

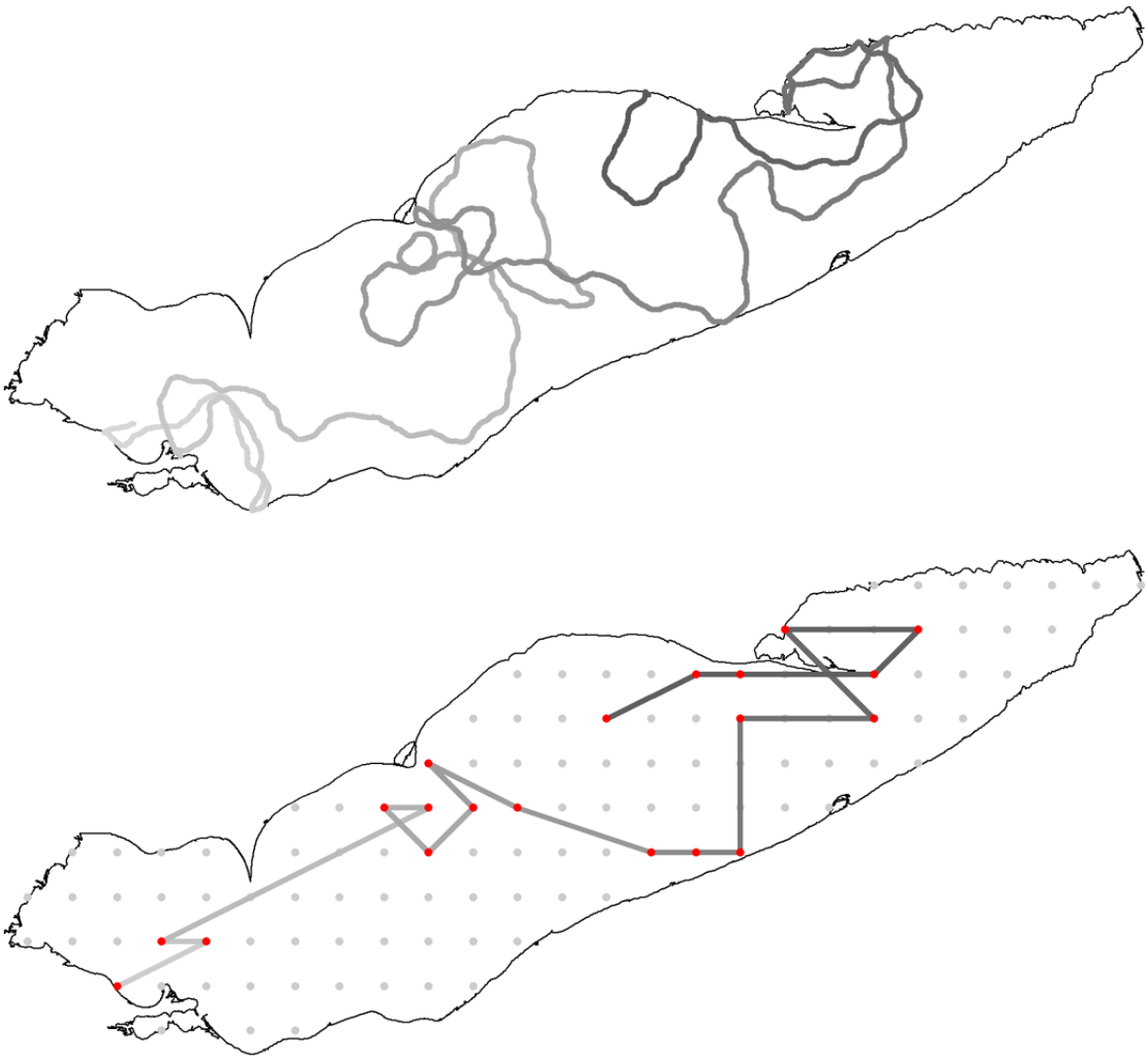



```
# Repeat with 10000 fish, no plot.
dph <- receiver_line_det_sim(rngFun=pdrf, recSpc=c(1250,2500),
 outerLim=c(625, 625), nsim=10000)
dph

## [1] 0.7771
```

Estimated detection probability dropped to about 78% of fish after losing a receiver, so our final decision might depend on how likely we are to lose a receiver during the study.

Similar to the way we've used `receiver_line_det_sim()` here, we could simulate detection probabilities over a range of fish swim speed, tag delay, or even detection range curves to understand how results might be sensitive to our assumptions or decisions regarding those parameters.

In addition to the one-dimensional receiver line simulation presented above, the `glatos` package contains a set of functions that can be used to simulate fish movement and detection on receivers in a virtual two-dimensional arena. For example, the image below shows simulated detection data from a virtual fish in Lake Erie with a grid of 116 receivers (15 km spacing).



We won't cover this function in detail here, but will briefly explain the components. First, `crw_in_polygon()` can be used to simulate fish movement paths as a random walk inside a polygon. Second, `transmit_along_path()` can be used to simulate tag signal transmissions along each path based on constant movement velocity, transmitter delay range, and duration of signal. Third, `detect_transmissions()` can be used to simulate detection of transmitter signals in a receiver network based on detection range curve, location of transmitter, and locations of receivers. Collectively, these functions can be used to explore, compare, and contrast theoretical performance of a wide range of transmitter and receiver network designs.

Simulation can be a powerful tool for decision making and the `glatos` package contains several simulation-based functions (e.g., `calc_collision_prob()`). However, it's important to keep in mind that simulations are a simplification of reality and that results are only as good as the assumptions and variables going in (i.e., 'garbage in, garbage out'). Fortunately, many processes that influence telemetry systems are fairly well understood. Our approach to these types of simulations is generally to be cautious: make conservative assumptions to yield conservative inferences. In this way, we can hope for the best but rest easy knowing we have 'planned for the worst'.

