Practical R for MBM

Chuck Frost

7/10/2020

Practical R for MBM field data manipulation and analysis

The purpose of this ongoing short course is to familiarize MBM personnel with the concepts of tidy data, basic data manipulation, and basic programming techniques as implemented in R. It is expected that participants have a working understanding of R and RStudio. We will not go into depth on other types of programming, specifically statistical programming and computer programming. The topics covered here will foster appropriate treatment of MBM data throughout the data life cycle and lead to cleaner and more useful data in the future.

Topics

1. R, RStudio, packages, help, intro to functions

2. Loading and saving, data types and structures

3. Tidy Data

4. Visualization

5. Working with dates
6. Creating functions, conditional statements, looping

1.0.1 What is R?

An open-source programming environment that serves as:

1. A giant calculator

2. An interface for data analysis and visualization

3. An interface for data manipulation and file handling

4. An interface for a simple and efficient programming language (S, that became R)

R isn't the "method." For example, "We used R to estimate population size." yuck

1.0.2 What is RStudio?

An integrated development environment (IDE) for R.

Provides tools, menus, help, and easy access to the best parts of R.

1.1.1 Packages

Base R automatically provides you with many common functions.

```r
mean(c(1,2,3))
```

```
## [1] 2
```

```r
sum(c(1,2,3))
```

```
## [1] 6
```

```r
var(c(1,2,3))
```

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

Other useful functions exist thanks to R users. Collections of related functions get wrapped into packages.

If you're looking for something NOT in base R, you need to install the related package.

The CRAN provides R community tested and approved packages. Check the packages tab in RStudio for some you already have installed but (probably) not loaded. Just click them to load. If you know what package you need, there are 2 easy ways to install it. The first is by clicking the install button in the packages tab. Search the CRAN for the package you want and install it. The second is:

```r
install.packages("PackageName")
```

Don't forget to load the package after installation, either by checking the box under packages, or by calling:

```r
library(PackageName)
```

Note that in install.packages() you are searching a string in quotes, "PackageName" while in library() you are calling a package object without quotes, PackageName.

1.1.2 Installing from GitHub

If a package isn't uploaded to CRAN, it can still be made available through GitHub. Packages installed from GitHub require an intermediate step:

```r
install.packages("devtools")

library(devtools)

install_github("USFWS/AKaerial", ref = "development")

library(AKaerial)
```

This code does 4 things:
1. Install devtools package
2. Load devtools package

3. Call install_github (from devtools), look for the GitHub account "USFWS" and repository "AKaerial" and load the package in the "development" branch
4. Load AKaerial package

Installing and loading AKaerial may take a while since it includes dependencies. Dependencies are packages that are specified within another package that must be included for that package to operate appropriately. Akaerial "depends" on several other packages and will check if you have these installed.

1.1.3 Functions

Once you have a package loaded, you can access functions written and included in that package. Typing the name of a function will spit out the code that makes up the function. This can be messy for a long or complicated function. Typing ?FunctionName will access the help file for a given function.

```
AdjustCounts

?AdjustCounts
```

This can be extremely helpful (if the help file is!) when troubleshooting why your code isn't working as you want it to. Note that the top of the help file also tells you FunctionName {PackageName} in case you can't figure out where your function is coming from. This could be the case if you have dozens of packages loaded or have borrowed someone else's code to help run your analysis. Don't underestimate the power of internet searches to find if something already exists to help you do your thing!

We will talk about functions in more depth later, but for now, the general idea is that functions take arguments, run processes, and (usually) return products saved as R objects.

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

avg = mean(numbers)

avg
```

```
## [1] 2
```

In this example, numbers is initiated as a vector of 3 integers: 1,2,3. We then run the function mean with numbers as its argument, and save the result as the object avg.

It can be helpful to think of functions as recipes and arguments as ingredients.

```
Mix = function(what.to.mix){
  dough = sum(what.to.mix)
  return(dough)
  }

Bake = function(what.to.bake, time.to.bake){

  cookies = what.to.bake * time.to.bake

  return(cookies)

}
```

```r
my.ingredients = c("sugar", "eggs", "butter", "flour")

my.dough = Mix(my.ingredients)

my.cookies = Bake(my.dough, 10)
```

2.1.0 Loading Files

There are many ways to read data into R. We will go over the most common way data is stored and used in R; as a data frame. A data frame is a 2-dimensional array (table) where each column represents a variable (category) and each row represents a unique observation. Let's start with creating your own data frame. This might be useful if you have a small data set.

```r
small.data = data.frame(
  name = c("Laura", "Zak", "Chuck"),
  size = c("small", "medium", "large"),
  tenure = c(6, "drifter",9)
    )
```

We can now check the structure of the data frame. This is a great first QA/QC check! If we know that tenure should be numeric, the structure should confirm it.

```r
str(small.data)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ name  : Factor w/ 3 levels "Chuck","Laura",..: 2 3 1
##  $ size  : Factor w/ 3 levels "large","medium",..: 3 2 1
##  $ tenure: Factor w/ 3 levels "6","9","drifter": 1 3 2
```

Notice how all 3 variables are shown as Factor, which is incorrect. Factors, in general are tough to work with. If we try to change the value here (use $ to grab a column and brackets to select a row [ ], or use only brackets to select [row, column]), it throws an error.

```r
small.data$tenure[2] = 3
```

```
## Warning in `[<-.factor`(`*tmp*`, 2, value = structure(c(1L, NA, 2L), .Label
## = c("6", : invalid factor level, NA generated
```

```r
small.data[2,3] = 3
```

```
## Warning in `[<-.factor`(`*tmp*`, iseq, value = 3): invalid factor level, NA
## generated
```

```r
levels(small.data$tenure)
```

```
## [1] "6"       "9"       "drifter"
```

Since R couldn't determine the structure of our tenure column, it defaulted to calling it a Factor, which can take any value. The unique set of values gets set as the levels of that factor. If you try to define a value outside of that set, you will get an error that results in NA. This is (almost) never what you want. So we can fix this in 2 ways:

First, we can use the "as." functions in base R to force R to evaluate something as a different data type. In this case, we can't just let R evaluate the factor as numeric since it will then take a numeric representation of that particular level of a factor in the set of all levels of that factor. What a mess. We have to first convert the factor to a character string using as.character, then to a numeric using as.numeric. This is called wrapping functions, when you pass as an argument to a new function the output of another function without saving the intermediate result. In the outer-most function call to as.numeric, anything that R can't represent as numeric will get NA. This is already messy and hard to trace changes to your data!

```
small.data$tenure=as.numeric(as.character(small.data$tenure))

str(small.data)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ name  : Factor w/ 3 levels "Chuck","Laura",..: 2 3 1
##  $ size  : Factor w/ 3 levels "large","medium",..: 3 2 1
##  $ tenure: num  6 NA 9
```

Or, since we are scripting our analysis, we can change it in our original code chunk and just re-run. Score 1 for a nice scripted workflow…but imagine doing this through thousands or hundreds of thousands of rows of data.

```
small.data = data.frame(
  name = c("Laura", "Zak", "Chuck"),
  size = c("small", "medium", "large"),
  tenure = c(6,3,9)
    )
```

While we are at it, lets just purge the factors from our data entirely. It is easy to make something a factor later, if we ever see the need. To do this, we just pass the argument stringsAsFactors = FALSE to our data frame function. Note that we have to add a comma after the declaration of our final column tenure.