Appendix A: Skill-Building Exercises

Data Types, Objects, and Subsetting

Vectors

1. Create a single vector containing the values 1 through 24, every 5th value between 25 and 95, and every hundredth value between 100 and 1000.

2. Use R's built-in constant `letters` to create a vector containing every other letter in the alphabet. `letters` is a 26-element vector of characters. Enter `?letters` into the R console to view the help page for the `letters` vector.

3. Copy and run the following vector assignment code into R and then use square bracket subsetting to rearrange the vector so that the months are in proper order: `vect_a <- c('April', 'September', 'February', 'March', 'October', 'December', 'May', 'November', 'July', 'January', 'August', 'June')`. Hint: a vector of index values can be used to subset multiple elements from a vector.

4. Copy and run the following vector assignment code into R and then use the R base function `sort()` to rearrange the vector in descending order: `vect_b <- c(5, 3, 9, 4, 8, 10, 6, 2, 7, 1)`. Enter `?sort` into the R console to view the help page for the function.

Matrices

1. Create a vector containing the values 1 through 100 and then coerce the vector into a matrix called `matrix_1` with 5 columns and 20 rows. Use R's base `rev()` function (enter `?rev` into the R console to view the help file) to fill the matrix in reverse order, so that the value 100 is in the top left corner (i.e., [1, 1]) of the matrix and the value 1 is in the bottom right corner (i.e., [20, 5]) of the matrix.

2. Use R's base `sum()` function (enter `?sum` into the R console to view the help file) to calculate the sum of all values in `matrix_1` created above.

3. Make every other value in `matrix_1` negative. Hint: Recall the example above where we added two matrices together; the matrices added together on an element by element bases. When performing multiplication between two objects of different length, R will recycle the elements in the shorter object. In other words, R will cycle through each element in the smaller object and then start back again at the beginning.

Lists

1. Create a named list called `list_1` with two elements. The first list element will be a 5 x 5 matrix called `evens` containing the even numbers between 1 and 50. The second list element will be a 5 x 5 matrix called `odds` containing the odd numbers between 1 and 50. Next, extract the two matrix elements from `list_1` using named list notation and add them together to create a third 5 x 5 matrix with elements equal to the sum of the two elements with the same indexed values from the `evens` and `odds` matrices.

2. Most of the statistical functions available in R save results in named lists. In the code that follows, we will conduct a two-sample t-test on two vectors of randomly-generated numbers and store the results in an object called `stats_list`. The `rnorm()` function draws `n` number of values randomly from a normal distribution with a mean equal to `mean` and standard deviation equal to `sd`. In addition to the

code required to perform the t-test, we have included a line that calls the **`set.seed()`**, which is used to fix the random number generation so that results are reproduced every time the code is run. Paste the following code into the R console and run it to perform the t-test. Next, extract the **`statistic`**, **`df`**, and **`p.value`** from the results and combine them into a single vector. Hint: Use the **`str()`** function to view the structure of the t.test results. You will see that the results are stored as a named list.

```
# Set seed to make random number generation reproducible.
set.seed(45)
# Run t-test on randomly-generated and store results in an object named
# 'stat_list'.
stat_list <- t.test(rnorm(n=20, mean=10, sd=2), rnorm(n=20, mean=15, sd=2))
```

Data Frames

R contains an example dataset called **`mtcars`**, which is a data frame containing various specifications for a series of motor vehicles. We will use this dataset to practice subsetting and sorting data frames. The contents of the **`mtcars`** data frame can be viewed by entering **`mtcars`** into the R console. The help file for the dataset can be seen by entering **`?mtcars`** into the R console. Take a look at the **`mtcars`** data frame. You might notice that it looks different than other data frames you have seen previously in the above examples - there is no column header above the vehicle names. The reason for this is that the vehicle names are row names, rather than being an actual column in the data frame. These values can be manipulated using the **`rownames()`**, but it is not very common to see row names used in data frames, so we will leave it to the reader to investigate this issue further at their leisure.

Perform the following manipulations on the **`mtcars`** data frame:

1. Extract the first 10 rows of **`mtcars`**. Hint: You may wish to look up the **`head()`** function.
2. Reorder the columns in **`mtcars`** so that columns **`am`** and **`gears`** appear between columns **`cyl`** and **`disp`**.
3. Reorder the rows in the **`mtcars`** data frame so that they appear in order of decreasing mileage (**`mpg`**).
4. Add a new column, **`kpl`**, to **`mtcars`** with the mileage (**`mpg`**) converted to kilometers per liter. Hint: 1 mile = 1.61 km and 1 US gallon = 3.79 liters.

Logic Statements Exercises

Again, we will use the built-in **`mtcars`** data frame to practice subsetting with logic statements. Perform the following logical subsetting operations:

1. Subset the **`mtcars`** data frame to display vehicles with 8 cylinders.
2. Subset the **`mtcars`** data frame to display vehicles that get better than 25 mpg.
3. Subset the **`mtcars`** data frame to display vehicles with automatic transmission AND greater than 200 gross horsepower.
4. Subset the **`mtcars`** data frame to display vehicles with automatic transmission AND either 4 OR 6 cylinders.

Conditional Statements Exercises

1. Write a script to simulate rolling a dice. Print the string ‘evens’ if the dice rolls an even number and ‘odds’ if the dice rolls an odd number.

2. Add a fuel efficiency rating column called `fer` to the `mtcars` data frame. Cars that get better than or equal to 20 mpg will receive a fuel efficiency rating of ‘good’ and those that get less than 20 mpg will receive a fuel efficiency rating of ‘poor’.

Repeated Operations Exercise

1. Write a script that flips a coin 100 times and keeps track of the number of times the coin lands on each possible outcome. Hint: You will need to create counter variables to keep track of the two outcomes.

String Manipulation Exercises

Paste the following code into the R console and run to create a data frame containing date and time strings that we will manipulate:

```
date_time_data <- data.frame(date=c('06/30/2015', '07/21/2016' ,'03/07/2014',
                                    '05/01/2014', '11/14/2016', '01/30/2015'),
                             time=c('01:45:20', '13:54:45', '11:23:07',
                                    '22:10:12', '07:51:02', '10:13:42'),
                             stringsAsFactors=FALSE)`.
```

1. Convert the `date` strings in `date_time_data` to strings with format ‘2015-05-30’. There are a couple of different ways this could be done. You may need to use several different functions to complete this task. Hint: Think about this manipulation as a series of smaller processes, and do them in sequence.

2. Combine the newly formatted dates from exercise 1 with their corresponding times to create a column of datetime stamps called `timestamp`. The new timestamps should have format ‘2015-05-30 19:05:32’.

Appendix B: Answers to Skill-Building Exercises

Data Types, Objects, and Subsetting

Vectors

1. Create a single vector containing the values 1 through 24, every 5th value between 25 and 95, and every hundredth value between 100 and 1000.

```
c(1:24, seq(from=25, to=95, by=5), seq(from=100, to=1000, by=100))

##  [1]    1    2    3    4    5    6    7    8    9   10   11   12   13   14
## [15]   15   16   17   18   19   20   21   22   23   24   25   30   35   40
## [29]   45   50   55   60   65   70   75   80   85   90   95  100  200  300
## [43]  400  500  600  700  800  900 1000
```

2. Use R's built-in constant `letters` to create a vector containing every other letter in the alphabet. `letters` is a 26-element vector of characters. Enter `?letters` into the R console to view the help page for the `letters` vector.

```
letters[seq(from=1, to=26, by=2)]

##  [1] "a" "c" "e" "g" "i" "k" "m" "o" "q" "s" "u" "w" "y"
```

3. Copy and run the following vector assignment code into R and then use square bracket subsetting to rearrange the vector so that the months are in proper order: `vect_a <- c('April', 'September', 'February', 'March', 'October', 'December', 'May', 'November', 'July', 'January', 'August', 'June')`. Hint: a vector of index values can be used to subset multiple elements from a vector.

```
# Create the vector of character strings.
vect_a <- c('April', 'September', 'February', 'March', 'October', 'December',
      'May', 'November', 'July', 'January', 'August', 'June')

# Re-order the vector so the months are in the proper order using square
# bracket subsetting.
vect_a[c(10, 3, 4, 1, 7, 12, 9, 11, 2, 5, 8, 6)]

##  [1] "January"   "February"  "March"     "April"     "May"
##  [6] "June"      "July"      "August"    "September" "October"
## [11] "November"  "December"
```

4. Copy and run the following vector assignment code into R and then use the R base function `sort()` to rearrange the vector in descending order: `vect_b <- c(5, 3, 9, 4, 8, 10, 6, 2, 7, 1)`. Enter `?sort` into the R console to view the help page for the function.

```
# Create the vectior.
vect_b <- c(5, 3, 9, 4, 8, 10, 6, 2, 7, 1)
# Sort the vector in decreasing order.
sort(vect_b, decreasing=TRUE)

##  [1] 10  9  8  7  6  5  4  3  2  1
```

Matrices

1. Create a vector containing the values 1 through 100 and then coerce the vector into a matrix called ***matrix_1*** with 5 columns and 20 rows. Use R's base ***rev()*** function (enter ***?rev*** into the R console to view the help file) to fill the matrix in reverse order, so that the value 100 is in the top left corner (i.e., [1, 1]) of the matrix and the value 1 is in the bottom right corner (i.e., [20, 5]) of the matrix.

```
# Create a vector containing the values 1 through 100.
v1 <- c(1:100)
# Populate the matrix with V1 in reverse order.
matrix_1 <- matrix(rev(v1), ncol=5)
# View first 6 rows.
head(matrix_1)

##      [,1] [,2] [,3] [,4] [,5]
## [1,]  100   80   60   40   20
## [2,]   99   79   59   39   19
## [3,]   98   78   58   38   18
## [4,]   97   77   57   37   17
## [5,]   96   76   56   36   16
## [6,]   95   75   55   35   15

# Note the same operation could have been done in a single line.
head(matrix(rev(1:100), ncol=5))

##      [,1] [,2] [,3] [,4] [,5]
## [1,]  100   80   60   40   20
## [2,]   99   79   59   39   19
## [3,]   98   78   58   38   18
## [4,]   97   77   57   37   17
## [5,]   96   76   56   36   16
## [6,]   95   75   55   35   15
```

2. Use R's base ***sum()*** function (enter ***?sum*** into the R console to view the help file) to calculate the sum of all values in ***matrix_1*** created above.

```
sum(matrix_1)

## [1] 5050
```

3. Make every other value in ***matrix_1*** negative. Hint: Recall the example above where we added two matrices together; the matrices added together on an element by element bases. When performing multiplication between two objects of different length, R will recycle the elements in the shorter object. In other words, R will cycle through each element in the smaller object and then start back again at the beginning.

```
# Make every other value in matrix_1 by multiplying it by vector c(1, -1).
# Because the vector is shorter than the matrix, R cycles through the vector
# alternating between multiplying by 1 and multiplying by -1.
head(matrix_1 * c(1, -1))

##      [,1] [,2] [,3] [,4] [,5]
## [1,]  100   80   60   40   20
## [2,]  -99  -79  -59  -39  -19
## [3,]   98   78   58   38   18
## [4,]  -97  -77  -57  -37  -17
## [5,]   96   76   56   36   16
## [6,]  -95  -75  -55  -35  -15
```

```
# We also could have created a second matrix of the same dimensions and
# multiplied the two together.
head(matrix_1 * matrix(rep(c(1, -1), 50), ncol=5))

##      [,1] [,2] [,3] [,4] [,5]
## [1,]  100   80   60   40   20
## [2,]  -99  -79  -59  -39  -19
## [3,]   98   78   58   38   18
## [4,]  -97  -77  -57  -37  -17
## [5,]   96   76   56   36   16
## [6,]  -95  -75  -55  -35  -15
```