```
small.data = data.frame(
  name = c("Laura", "Zak", "Chuck"),
  size = c("small", "medium", "large"),
  tenure = c(6,3,9),
  stringsAsFactors = FALSE
    )

str(small.data)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ name  : chr  "Laura" "Zak" "Chuck"
##  $ size  : chr  "small" "medium" "large"
##  $ tenure: num  6 3 9
```

Now we are clean…maybe. Our structure says our name and size columns are chr (character) and our tenure correctly shows (num) numeric. Now we want add a new column (greatness) using the values in an existing column. We can both create and define the values of the new column in one step.

```r
small.data$greatness = small.data$tenure^3

str(small.data)
```

```
## 'data.frame':    3 obs. of  4 variables:
##  $ name     : chr  "Laura" "Zak" "Chuck"
##  $ size     : chr  "small" "medium" "large"
##  $ tenure   : num  6 3 9
##  $ greatness: num  216 27 729
```

```r
small.data
```

```
##    name   size tenure greatness
## 1 Laura  small      6       216
## 2   Zak medium      3        27
## 3 Chuck  large      9       729
```

Now we have our 4 columns and the structure even gives us what it can display of the values. Now what if we want to add another row of data? The function rbind (row bind) makes it easy. As long as our new row is in the same order and type as the columns of our data frame, we can simply call:

```r
small.data = rbind(small.data, c("Hannah", "small", 1, 1))

str(small.data)
```

```
## 'data.frame':    4 obs. of  4 variables:
##  $ name     : chr  "Laura" "Zak" "Chuck" "Hannah"
##  $ size     : chr  "small" "medium" "large" "small"
##  $ tenure   : chr  "6" "3" "9" "1"
##  $ greatness: chr  "216" "27" "729" "1"
```

```r
small.data
```

```
##     name   size tenure greatness
## 1  Laura  small      6       216
## 2    Zak medium      3        27
## 3  Chuck  large      9       729
## 4 Hannah  small      1         1
```

As you can imagine, the same thing can be done for columns using cbind. Let's add another one:

```r
small.data = cbind(small.data, interests=c("Bedazzling", "K-Pop", "Couponing", "Oranges"))

small.data
```

```
##     name   size tenure greatness  interests
## 1  Laura  small      6       216 Bedazzling
## 2    Zak medium      3        27      K-Pop
## 3  Chuck  large      9       729  Couponing
## 4 Hannah  small      1         1    Oranges
```

2.2.0 Reading in a data file

If you have a data file that exists (hopefully) as .csv or .txt, base R has a function that will read your data into a data frame for you. We will demo this with a .csv of fake aerial survey data. Remember that R works best (and sometimes only!) with tidy data. Your file should at least be rectangular (no loose columns, no long or dangling rows) and preferably will have column headings, though those can also be defined later. The function read.table will be the most generic and robust method, but there is also read.csv that saves a little time since you are predefining the formatting.

Plain text format is the simplest way to store text. One step up are delimited files. The most common types of delimited files and tab- and comma-delimited. Comma-delimited files are commonly saved as .csv (comma separated values). R does not care how your files are delimited, you just have to know the delimiting character(s) and pass them as arguments in the function call.

```r
my.data = read.table("SomeData.txt", header = TRUE, sep = " ")
```

This tells R to find the file SomeData.txt (it will default to your working directory, but you can path it anywhere). It will open it, read the header row into the data frame my.data, then scan the file for spaces (sep = " ") and add entries between spaces to the values in each row. What could possibly go wrong with space-delimiting? Or even comma-delimiting?

To illustrate further concepts below we will be using an artificial data set I modified for this demo. It is located on the GitHub repository for this short course. To read it in, we will use read.csv, which tells R to expect a comma-delimited file. Oh, and we should also get rid of those nasty factors for now.

```r
aerial=read.csv(
  "https://raw.githubusercontent.com/cfrost3/MBM_R_Short_Course/master/BLSC_2018_RawObs_Fake.csv",
                header=TRUE,
                stringsAsFactors=FALSE)

str(aerial)
```

```
## 'data.frame':    624 obs. of  16 variables:
##  $ Year      : int  2018 2018 2018 2018 2018 2018 2018 2018 2018 2018 ...
##  $ Month     : chr  "6" "6" "6" "6" ...
##  $ Day       : int  15 15 15 15 15 15 15 15 15 15 ...
##  $ Seat      : chr  "LF" "LF" "LF" "LF" ...
##  $ Observer  : chr  "FAKE" "FAKE" "FAKE" "FAKE" ...
##  $ Strata    : chr  NA NA NA NA ...
##  $ Transect  : chr  "19" "19" "19" "19" ...
##  $ Segment   : logi  NA NA NA NA NA NA ...
##  $ Flight_Dir: chr  "88" "88" "89" "89" ...
##  $ Wind_Dir  : chr  "45" "45" "45" "45" ...
##  $ Wind_Vel  : chr  "24" "24" "24" "24" ...
##  $ Lat       : num  61.1 61.1 61.2 61.2 61.2 ...
##  $ Long      : num  -166 -165 -165 -165 -165 ...
##  $ Species   : chr  "START" "SCAU4" "LTDUF4" "BLSC4" ...
##  $ Num       : chr  "19" "1" "1" "1" ...
##  $ Obs_Type  : chr  "open" "single" "single" "pair" ...
```

Hey! We have real (fake) data! Anything stand out immediately in the structure of the file?

2.3.1 Data Types

R has 4 basic data types you will use (6 in total, actually, including raw and complex):
1. Integer (1, 2, 3, 4)
2. Numeric (2, 4, 2.4, 45.48)
3. Character ("cat", "DOG", "15d", "and so on")
4. Logical (TRUE, FALSE)

2.3.2 Data Structures

Depending on how you combine data types, you can end up with one of several data structures:
1. Vector - a collection of all one data type
2. Matrix - multidimensional collection of vectors
3. Factor - nomical set of unique values and a vector of integer indices
4. List - open format to contain data structure elements
5. Data frame - our most common, discussed above

There are many other data structures in R, including user-defined and package-defined structures. We will mostly stick to the common ones.

2.3.3 Useful Data Structure Functions

Base R comes with many quick ways to assess your data structures. We will go into more depth later, but for now, explore commands such as:

```
a = c(1,2,3,4,10.1)

length(a)  #how long is the vector?
```

```
## [1] 5
```

```
mean(a)  #mean value
```

```
## [1] 4.02
```

```
var(a)  #variance
```

```
## [1] 12.802
```

```
typeof(a)  #what data type?  (double means floating decimal numeric)
```

```
## [1] "double"
```

```
is.na(a)  #any NA or missing values?
```

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

```r
summary(a)  #some basic summarizing statistics
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    2.00    3.00    4.02    4.00   10.10
```

```r
max(a)  #maximum value
```

```
## [1] 10.1
```

```r
min(a)  #minimum value
```

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

```r
dim(small.data)  #what are the dimensions of a structure?
```

```
## [1] 4 5
```

```r
class(small.data)  #what class of structure is it?
```

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

```r
names(small.data)  #what are the names of the objects that make up the structure?
```

```
## [1] "name"      "size"      "tenure"    "greatness" "interests"
```

```r
unique(small.data$size)  #what are all of the unique values in a range?
```

```
## [1] "small"  "medium" "large"
```

3.0.0 Tidy Data

Tidy data refers to a data set that satisfies 3 conditions:
1. Every column is 1 variable
2. Every row is 1 observation
3. Every cell is 1 value

In R, tidy data lends itself seamlessly to the vectorized functions we discussed above (and thousands of others). In addition to summarizing functions, such as length(), mean(), max(), min(), etc., plot functions in R work sometimes exclusively and other times much cleaner and more efficiently with tidy data.

It is important to note here that quality controlled data (typos removed, missing data correctly recorded, etc.) and tidy data are 2 separate and important characteristics.

Messy data (data that is not tidy) is often the result of database design by a data collector, rather than an analyst or data manager. This is not meant to place blame. A data collector has one primary purpose; translate the raw observations in the field into some sort of tabular archive. That archive will generally, then, take the form and function that was easiest and fastest for the collector to enter the data (a very reasonable result since it is often preferrable to get data collected and archived quickly). A data analyst similarly can have one primary purpose; to analyze data. While the analyst would also like to do things relatively as quickly as possible, messy data can make analysis lengthy, difficult, costly, and sometimes impossible. In MBM, for example, we have spent easily over 90% of our time cleaning data instead of on analysis (and the general consensus among similar analysts across all fields is 75-80% of time on cleaning).

3.1.0 Common Problems

There are 2 common problems with data that we run into frequently in MBM:
1. Using values as column names
2. Multiple variables in one column

3.1.1 Values as Column Names

Consider the following data frame:

```
messy1 = data.frame(Species = rep(c("COGO", "BOOW"), each = 3),
                    Box=c(1:6),
                    Visit1 = c(5,6,6,4,5,4),
                    Visit2 = c(5,6,5,4,5,7),
                    Visit3 = c(0,6,5,4,5,0),
                    Visit4 = c(NA,0,0,1,0,NA),
                    Visit5 = c(NA,NA,NA,0,NA,NA),
                    stringsAsFactors = FALSE
                    )


messy1
```

```
##   Species Box Visit1 Visit2 Visit3 Visit4 Visit5
## 1    COGO   1      5      5      0     NA     NA
## 2    COGO   2      6      6      6      0     NA
## 3    COGO   3      6      5      5      0     NA
## 4    BOOW   4      4      4      4      1      0
## 5    BOOW   5      5      5      5      0     NA
## 6    BOOW   6      4      7      0     NA     NA
```

Imagine they are visits to nests of 2 different species and the counts of the eggs that were found on each visit. Why is this messy? Looks clean enough. Easy enough for the data collector. Notice how columns 3-7 are labeled? This is a prime example of using a value for a variable (in this case, visit number) as the column header. It now becomes awkward for those nests that don't share the same number of visits. As an analyst, how can I script a quick summary of the number of times a nest was visited? Maybe by counting the number of cells that have a value greater than or equal to 0 in a row?

```
#give me the total of row 1 values greater than or equal to 0
v = sum(messy1[1,]>0)

v
```

```
## [1] NA
```

Ouch. Oh yeah, the NA values. A sum containing an NA is always NA. So let's remove those.

```
#give me the total of row 1 values greater than or equal to 0, and skip the NAs
v = sum(messy1[1,]>0, na.rm = TRUE)

v
```

```
## [1] 4
```

Ok, looks like it worked. But not when we spot check it. We forgot that Box will (probably) always be greater than or equal to 0. So let's tack on a modifier.

```r
#give the total of row 1 values >= to 0, skip the NAs, subtract 1 for Box
v = sum(messy1[1,]>0, na.rm = TRUE)-1

v
```

```
## [1] 3
```

Well, we now correctly show 3 visits. But what could possibly go wrong with this in a larger script? Or imagine in an ACTUAL data set, where we have dozens of columns. This is the kind of thing that would drive an analyst bonkers AND would be a nightmare to fix after it has been done for years on a large data set. We won't even attempt to reshape this using base R commands. Lucky for us, we have the tidyverse package.

Tidyverse is actually a collection of packages developed by data scientists, for data scientists. When you install tidyverse, you get them all. It streamlines importing, cleaning, tidying, visualizing, analyzing, and reporting any size data set. Most (all) of the functions and functionality available in the tidyverse are also possible in base R, but often with significantly more (and generally more confusing) code. We can't possibly describe them all, but we will hit some of the main functions.

First, let's install the tidyverse package. This may take a while depending on how many of the packages you already have. We will also reload our fake aerial survey data to play with.

```r
install.packages("tidyverse")
```

```r
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 3.6.3

## Warning: package 'ggplot2' was built under R version 3.6.3

## Warning: package 'tibble' was built under R version 3.6.3

## Warning: package 'tidyr' was built under R version 3.6.3

## Warning: package 'readr' was built under R version 3.6.1

## Warning: package 'purrr' was built under R version 3.6.3

## Warning: package 'dplyr' was built under R version 3.6.3

## Warning: package 'forcats' was built under R version 3.6.3
```

```r
aerial=read.csv(
  "https://raw.githubusercontent.com/cfrost3/MBM_R_Short_Course/master/BLSC_2018_RawObs_Fake.csv",
                header=TRUE,
                stringsAsFactors=FALSE)
```

If you recall, we used the $ operator to reference a column in a data frame. Tidyverse provides a cleaner syntax with the select function.

```r
#grab Transect, Species, and Num only from aerial
aerial.subset = select(aerial, Transect, Species, Num)

head(aerial.subset)  #only show me the first few rows
```

```
##   Transect Species Num
## 1       19   START  19
## 2       19   SCAU4   1
## 3       19  LTDUF4   1
## 4       19   BLSC4   1
## 5       19   SCAU2   1
## 6       19   BLSC4   1
```

Quick, easy, and intuitive. Now let's only keep observations of 3 or more.

```r
aerial.subset2 = filter(aerial.subset, Num > 2) #give me observations greater than 2

head(aerial.subset2)
```

```
##   Transect Species Num
## 1       15   SCAU1   5
## 2       16   BLSC3   3
## 3       27   START  27
## 4       27   ENDPT  27
## 5       26   START  26
## 6       26  BLSCF2   4
```

These are useful, but notice how we keep having to save out our results as new names with an increasing number on the end? This will get messy and out of control fast as we add on more filters and subsets and summaries. To combat this, the Tidyverse (specifically the magrittr package) has the pipe operator (%>%). The pipe operator chains together a series of functions and uses the results of the previous pipes in the subsequent pipes. It makes for a cleaner sequence that you can track later. For example, the sequence below will subset our columns and filter to bigger observations as we did above, but also filter out the START and ENDPT observations, then count the number of observations by species that met the conditions leading up to the aggregate.

```r
obs.by.species = aerial %>%

  #grab Transect, Species, and Num only from aerial
  select(Transect, Species, Num)  %>%

  #Num greater than 2, but remove START and ENDPT observations
  filter(Num > 2 & !(Species %in% c("START", "ENDPT")) ) %>%


  aggregate(Num~Species, .,FUN=length)

  names(obs.by.species)[2]="N.obs"

obs.by.species
```

```
##   Species N.obs
## 1   BLSC3      2
## 2   BLSC4      1
## 3  BLSCF2      1
## 4  BLSCF3      2
## 5   LTDU3      1
## 6   SCAU1      6
## 7   SCAU2      2
## 8   SCAU3      7
```

We have a lot going on there. Note the command %in% that will only keep string values that it finds within a set of other string values. But in this case, I wrap it in !, which can be read as "not." So that line becomes "filter down to observations greater than 2 and where Species is not START or ENDPT."

Now that we have the pipe in our toolbox, let's revisit the actual messy data problem.