Lists

1. Create a named list called `list_1` with two elements. The first list element will be a 5 x 5 matrix called `evens` containing the even numbers between 1 and 50. The second list element will be a 5 x 5 matrix called `odds` containing the odd numbers between 1 and 50. Next, extract the two matrix elements from `list_1` using named list notation and add them together to create a third 5 x 5 matrix with elements equal to the sum of the two elements with the same indexed values from the `evens` and `odds` matrices.

```
# Create a names list object containing two matrices, one named 'evens' and
# the other named 'odds'.
list_1 <- list(evens=matrix(seq(from=2, to=50, by=2), ncol=5),
               odds=matrix(seq(from=1, to=50, by=2), ncol=5))
# View the list.
list_1

## $evens
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2   12   22   32   42
## [2,]    4   14   24   34   44
## [3,]    6   16   26   36   46
## [4,]    8   18   28   38   48
## [5,]   10   20   30   40   50
##
## $odds
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1   11   21   31   41
## [2,]    3   13   23   33   43
## [3,]    5   15   25   35   45
## [4,]    7   17   27   37   47
## [5,]    9   19   29   39   49

# Subset out the individual matrices from list_1 and add them together.
list_1$evens + list_1$odds

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    3   23   43   63   83
## [2,]    7   27   47   67   87
## [3,]   11   31   51   71   91
## [4,]   15   35   55   75   95
## [5,]   19   39   59   79   99
```

2. Most of the statistical functions available in R save results in named lists. In the code that follows, we will conduct a two-sample t-test on two vectors of randomly-generation numbers and store the results in an object called `stats_list`. The `rnorm()` function draws `n` number of values randomly from a

normal distribution with a mean equal to **mean** and standard deviation equal to **sd**. In addition to the code required to perform the t-test, we have included a line that calls the **set.seed()**, which is used to fix the random number generation so that results are reproduced every time the code is run. Paste the following code into the R console and run it to perform the t-test. Next, extract the **statistic**, **df**, and **p.value** from the results and combine them into a single vector. Hint: Use the **str()** function to view the structure of the t.test results. You will see that the results are stored as a named list.

```
# Set seed to make random number generation reproducible.
set.seed(45)
# Run t-test on randomly-generated and store results in an object named
# `stat_list`.
stat_list <- t.test(rnorm(n=20, mean=10, sd=2), rnorm(n=20, mean=15, sd=2))
# View the structure of the results object
str(stat_list)

## List of 9
##  $ statistic  : Named num -6.88
##   ..- attr(*, "names")= chr "t"
##  $ parameter  : Named num 36.8
##   ..- attr(*, "names")= chr "df"
##  $ p.value    : num 4.25e-08
##  $ conf.int   : atomic [1:2] -6.32 -3.44
##   ..- attr(*, "conf.level")= num 0.95
##  $ estimate   : Named num [1:2] 10.2 15.1
##   ..- attr(*, "names")= chr [1:2] "mean of x" "mean of y"
##  $ null.value : Named num 0
##   ..- attr(*, "names")= chr "difference in means"
##  $ alternative: chr "two.sided"
##  $ method     : chr "Welch Two Sample t-test"
##  $ data.name  : chr "rnorm(n = 20, mean = 10, sd = 2) and rnorm(n = 20, mean
= 15, sd = 2)"
##  - attr(*, "class")= chr "htest"

c(stat_list$statistic, stat_list$parameter, stat_list$p.value)

##             t            df
## -6.876073e+00  3.682540e+01  4.250306e-08
```

Data Frames

R contains an example dataset called **mtcars**, which is a data frame containing various specifications for a series of motor vehicles. We will use this dataset to practice subsetting and sorting data frames. The contents of the **mtcars** data frame can be viewed by entering **mtcars** into the R console. The help file for the dataset can be seen by entering **?mtcars** into the R console. Take a look at the **mtcars** data frame. You might notice that it looks different than other data frames you have seen previously in the above examples - there is no column header above the vehicle names. The reason for this is that the vehicle names are row names, rather than being an actual column in the data frame. These values can be manipulated using the **rownames()**, but it is not very common to see row names used in data frames, so we will leave it to the reader to investigate this issue further at their leisure.

Perform the following manipulations on the **mtcars** data frame:

1. Extract the first 10 rows of **mtcars**. Hint: You may wish to look up the **head()** function.

```
# Subset out first ten rows using square bracket subsetting.
mtcars[1:10,]
```

```
##                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4

# Could also have used the `head()` function, which returns the first n rows
# of a data frame.
head(mtcars, 10)

##                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

2. Reorder the columns in **mtcars** so that columns **am** and **gears** appear between columns **cyl** and **disp**.

```
# Reorder columns. `head()` limits the output to 6 rows.
head(mtcars[, c(1, 2, 9, 10, 3:8, 11)])

##                    mpg cyl am gear disp  hp drat    wt  qsec vs carb
## Mazda RX4         21.0   6  1    4  160 110 3.90 2.620 16.46  0    4
## Mazda RX4 Wag     21.0   6  1    4  160 110 3.90 2.875 17.02  0    4
## Datsun 710        22.8   4  1    4  108  93 3.85 2.320 18.61  1    1
## Hornet 4 Drive    21.4   6  0    3  258 110 3.08 3.215 19.44  1    1
## Hornet Sportabout 18.7   8  0    3  360 175 3.15 3.440 17.02  0    2
## Valiant           18.1   6  0    3  225 105 2.76 3.460 20.22  1    1
```

3. Reorder the rows in the **mtcars** data frame so that they appear in order of decreasing mileage (**mpg**).

```
# Reorder rows. `head()` limits the output to 6 rows.
head(mtcars[order(mtcars$mpg, decreasing=TRUE),])

##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
```

4. Add a new column, **kpl**, to **mtcars** with the mileage (**mpg**) converted to kilometers per liter. Hint: 1 mile = 1.61 km and 1 US gallon = 3.79 liters.