```
messy1
```

```
##   Species Box Visit1 Visit2 Visit3 Visit4 Visit5
## 1    COGO   1      5      5      0     NA     NA
## 2    COGO   2      6      6      6      0     NA
## 3    COGO   3      6      5      5      0     NA
## 4    BOOW   4      4      4      4      1      0
## 5    BOOW   5      5      5      5      0     NA
## 6    BOOW   6      4      7      0     NA     NA
```

We have 6 nest boxes that were visited up to 5 times and we just want a summary of the number of visits and the average number of eggs in the nest each visit. We have incorrectly recorded values of our variable Visit as column headers. Let's fix it.

```
tidy1 = messy1 %>%
  pivot_longer(c("Visit1", "Visit2","Visit3","Visit4","Visit5",),
               names_to = "Visit",
               values_to = "eggs")

tidy1
```

```
## # A tibble: 30 x 4
##    Species   Box Visit   eggs
##    <chr>   <int> <chr>  <dbl>
##  1 COGO        1 Visit1     5
##  2 COGO        1 Visit2     5
##  3 COGO        1 Visit3     0
##  4 COGO        1 Visit4    NA
##  5 COGO        1 Visit5    NA
##  6 COGO        2 Visit1     6
##  7 COGO        2 Visit2     6
##  8 COGO        2 Visit3     6
##  9 COGO        2 Visit4     0
## 10 COGO        2 Visit5    NA
## # ... with 20 more rows
```

This result is. . . ehhhh. It looks ok, but we can do better.

```
tidy1 = messy1 %>%
  pivot_longer(
   cols = starts_with("Visit"),
   names_to = "Visit",
   names_prefix = "Visit",
   values_to = "Eggs",
   values_drop_na = TRUE
 )

head(tidy1,10)
```

```
## # A tibble: 10 x 4
##    Species   Box Visit  Eggs
##    <chr>   <int> <chr> <dbl>
##  1 COGO        1 1         5
##  2 COGO        1 2         5
##  3 COGO        1 3         0
##  4 COGO        2 1         6
##  5 COGO        2 2         6
##  6 COGO        2 3         6
##  7 COGO        2 4         0
##  8 COGO        3 1         6
##  9 COGO        3 2         5
## 10 COGO        3 3         5
```

We took messy1 and applied pivot_longer, which will translate our column values into expanded row values. We further parameterized it by asking for all columns that started with Visit (cols = starts_with("Visit")), and removed the prefix Visit (names_prefix = "Visit") and sent the remainder to a new Visit column (names_to = "Visit"). We then took the row values (counts of eggs, though you can't tell from the data) and sent them to the appropriate new Species-Box-Visit row combination and labelled the new column Eggs (values_to = "Eggs"). Finally, we removed the NAs. These were causing problems above and since they were really just empty cells and not visits, they are better off gone. Now we can cleanly summarize our data!

```
vis = tidy1 %>%
  aggregate(Visit~Species + Box, ., FUN=max)
names(vis)[3] = "Num.Visits"

vis
```

```
##   Species Box Num.Visits
## 1    COGO   1          3
## 2    COGO   2          4
## 3    COGO   3          4
## 4    BOOW   4          5
## 5    BOOW   5          4
## 6    BOOW   6          3
```

```
avg.egg = tidy1 %>%
  filter(Visit == 1) %>%
  aggregate(Eggs~Species, ., FUN=mean)
names(avg.egg)[2] = "Avg.Eggs"

avg.egg
```

```
##   Species Avg.Eggs
## 1    BOOW 4.333333
## 2    COGO 5.666667
```

```
#Or using just tidyverse

tidy1 %>%
 filter(Visit == 1) %>%
 group_by(Species) %>%
 summarize(Avg.Eggs=mean(Eggs))
```

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

```
## # A tibble: 2 x 2
##   Species Avg.Eggs
##   <chr>      <dbl>
## 1 BOOW        4.33
## 2 COGO        5.67
```

3.1.2 Multiple Variables in One Column

Another common problem in MBM data sets is combining the values of multiple variables into a single column. Sometimes this is what we really thought was the best way to store values, and other times we make excuses about the limitations of our data collection programs or devices to store things certain ways. The results are messy data either way, and this serves to increase the rate of decay of the actual information we can extract from our data. Consider the following (fake) data:

```
messy2 = data.frame(monthday=paste(rep(6,10), c(1:10), sep="/"), Grade=rep(c("A", "F"), 5))

messy2
```

```
##    monthday Grade
## 1       6/1     A
## 2       6/2     F
## 3       6/3     A
## 4       6/4     F
## 5       6/5     A
## 6       6/6     F
## 7       6/7     A
## 8       6/8     F
## 9       6/9     A
## 10     6/10     F
```

In this simple example, we have monthday, which is the month and day separated by a period, and a fictitious grade received on that day. We should really have month and day as their own columns. Let's separate.

```
messy2 %>%
    separate(monthday, into=c("Month", "Day"), sep="/")
```

```
##    Month Day Grade
## 1      6   1     A
## 2       6   2     F
## 3       6   3     A
## 4       6   4     F
## 5       6   5     A
## 6       6   6     F
## 7       6   7     A
## 8       6   8     F
## 9       6   9     A
## 10      6  10     F
```

That was suspiciously easy…so let's try it with our aerial data. The column Species sounds like it should be the species. But it isn't. It is actually 3 values: species, behavior, and distance. But that isn't the end of it. The behavior was only recorded if the bird was flying. Pretty messy. So let's tidy it.

```r
unique(aerial$Species)
```

```
##  [1] "START"  "SCAU4"  "LTDUF4" "BLSC4"  "SCAU2"  "BLSCF3" "BLSC3" 
##  [8] "BLSCF4" "SCAU3"  "BLSC2"  "LTDU4"  "SCAU1"  "LTDU3"  "BLSC1" 
## [15] "ENDPT"  "LTDU2"  "LTDU1"  "BLSCF1" "BLSCF2" "WWSC2"
```

```r
aerial.tidy = aerial %>%
 filter(!(Species %in% c("START", "ENDPT"))) %>%  #remove start and end points
 select(Lat, Long, Species, Num) %>% #filter to just 4 columns for illustration
 separate(Species, into=c("Species", "Distance"), sep = -1) %>% #take away the rightmost string value
 separate(Species, into=c("Species", "Behavior"), sep = 4)  #take away the first 4 string values


 head(aerial.tidy)
```

```
##       Lat      Long Species Behavior Distance Num
## 1 61.1457 -165.4586    SCAU                 4   1
## 2 61.1564 -164.8116    LTDU        F        4   1
## 3 61.1587 -164.6340    BLSC                 4   1
## 4 61.1595 -164.5453    SCAU                 2   1
## 5 61.1605 -164.4803    BLSC                 4   1
## 6 61.1605 -164.4803    BLSC                 4   1
```

```r
 unique(aerial.tidy$Species)
```

```
## [1] "SCAU" "LTDU" "BLSC" "WWSC"
```

An initial look at the Species column showed that the Distance was always the last value in the string. So we were able to tell separate that we need to pull that one out.

```r
 separate(Species, into=c("Species", "Distance"), sep = -1)
```

Then after pulling that one out, we saw that the species code was always the first 4 characters in the string, so we were able to tell separate to always give us those.

```r
 separate(Species, into=c("Species", "Behavior"), sep = 4)
```

With the pipe, we did it all in one command, and we can move on to bigger and better things, such as visualization!

4.0.0 Tables and Figures

The first step in any data manipulation, analysis, QA/QC process, or almost any other data-related activity should be to visualize your data. This will be the best way to catch any potential inconsistencies or errors before you embed them deep in an analysis or report. Base R provides many ways to visualize data quickly and easily, but contributed packages such as DT, kableExtra, ggplot2, and leaflet expand on base R to provide publication-quality tables and figures. We will start with base R tables.

4.1.1 Base R Tables

Let's make sure we have our aerial data loaded and tidy.

```r
library(tidyverse)

aerial=read.csv(
  "https://raw.githubusercontent.com/cfrost3/MBM_R_Short_Course/master/BLSC_2018_RawObs_Fake.csv",
  header=TRUE,
  stringsAsFactors=FALSE)

aerial.tidy = aerial %>%
 filter(!(Species %in% c("START", "ENDPT"))) %>%  #remove start and end points
 select(Lat, Long, Species, Num) %>% #filter to just 4 columns for illustration
 separate(Species, into=c("Species", "Distance"), sep = -1) %>% #take away the rightmost string value
 separate(Species, into=c("Species", "Behavior"), sep = 4)  #take away the first 4 string values


 head(aerial.tidy)
```

```
##       Lat      Long Species Behavior Distance Num
## 1 61.1457 -165.4586    SCAU                 4   1
## 2 61.1564 -164.8116    LTDU        F        4   1
## 3 61.1587 -164.6340    BLSC                 4   1
## 4 61.1595 -164.5453    SCAU                 2   1
## 5 61.1605 -164.4803    BLSC                 4   1
## 6 61.1605 -164.4803    BLSC                 4   1
```

```r
table(aerial.tidy$Species,aerial.tidy$Distance)
```

```
##
##          1   2   3   4
##   BLSC  23  43 146  42
##   LTDU   5   7  11   2
##   SCAU  54  53 120  12
##   WWSC   0   1   0   0
```

The format is table(rows, columns), and we can even add multidimensionality:

```
table(aerial.tidy$Species,aerial.tidy$Distance, aerial.tidy$Behavior)
```

```
## , ,  =
##
##
##          1   2   3   4
##   BLSC  21  42 136  39
##   LTDU   5   7  11   1
##   SCAU  54  53 120  12
##   WWSC   0   1   0   0
##
## , ,  = F
##
##
##          1   2   3   4
##   BLSC   2   1  10   3
##   LTDU   0   0   0   1
##   SCAU   0   0   0   0
##   WWSC   0   0   0   0
```

One important thing to remember with table in base R is that the default summary will simply give you a count of the number of times the 2 observations occurred together (NOT the total number of things we counted by species in this example). Always keep track of what you are visualizing. You can also easily calculate the proportion of observations in a table.

```
prop.table(table(aerial.tidy$Species,aerial.tidy$Distance))
```

```
##
##                  1           2           3           4
##   BLSC 0.044315992 0.082851638 0.281310212 0.080924855
##   LTDU 0.009633911 0.013487476 0.021194605 0.003853565
##   SCAU 0.104046243 0.102119461 0.231213873 0.023121387
##   WWSC 0.000000000 0.001926782 0.000000000 0.000000000
```

This proportion table tells us that our most frequent observation (roughly 28% of the time) was BLSC in distance bin 3. Notice how we had to wrap the table() function with prop.table()? That is because while table() expects (rows, columns) format, prop.table expects a table() as input. We can even wrap the whole thing in a max() to verify our 28%.

```
max(prop.table(table(aerial.tidy$Species,aerial.tidy$Distance)))
```

```
## [1] 0.2813102
```

One thing we would certainly want to know from our aerial data collection is the number (Num) of each species we counted. We can get that number (albeit inefficiently) using tables.

```
table(aerial.tidy$Species,aerial.tidy$Num)
```

```
##
##          1  10   2  25   3  30   4   5   6   7   d   t
##   BLSC 245   0   3   0   4   0   1   0   1   0   0   0
##   LTDU  24   0   0   0   0   0   0   0   0   0   0   1
##   SCAU 215   1   8   2   7   1   1   1   1   1   1   0
##   WWSC   1   0   0   0   0   0   0   0   0   0   0   0
```

```
str(aerial.tidy)
```

```
## 'data.frame':    519 obs. of  6 variables:
##  $ Lat     : num  61.1 61.2 61.2 61.2 61.2 ...
##  $ Long    : num  -165 -165 -165 -165 -164 ...
##  $ Species : chr  "SCAU" "LTDU" "BLSC" "SCAU" ...
##  $ Behavior: chr  "" "F" "" "" ...
##  $ Distance: chr  "4" "4" "4" "2" ...
##  $ Num     : chr  "1" "1" "1" "1" ...
```

Well, that won't work. It seems we've identified a QA/QC issue. We could've caught this by checking the structure of our data frame as well (see how Num shows as chr?). Let's pluck those 2 offenders out and fix them. We will find them and fix them in 2 different ways for illustration only. One useful function for this is which(). It will return the index or indices (similar to the row number in Excel) of the row(s) of the observation(s) that fit a condition. In our case, we want to know where the d and t are hanging out.

```
which(aerial.tidy$Num %in% c("d", "t"))  #what positions are the d and t in?
```

```
## [1] 427 438
```

```
aerial.tidy$Num[427] #verify
```

```
## [1] "d"
```

```
aerial.tidy[427,] #show me that whole row 427 and all of its columns
```

```
##         Lat      Long Species Behavior Distance Num
## 427 66.2326 -165.6542    SCAU                 1   d
```

```
aerial.tidy[c(427,438),] #show me both row 427 and 438 and all of their columns
```

```
##         Lat      Long Species Behavior Distance Num
## 427 66.2326 -165.6542    SCAU                 1   d
## 438 66.2322 -164.0431    LTDU                 3   t
```

Let's imagine that we had our observer go back through the recordings (because both the observer and recordings still exist), and they found that d should be 2 and t should be 1. Since we know the indices of the offenders, we can just overwrite them.

```
aerial.tidy$Num[427] = 2
```

Alternatively, we can grab the index and do something to it all at once:

```
aerial.tidy$Num[aerial.tidy$Num == "t"] = 1

#find any values of Num == "t" and replace with 1
```

If we table it again, we can see that we have at least removed the d and t observations. But the structure shows Num still contained as character instead of numeric.

```
table(aerial.tidy$Species,aerial.tidy$Num)
```

```
##
##          1  10   2  25   3  30   4   5   6   7
##   BLSC 245   0   3   0   4   0   1   0   1   0
##   LTDU  25   0   0   0   0   0   0   0   0   0
##   SCAU 215   1   9   2   7   1   1   1   1   1
##   WWSC   1   0   0   0   0   0   0   0   0   0
```

```
str(aerial.tidy)
```

```
## 'data.frame':    519 obs. of  6 variables:
##  $ Lat     : num  61.1 61.2 61.2 61.2 61.2 ...
##  $ Long    : num  -165 -165 -165 -165 -164 ...
##  $ Species : chr  "SCAU" "LTDU" "BLSC" "SCAU" ...
##  $ Behavior: chr  "" "F" "" "" ...
##  $ Distance: chr  "4" "4" "4" "2" ...
##  $ Num     : chr  "1" "1" "1" "1" ...
```

To fix that, we can just tell aerial.tidy to reconsider that column as numeric. If we get any warnings or errors, we can assume there were other offenders in there. In our case, there weren't. Now when we table it, the columns even reorder correctly.