```
mtcars$kpl <- mtcars$mpg*1.61/3.79
# View first 6 rows of `mtcars`.
head(mtcars)

##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
##                       kpl
## Mazda RX4         8.920844
## Mazda RX4 Wag     8.920844
## Datsun 710        9.685488
## Hornet 4 Drive    9.090765
## Hornet Sportabout 7.943799
## Valiant           7.688918
```

Logic Statements Exercises

Again, we will use the built-in ***mtcars*** data frame to practice subsetting with logic statements. Perform the following logical subsetting operations:

1. Subset the ***mtcars*** data frame to display vehicles with 8 cylinders.

```
# Display vehicles with 8 cylinders. `head()` limits the output to 6 rows.
head(mtcars[mtcars$cyl == 8,])

##                     mpg cyl  disp  hp drat   wt  qsec vs am gear carb
## Hornet Sportabout  18.7   8 360.0 175 3.15 3.44 17.02  0  0    3    2
## Duster 360         14.3   8 360.0 245 3.21 3.57 15.84  0  0    3    4
## Merc 450SE         16.4   8 275.8 180 3.07 4.07 17.40  0  0    3    3
## Merc 450SL         17.3   8 275.8 180 3.07 3.73 17.60  0  0    3    3
## Merc 450SLC        15.2   8 275.8 180 3.07 3.78 18.00  0  0    3    3
## Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.25 17.98  0  0    3    4
##                         kpl
## Hornet Sportabout  7.943799
## Duster 360         6.074670
## Merc 450SE         6.966755
## Merc 450SL         7.349077
## Merc 450SLC        6.456992
## Cadillac Fleetwood 4.417942
```

2. Subset the ***mtcars*** data frame to display vehicles that get better than 25 mpg.

```
# Display vehicles that get better than 25 mpg. `head()` limits the output to
# 6 rows.
head(mtcars[mtcars$mpg > 25,])

##                  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Fiat 128        32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic     30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Fiat X1-9       27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2   26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa    30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
```

```
##                       kpl
## Fiat 128         13.76359
## Honda Civic      12.91398
## Toyota Corolla   14.40079
## Fiat X1-9        11.59710
## Porsche 914-2    11.04485
## Lotus Europa     12.91398
```

3. Subset the ***mtcars*** data frame to display vehicles with automatic transmission AND greater than 200 gross horsepower.

```
# Display vehicles with automatic transmission AND greater than 200 gross
# horsepower. `head()` limits the output to 6 rows.
head(mtcars[mtcars$am == 0 & mtcars$hp > 200,])

##                      mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
## Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
##                          kpl
## Duster 360          6.074670
## Cadillac Fleetwood  4.417942
## Lincoln Continental 4.417942
## Chrysler Imperial   6.244591
## Camaro Z28          5.649868
```

4. Subset the ***mtcars*** data frame to display vehicles with automatic transmission AND either 4 OR 6 cylinders.

```
# Using & and |.
mtcars[mtcars$am == 0 & (mtcars$cyl == 4 | mtcars$cyl == 6),]

##                  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Hornet 4 Drive  21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Valiant         18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Merc 240D       24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230        22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280        19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C       17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Toyota Corona   21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
##                        kpl
## Hornet 4 Drive   9.090765
## Valiant          7.688918
## Merc 240D       10.365172
## Merc 230         9.685488
## Merc 280         8.156201
## Merc 280C        7.561478
## Toyota Corona    9.133245

# Could also have use the %in% operator.
mtcars[mtcars$am == 0 & mtcars$cyl %in% c(4, 6),]

##                  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Hornet 4 Drive  21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Valiant         18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Merc 240D       24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
```

```
## Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
##                       kpl
## Hornet 4 Drive  9.090765
## Valiant         7.688918
## Merc 240D      10.365172
## Merc 230        9.685488
## Merc 280        8.156201
## Merc 280C       7.561478
## Toyota Corona   9.133245
```

Conditional Statements Exercises

1. Write a script to simulate rolling a dice. Print the string 'evens' if the dice rolls an even number and 'odds' if the dice rolls an odd number.

```
# Using the %in% operator to determine if roll is within a vector containing
# the evens outcomes.
roll <- sample(c(1:6), 1)
roll

## [1] 2

if(roll %in% c(2, 4, 6)){
 print('evens')
}else{
 print('odds')
}

## [1] "evens"

# Another way to determine generally whether a number is even or odd is to
# use modulus. The modulus operator (`%%`) divides the number on the left of
# the operator by the number on the right of the operator and returns the
# remainder. For example, 5 can be divided by 2 twice, with a remainder of 1.
# If a number is even, then 2 should divide into it with no remainder. We can
# therefore use this method to test more generally whether our roll was even
# or odd.
if(roll%%2 == 0){
 print('evens')
}else{
 print('odds')
}

## [1] "evens"
```

2. Add a fuel efficiency rating column called `fer` to the `mtcars` data frame. Cars that get better than or equal to 20 mpg will receive a fuel efficiency rating of 'good' and those that get less than 20 mpg will receive a fuel efficiency rating of 'poor'.

```
# Add a fuel efficiency rating column called `fer` to the `mtcars` data frame.
# `head()` limits the output to 6 rows.
mtcars$fer <- ifelse(mtcars$mpg >= 20, 'good', 'poor')
# View first 6 rows.
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
##                        kpl  fer
## Mazda RX4         8.920844 good
## Mazda RX4 Wag     8.920844 good
## Datsun 710        9.685488 good
## Hornet 4 Drive    9.090765 good
## Hornet Sportabout 7.943799 poor
## Valiant           7.688918 poor
```

Repeated Operations Exercise

1. Write a script that flips a coin 100 times and keeps track of the number of times the coin lands on each possible outcome. Hint: You will need to create counter variables to keep track of the two outcomes.

```
# Create counter variables.
num_heads <- 0
num_tails <- 0
# Flip a coin 100 times, and add 1 to whatever counter the coin lands on.
for(i in 1:100){
 flip <- sample(c('heads', 'tails'), 1)
 if(flip == 'heads'){
  num_heads <- num_heads + 1
 }else{
  num_tails <- num_tails + 1
 }
}
# Number of heads flips.
num_heads

## [1] 44

# Number of tails flips.
num_tails

## [1] 56
```

Note that we could have done this exercise without the loop because `sample()` let's us take more than one sample at a time. Therefore, we could have taken 100 samples in the original call to `sample()` and then used square bracket subsetting and the `length()` function to determine how many of each outcomes occurred.

```
# Sample from our head/tails vector 100 times. The `replace=TRUE` argument
# tells R to put our selection back in the pot each time so it can be
# selected again (if we did not set this to TRUE we would have run out of
# values to sample on the 3rd attempt).
coin_flips <- sample(c('heads', 'tails'), 100, replace=TRUE)
# View first 20 elements of the vector.
coin_flips
```

```
##   [1] "heads" "heads" "heads" "heads" "tails" "tails" "tails" "tails"
##   [9] "heads" "tails" "tails" "tails" "heads" "heads" "tails" "tails"
##  [17] "heads" "heads" "heads" "heads" "heads" "tails" "heads" "tails"
##  [25] "heads" "tails" "heads" "heads" "heads" "tails" "heads" "heads"
##  [33] "tails" "heads" "heads" "tails" "tails" "heads" "heads" "tails"
##  [41] "tails" "heads" "tails" "tails" "heads" "tails" "heads" "heads"
##  [49] "heads" "tails" "heads" "tails" "tails" "tails" "tails" "heads"
##  [57] "tails" "tails" "tails" "tails" "tails" "tails" "heads" "heads"
##  [65] "tails" "tails" "tails" "heads" "heads" "tails" "tails" "tails"
##  [73] "tails" "heads" "heads" "heads" "heads" "heads" "heads" "tails"
##  [81] "tails" "tails" "heads" "heads" "heads" "tails" "tails" "tails"
##  [89] "tails" "tails" "heads" "heads" "heads" "heads" "heads" "tails"
##  [97] "tails" "heads" "tails" "tails"

# Number of heads flips.
length(coin_flips[coin_flips == 'heads'])

## [1] 49

# Number of tails flips.
length(coin_flips[coin_flips == 'tails'])

## [1] 51
```

String Manipulation Exercises

Paste the following code into the R console and run to create a data frame containing date and time strings that we will manipulate: `date_time_data <- data.frame(date=c('06/30/2015', '07/21/2016' ,'03/07/2014', '05/01/2014', '11/14/2016', '01/30/2015'), time=c('01:45:20', '13:54:45', '11:23:07', '22:10:12', '07:51:02', '10:13:42'), stringsAsFactors=FALSE)`.

```
# Create the data frame.
date_time_data <- data.frame(date=c('06/30/2015', '07/21/2016' ,'03/07/2014',
                  '05/01/2014', '11/14/2016', '01/30/2015'),
                time=c('01:45:20', '13:54:45', '11:23:07',
                  '22:10:12', '07:51:02', '10:13:42'),
                stringsAsFactors=FALSE)
```

1. Convert the `date` strings in `date_time_date` to strings with format '2015-05-30'. There are a couple of different ways this could be done. You may need to use several different functions to complete this task. Hint: Think about this manipulation as a series of smaller processes, and do them in sequence.

```
# Using `substr()` and `paste()`.
date_time_data$date_substr <- paste(substr(date_time_data$date, 7, 10),
                  substr(date_time_data$date, 1, 2),
                  substr(date_time_data$date, 4, 5),
                  sep='-')

# Using `strsplit()`, `sapply()` and `paste()`. Place results in new column
# called 'date2'.
split_time <- strsplit(date_time_data$date, '/')
date_time_data$date_ssaply <- sapply(split_time, FUN=function(x)
 paste(x[[3]], x[[1]], x[[2]], sep='-'))

date_time_data
```

```
##          date     time date_substr date_ssaply
## 1 06/30/2015 01:45:20  2015-06-30  2015-06-30
## 2 07/21/2016 13:54:45  2016-07-21  2016-07-21
## 3 03/07/2014 11:23:07  2014-03-07  2014-03-07
## 4 05/01/2014 22:10:12  2014-05-01  2014-05-01
## 5 11/14/2016 07:51:02  2016-11-14  2016-11-14
## 6 01/30/2015 10:13:42  2015-01-30  2015-01-30
```

2. Combine the newly formatted dates from exercise 1 with their corresponding times to create a column of datetime stamps called ***timestamp***. The new timestamps should have format '2015-05-30 19:05:32'.

```
date_time_data$timestamp <- paste(date_time_data$date_substr,
                 date_time_data$time)
date_time_data