```
aerial.tidy$Num = as.numeric(aerial.tidy$Num)

str(aerial.tidy)
```

```
## 'data.frame':    519 obs. of  6 variables:
##  $ Lat     : num  61.1 61.2 61.2 61.2 61.2 ...
##  $ Long    : num  -165 -165 -165 -165 -164 ...
##  $ Species : chr  "SCAU" "LTDU" "BLSC" "SCAU" ...
##  $ Behavior: chr  "" "F" "" "" ...
##  $ Distance: chr  "4" "4" "4" "2" ...
##  $ Num     : num  1 1 1 1 1 1 1 1 1 1 ...
```

```
table(aerial.tidy$Species,aerial.tidy$Num)
```

```
##
##          1   2   3   4   5   6   7  10  25  30
##   BLSC 245   3   4   1   0   1   0   0   0   0
##   LTDU  25   0   0   0   0   0   0   0   0   0
##   SCAU 215   9   7   1   1   1   1   1   2   1
##   WWSC   1   0   0   0   0   0   0   0   0   0
```

Now if we want a basic summary of our observations, we can use margin.table to sum the table across either or both dimensions. In other words, a sum by rows, a sum by columns, or a total sum (keeping in mind our unit is observation and NOT number of things seen). Note that margin.table is like prop.table and the argument passed must be a table object.

```
#The 1 specifies the first (row) dimension (how many times did we observe BLSC?)
margin.table(table(aerial.tidy$Species,aerial.tidy$Num), 1)
```

```
##
## BLSC LTDU SCAU WWSC
##  254   25  239    1
```

```
#The 2 specifies the second (column) dimension (how many times did we see only 1 of something?)
margin.table(table(aerial.tidy$Species,aerial.tidy$Num), 2)
```

```
##
##   1   2   3   4   5   6   7  10  25  30
## 486  12  11   2   1   2   1   1   2   1
```

```
#No dimension argument gives us the sum over both rows and columns (how many total observations were th
margin.table(table(aerial.tidy$Species,aerial.tidy$Num))
```

```
## [1] 519
```

4.1.2 Tables in Contributed Packages

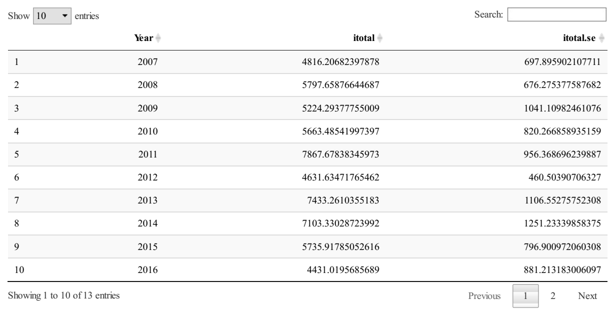
We will demonstrate table functions from 2 contributed (user-developed) packages, DT and kableExtra. DT stands for data table and works best to visualize large data sets in a clean, searchable, sortable, filterable way. kableExtra is a handy package for creating tables that use the functionality of html coding to render in either html or pdf reports. We need to make sure we have both installed and loaded.

```
install.packages(c("DT", "kableExtra", "htmltools", "zoo"))
```

```
library(DT)
library(kableExtra)
library(AKaerial)
library(tidyverse)
library(htmltools)
library(zoo)
```

For the first example, we will load a table of real aerial index estimates from the AKaerial package and use some of our tidyverse functions to trim it down for a specific table using datatable from the package DT.

```
ACPHistoric$combined %>%
  filter(Species=="SPEI") %>%
  select(Year, itotal, itotal.se) %>%
  datatable()
```



	Year	itotal	itotal.se
1	2007	4816.20682397878	697.895902107711
2	2008	5797.65876644687	676.275377587682
3	2009	5224.29377755009	1041.10982461076
4	2010	5663.48541997397	820.266858935159
5	2011	7867.67838345973	956.368696239887
6	2012	4631.63471765462	460.50390706327
7	2013	7433.2610355183	1106.55275752308
8	2014	7103.33028723992	1251.23339858375
9	2015	5735.91785052616	796.900972060308
10	2016	4431.0195685689	881.213183006097

That's a solid first effort, but we can do better. Our itotal column really shouldn't allow decimal amounts of birds, and the standard error could use some trimming. I also don't particularly like seeing the row numbers in there. And the column names could be better. And we need a title, maybe shrink the width, center that middle column, and show me all of the entries at once since there are only 13.

```r
ACPHistoric$combined %>%
  filter(Species=="SPEI") %>%
  select(Year, itotal, itotal.se) %>%
  mutate_at("itotal", ~round(., 0)) %>%
  mutate_at("itotal.se", ~round(., 2)) %>%
  datatable(rownames=FALSE,  #cut out those row numbers showing
            fillContainer = TRUE,  #auto-size the column widths
            colnames=c("Year","Indicated Total","SE"),  #change column names
            #set the caption obtusely using html styling
            caption = htmltools::tags$caption( style = 'caption-side: top;
                                                text-align: center;
                                                color:black;
                                                font-size:100% ;',
                                                'Table 1: Indicated Total SPEI on the ACP, 2007-2019.')
            options=list(
              autoWidth=TRUE,
              #center the column indexed 1 (starts with 0)
              columnDefs = list(list(className = 'dt-center', targets = 1)),
              pageLength=length(.[,1]),  #display all of the data points
              dom=""))   #take away the search and page functionality
```

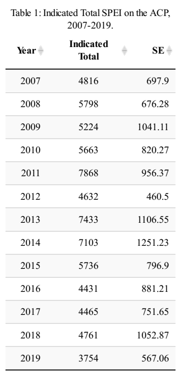


Table 1: Indicated Total SPEI on the ACP, 2007-2019.

Year	Indicated Total	SE
2007	4816	697.9
2008	5798	676.28
2009	5224	1041.11
2010	5663	820.27
2011	7868	956.37
2012	4632	460.5
2013	7433	1106.55
2014	7103	1251.23
2015	5736	796.9
2016	4431	881.21
2017	4465	751.65
2018	4761	1052.87
2019	3754	567.06

Now that's a table! It renders a little small though in this pdf since html styling doesn't jive well with pdf output. It should look great in your console or in an html report. We used a new tidyverse function mutate_at. That changes the styling or content of certain columns only, referenced by name or number. In this case, we applied the function round() to the piped data frame and told it to use 0 and 2 decimal places. The rest of the changes were implemented as arguments to datatable().

Now we will implement a similar table using kableExtra.

```r
YKGHistoric$combined %>%
  filter(Species=="BRAN") %>%
  select(Year, itotal, itotal.se) %>%
  kable()
```

Year	itotal	itotal.se
1985	4587.359	1377.462
1986	13065.045	2009.412
1987	12316.825	1228.657
1988	19774.425	2354.796
1989	26414.090	3239.270
1990	28371.901	3450.564
1991	21219.565	2389.966
1992	19531.009	1910.923
1993	31543.300	3336.424
1994	30487.396	2591.968
1995	34041.643	3187.085
1996	29077.673	2365.832
1997	30450.473	2757.902
1998	20518.538	1500.905
1999	21101.275	1846.622
2000	24619.576	2270.803
2001	30592.181	2658.417
2002	19623.449	1361.620
2003	20019.753	1555.904
2004	19172.114	1466.754
2005	20666.951	1734.801
2006	18835.411	1616.762
2007	19891.241	1858.726
2008	28361.608	1930.942
2009	23000.566	1516.937
2010	23107.933	1811.589
2011	16261.917	1243.417
2012	21650.089	2671.557
2013	23867.171	2953.542
2014	23160.193	2394.913
2015	20856.023	1532.706
2016	29738.491	2331.881
2017	21038.565	1796.098
2018	26132.308	2317.572
2019	22099.878	1869.757

That might be good enough for some journals… but not for me. Let's get our rounding going, change those column names, and even add a 3-year and 10-year rolling average (from the zoo package). Eh… and a caption, a footnote, and a color scheme, just to give it that "next-level" feel.

```
YKGHistoric$combined %>%
  filter(Species=="EMGO") %>%

  select(Year, itotal, itotal.se) %>%

  #rollapply in zoo rolls through a vector and applies a function to the segments

  mutate(avg3=rollapply(itotal,3,mean,align='right',fill=NA)) %>%

  mutate(avg10=rollapply(itotal,10,mean,align='right',fill=NA)) %>%

  mutate_at(c("itotal", "avg3", "avg10"), ~round(., 0)) %>%
  mutate_at("itotal.se", ~round(., 2)) %>%

  #apply our conditional population objective coloring
  mutate_at("itotal", function(x) {
    cell_spec(x, "html", color = ifelse(x > 26000, "green", "red"))
    }) %>%

  kable(format="html",
        escape = F, #html scheme requirement to make the color statement work
        #now we paste in the footnote denotation on the indicated total column
        col.names = c("Year", "Indicated Total*",
                      "SE", "3-year Avg", "10-year Avg"),
        #and add the top caption
        caption = "Table 2: Indicated Total EMGO on the YK-Delta, 1985-2019, including 3- and 10-year A

  #adding the footnote is done in its own function so we pipe the kable to it

  footnote(symbol= "Indicated Total is used by the AMBCC to regulate harvest. Values of 26000 and above

  #this tells us to use the default bordered style
  kable_styling("bordered",
                full_width=FALSE,
                font_size = 14)
```

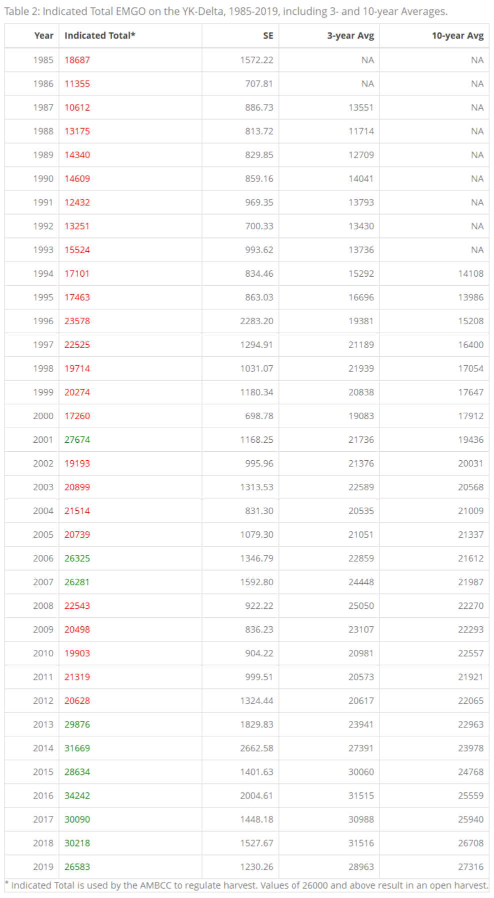


Table 2: Indicated Total EMGO on the YK-Delta, 1985-2019, including 3- and 10-year Averages.

Year	Indicated Total*	SE	3-year Avg	10-year Avg
1985	18687	1572.22	NA	NA
1986	11355	707.81	NA	NA
1987	10612	886.73	13551	NA
1988	13175	813.72	11714	NA
1989	14340	829.85	12709	NA
1990	14609	859.16	14041	NA
1991	12432	969.35	13793	NA
1992	13251	700.33	13430	NA
1993	15524	993.62	13736	NA
1994	17101	834.46	15292	14108
1995	17463	863.03	16696	13986
1996	23578	2283.20	19381	15208
1997	22525	1294.91	21189	16400
1998	19714	1031.07	21939	17054
1999	20274	1180.34	20838	17647
2000	17260	698.78	19083	17912
2001	27674	1168.25	21736	19436
2002	19193	995.96	21376	20031
2003	20899	1313.53	22589	20568
2004	21514	831.30	20535	21009
2005	20739	1079.30	21051	21337
2006	26325	1346.79	22859	21612
2007	26281	1592.80	24448	21987
2008	22543	922.22	25050	22270
2009	20498	836.23	23107	22293
2010	19903	904.22	20981	22557
2011	21319	999.51	20573	21921
2012	20628	1324.44	20617	22065
2013	29876	1829.83	23941	22963
2014	31669	2662.58	27391	23978
2015	28634	1401.63	30060	24768
2016	34242	2004.61	31515	25559
2017	30090	1448.18	30988	25940
2018	30218	1527.67	31516	26708
2019	26583	1230.26	28963	27316

* Indicated Total is used by the AMBCC to regulate harvest. Values of 26000 and above result in an open harvest.

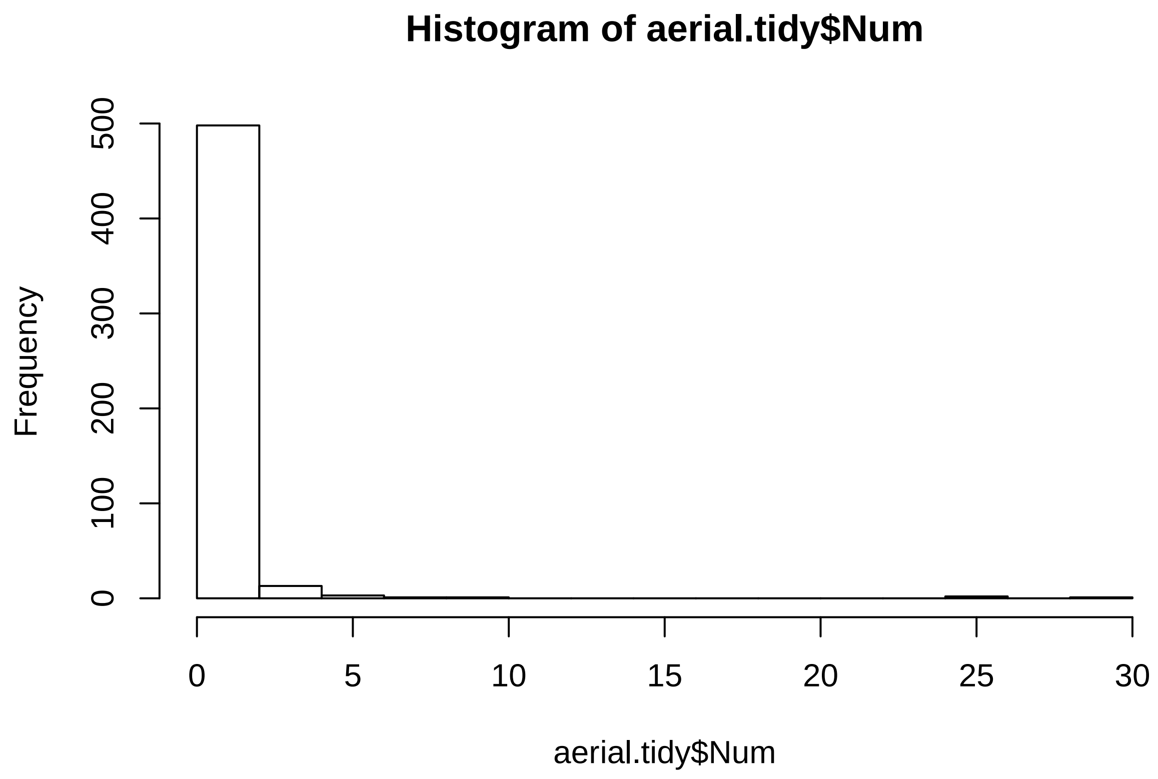
The result could easily be hung on the nearest refrigerator (sorry, the styling will look funny as a pdf). The majority of the work in creating a nice table is getting the formatting just right for your target. It will take some practice to figure out where the option is that changes each of your style elements. Don't forget that a package probably already exists to do exactly whatever manipulation you are dreaming of doing to your data. Search engines are powerful tools.

4.2.1 Plotting in Base R

Base R comes loaded with relatively robust plotting capability. Just like with tables, though, the R community identified many features that were either missing, or difficult to implement, and expanded the R universe with contributed plotting content. We will start in base R with the basics.

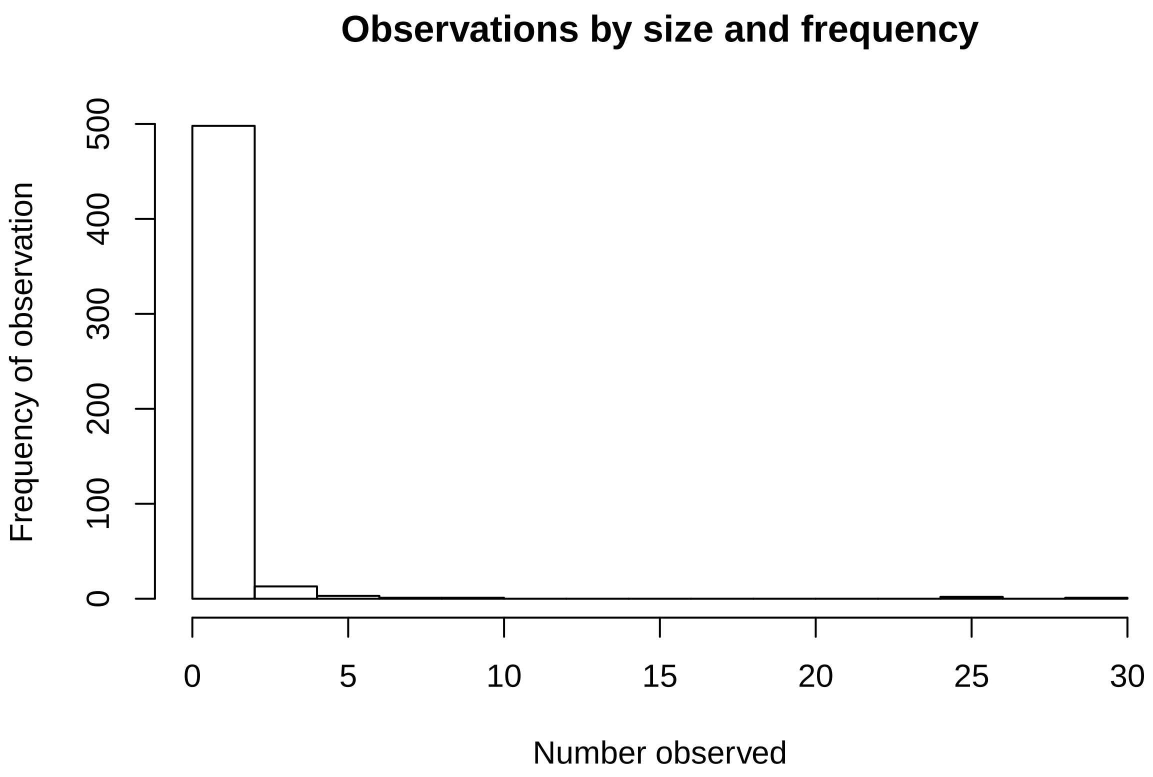
We will revisit our aerial.tidy that we created earlier. Note that we aren't applying any pipes just yet since these visualizations are usually 1 or 2 contrasts and nothing else gets displayed, unlike a table where we want to choose exactly what data doesn't get displayed.