##          date     time date_substr date_ssaply           timestamp
## 1 06/30/2015 01:45:20  2015-06-30  2015-06-30 2015-06-30 01:45:20
## 2 07/21/2016 13:54:45  2016-07-21  2016-07-21 2016-07-21 13:54:45
## 3 03/07/2014 11:23:07  2014-03-07  2014-03-07 2014-03-07 11:23:07
## 4 05/01/2014 22:10:12  2014-05-01  2014-05-01 2014-05-01 22:10:12
## 5 11/14/2016 07:51:02  2016-11-14  2016-11-14 2016-11-14 07:51:02
## 6 01/30/2015 10:13:42  2015-01-30  2015-01-30 2015-01-30 10:13:42
```

Appendix C: Useful Functions for Visualizing and Analyzing Acoustic Telemetry Data

Below are some functions that we have found useful for visualizing and analyzing acoustic telemetry. The list is the exhaustive, and we will continue to add to the list as we encounter more useful functions.

Plotting Functions

Function	Package	Description
plot()	graphics	A generic function for plotting objects - plot properties based on object properties
barplot()	graphics	Creates a bar plot with vertical or horizontal bars
boxplot()	graphics	Creates a box-and-whisker plot
image()	graphics	Creates a grid of colored rectangles (i.e., a raster)
par()	graphics	Used to set or query graphical parameters
layout()	graphics	Divides the plotting device into rows and columns for making multi-panel plots
text()	graphics	Adds text to a plot
mtext()	graphics	Adds text to the margins of a plot
axis()	graphics	Add custom axis to the plot
legend()	graphics	Adds legends to plots
box()	graphics	Draws a box (border) around a plot
points()	graphics	Add points to an existing plot
lines()	graphics	Add line a scatter data to existing plot
abline()	graphics	Adds a straight line to an existing plot
polygon()	graphics	Draws a polygon on an existing plot
arrows()	graphics	Draw arrows between points - useful for making error bars
symbols()	graphics	Draws circles, squares, rectangles, stars, thermometers, and boxplots on a plot
colorRampPalette()	grDevices	Interpolates a set of given colors and creates a new color palette
colors()	grDevices	Returns the built-in color names that R accepts

Data Summary and Manipulation Functions

Function	Package	Description
summary()	base	Generic function the produces summaries specific to the input object type
fivenum()	stats	Returns Tukey's five number summary (min, lower-hinge, median, upper-hinge, max)
mean()	base	Returns the mean of elements in an object
sd()	stats	Returns the standard deviation of elements in an object
median()	stats	Returns the median value of elements in an object
sum()	base	Returns the sum of elements in a vector
range()	base	Returns a vector containing the min and max value of elements in an object

aggregate()	stats	Computes summary statistics of data subsets
ddply()	plyr	Apply functions to split data frames
apply()	base	Apply functions by the margins (rows or columns) of an array or matrix
sapply()	base	Apply functions over elements in a vector or list
findInterval()	base	Find interval of a value, given a vector of non-decreasing breakpoints
seq	base	Generates a regular sequence of values between two numbers
approx()	base	Linear interpolation
expand.grid()	base	Create a data frame from all combinations of factors
merge()	base	Merge two data frames
which()	base	Determine which indexes evaluate to TRUE
with()	base	Evaluate expression in environment constructed from data
match()	base	Match vector of positions in first vector with second vector
row.names()	base	Get and set row names for data frames
t()	base	Transpose matrix
order()	base	Returns the index values of a vector sorted in ascending or descending order
sort()	base	Sorts the values in a vector in ascending or descending order
paste()	base	Concatenate vectors after converting to character

GIS Functions

Function	Package	Description
readOGR()	rgdal	Reads OGR data layer into spatial vector object - read ESRI shapefile layers into R
readGDAL()	rgdal	Reads GDAL grid maps (i.e., geotiff) and returns a spatial object
project()	rgdal	Interface to PROJ.4 for projecting geographic position data
spTransform()	sp	Transforms projected data to a new coordinate reference system (e.g., lat/lon to UTM)
distGeo()	geosphere	Calculates the shortest distance between two geographic locations
bearing()	argosfilter	Calculates the bearing in degrees between two geographic locations
point.in.polygon()	sp	Determines whether a point is within a polygon
pnt.in.poly()	SDMTools	Determines whether a point is within a polygon or not within a polygon c(0,1)

Appendix D: Built-in Plotting Colors by Name: colors()

white	darkcyan	goldenrod	gray67	grey32	honeydew4	lightsteelblue1	orchid4	sienna1
aliceblue	darkgoldenrod	goldenrod1	gray68	grey33	hotpink	lightsteelblue2	palegoldenrod	sienna2
antiquewhite	darkgoldenrod1	goldenrod2	gray69	grey34	hotpink1	lightsteelblue3	palegreen	sienna3
antiquewhite1	darkgoldenrod2	goldenrod3	gray70	grey35	hotpink2	lightsteelblue4	palegreen1	sienna4
antiquewhite2	darkgoldenrod3	goldenrod4	gray71	grey36	hotpink3	lightyellow	palegreen2	skyblue
antiquewhite3	darkgoldenrod4	gray	gray72	grey37	hotpink4	lightyellow1	palegreen3	skyblue1
antiquewhite4	darkgray	gray0	gray73	grey38	indianred	lightyellow2	palegreen4	skyblue2
aquamarine	darkgreen	gray1	gray74	grey39	indianred1	lightyellow3	paleturquoise	skyblue3
aquamarine1	darkgrey	gray2	gray75	grey40	indianred2	lightyellow4	paleturquoise1	skyblue4
aquamarine2	darkkhaki	gray3	gray76	grey41	indianred3	limegreen	paleturquoise2	slateblue
aquamarine3	darkmagenta	gray4	gray77	grey42	indianred4	linen	paleturquoise3	slateblue1
aquamarine4	darkolivegreen	gray5	gray78	grey43	ivory	magenta	paleturquoise4	slateblue2
azure	darkolivegreen1	gray6	gray79	grey44	ivory1	magenta1	palevioletred	slateblue3
azure1	darkolivegreen2	gray7	gray80	grey45	ivory2	magenta2	palevioletred1	slateblue4
azure2	darkolivegreen3	gray8	gray81	grey46	ivory3	magenta3	palevioletred2	slategray
azure3	darkolivegreen4	gray9	gray82	grey47	ivory4	magenta4	palevioletred3	slategray1
azure4	darkorange	gray10	gray83	grey48	khaki	maroon	palevioletred4	slategray2
beige	darkorange1	gray11	gray84	grey49	khaki1	maroon1	papayawhip	slategray3
bisque	darkorange2	gray12	gray85	grey50	khaki2	maroon2	peachpuff	slategray4
bisque1	darkorange3	gray13	gray86	grey51	khaki3	maroon3	peachpuff1	slategrey
bisque2	darkorange4	gray14	gray87	grey52	khaki4	maroon4	peachpuff2	snow
bisque3	darkorchid	gray15	gray88	grey53	lavender	mediumaquamarin	peachpuff3	snow1
bisque4	darkorchid1	gray16	gray89	grey54	lavenderblush	mediumblue	peachpuff4	snow2
black	darkorchid2	gray17	gray90	grey55	lavenderblush1	mediumorchid	peru	snow3
blanchedalmond	darkorchid3	gray18	gray91	grey56	lavenderblush2	mediumorchid1	pink	snow4
blue	darkorchid4	gray19	gray92	grey57	lavenderblush3	mediumorchid2	pink1	springgreen
blue1	darkred	gray20	gray93	grey58	lavenderblush4	mediumorchid3	pink2	springgreen1
blue2	darksalmon	gray21	gray94	grey59	lawngreen	mediumorchid4	pink3	springgreen2
blue3	darkseagreen	gray22	gray95	grey60	lemonchiffon	mediumpurple	pink4	springgreen3
blue4	darkseagreen1	gray23	gray96	grey61	lemonchiffon1	mediumpurple1	plum	springgreen4
blueviolet	darkseagreen2	gray24	gray97	grey62	lemonchiffon2	mediumpurple2	plum1	steelblue
brown	darkseagreen3	gray25	gray98	grey63	lemonchiffon3	mediumpurple3	plum2	steelblue1
brown1	darkseagreen4	gray26	gray99	grey64	lemonchiffon4	mediumpurple4	plum3	steelblue2
brown2	darkslateblue	gray27	gray100	grey65	lightblue	mediumseagreen	plum4	steelblue3
brown3	darkslategray	gray28	green	grey66	lightblue1	mediumslateblue	powderblue	steelblue4
brown4	darkslategray1	gray29	green1	grey67	lightblue2	mediumspringgree	purple	tan
burlywood	darkslategray2	gray30	green2	grey68	lightblue3	mediumturquoise	purple1	tan1
burlywood1	darkslategray3	gray31	green3	grey69	lightblue4	mediumvioletred	purple2	tan2
burlywood2	darkslategray4	gray32	green4	grey70	lightcoral	midnightblue	purple3	tan3
burlywood3	darkslategrey	gray33	greenyellow	grey71	lightcyan	mintcream	purple4	tan4
burlywood4	darkturquoise	gray34	grey	grey72	lightcyan1	mistyrose	red	thistle
cadetblue	darkviolet	gray35	grey0	grey73	lightcyan2	mistyrose1	red1	thistle1
cadetblue1	deeppink	gray36	grey1	grey74	lightcyan3	mistyrose2	red2	thistle2
cadetblue2	deeppink1	gray37	grey2	grey75	lightcyan4	mistyrose3	red3	thistle3
cadetblue3	deeppink2	gray38	grey3	grey76	lightgoldenrod	mistyrose4	red4	thistle4
cadetblue4	deeppink3	gray39	grey4	grey77	lightgoldenrod1	moccasin	rosybrown	tomato
chartreuse	deeppink4	gray40	grey5	grey78	lightgoldenrod2	navajowhite	rosybrown1	tomato1
chartreuse1	deepskyblue	gray41	grey6	grey79	lightgoldenrod3	navajowhite1	rosybrown2	tomato2
chartreuse2	deepskyblue1	gray42	grey7	grey80	lightgoldenrod4	navajowhite2	rosybrown3	tomato3
chartreuse3	deepskyblue2	gray43	grey8	grey81	ghtgoldenrodyello	navajowhite3	rosybrown4	tomato4
chartreuse4	deepskyblue3	gray44	grey9	grey82	lightgray	navajowhite4	royalblue	turquoise
chocolate	deepskyblue4	gray45	grey10	grey83	lightgreen	navy	royalblue1	turquoise1
chocolate1	dimgray	gray46	grey11	grey84	lightgrey	navyblue	royalblue2	turquoise2
chocolate2	dimgrey	gray47	grey12	grey85	lightpink	oldlace	royalblue3	turquoise3
chocolate3	dodgerblue	gray48	grey13	grey86	lightpink1	olivedrab	royalblue4	turquoise4
chocolate4	dodgerblue1	gray49	grey14	grey87	lightpink2	olivedrab1	saddlebrown	violet
coral	dodgerblue2	gray50	grey15	grey88	lightpink3	olivedrab2	salmon	violetred
coral1	dodgerblue3	gray51	grey16	grey89	lightpink4	olivedrab3	salmon1	violetred1
coral2	dodgerblue4	gray52	grey17	grey90	lightsalmon	olivedrab4	salmon2	violetred2
coral3	firebrick	gray53	grey18	grey91	lightsalmon1	orange	salmon3	violetred3
coral4	firebrick1	gray54	grey19	grey92	lightsalmon2	orange1	salmon4	violetred4
cornflowerblue	firebrick2	gray55	grey20	grey93	lightsalmon3	orange2	sandybrown	wheat
cornsilk	firebrick3	gray56	grey21	grey94	lightsalmon4	orange3	seagreen	wheat1
cornsilk1	firebrick4	gray57	grey22	grey95	lightseagreen	orange4	seagreen1	wheat2
cornsilk2	floralwhite	gray58	grey23	grey96	lightskyblue	orangered	seagreen2	wheat3
cornsilk3	forestgreen	gray59	grey24	grey97	lightskyblue1	orangered1	seagreen3	wheat4
cornsilk4	gainsboro	gray60	grey25	grey98	lightskyblue2	orangered2	seagreen4	whitesmoke
cyan	ghostwhite	gray61	grey26	grey99	lightskyblue3	orangered3	seashell	yellow
cyan1	gold	gray62	grey27	grey100	lightskyblue4	orangered4	seashell1	yellow1
cyan2	gold1	gray63	grey28	honeydew	lightslateblue	orchid	seashell2	yellow2
cyan3	gold2	gray64	grey29	honeydew1	lightslategray	orchid1	seashell3	yellow3
cyan4	gold3	gray65	grey30	honeydew2	lightslategrey	orchid2	seashell4	yellow4
darkblue	gold4	gray66	grey31	honeydew3	lightsteelblue	orchid3	sienna	yellowgreen