```r
hist(aerial.tidy$Num)
```



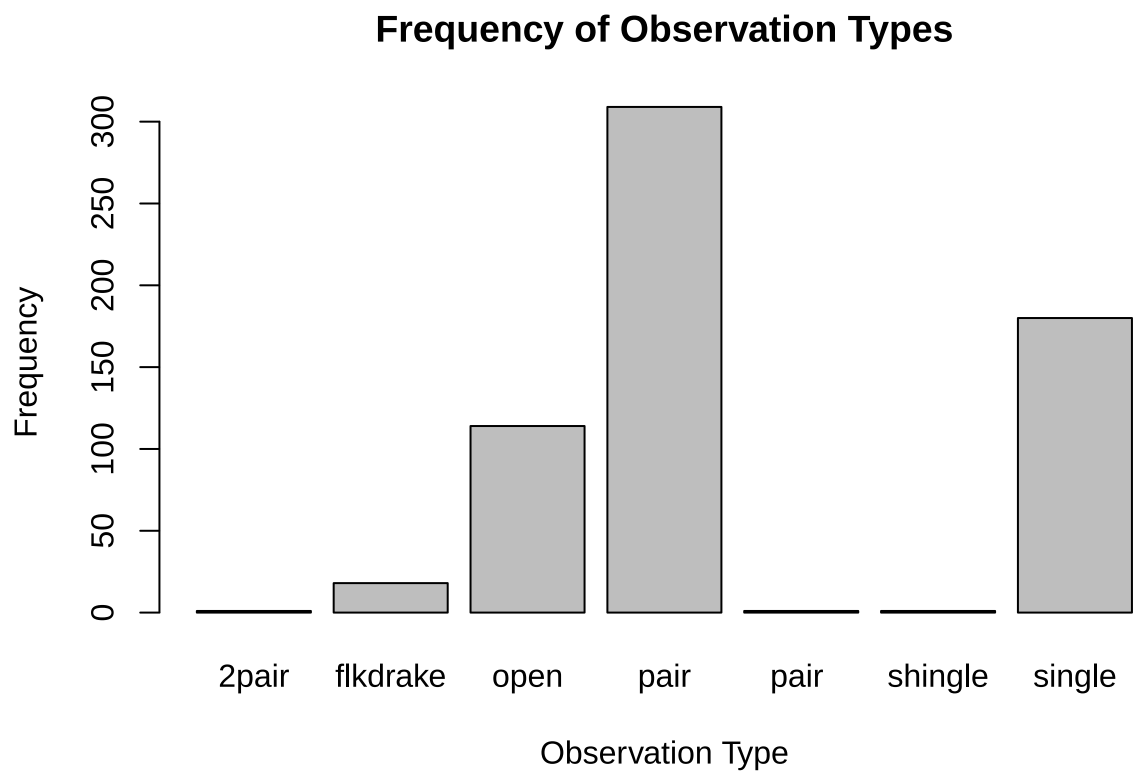
The default x and y axis labels and main title leave us feeling a little empty. Let's fix.

```r
hist(aerial.tidy$Num,
     xlab = "Number observed",
     ylab = "Frequency of observation",
     main = "Observations by size and frequency")
```



Now let's make a barplot. It looks similar to the histogram, but the histogram only works with numeric vectors. A barplot can work with strings and factors converted to tables.

```
counts=table(aerial$Obs_Type)

barplot(counts,
        xlab="Observation Type",
        ylab="Frequency",
        main="Frequency of Observation Types")
```



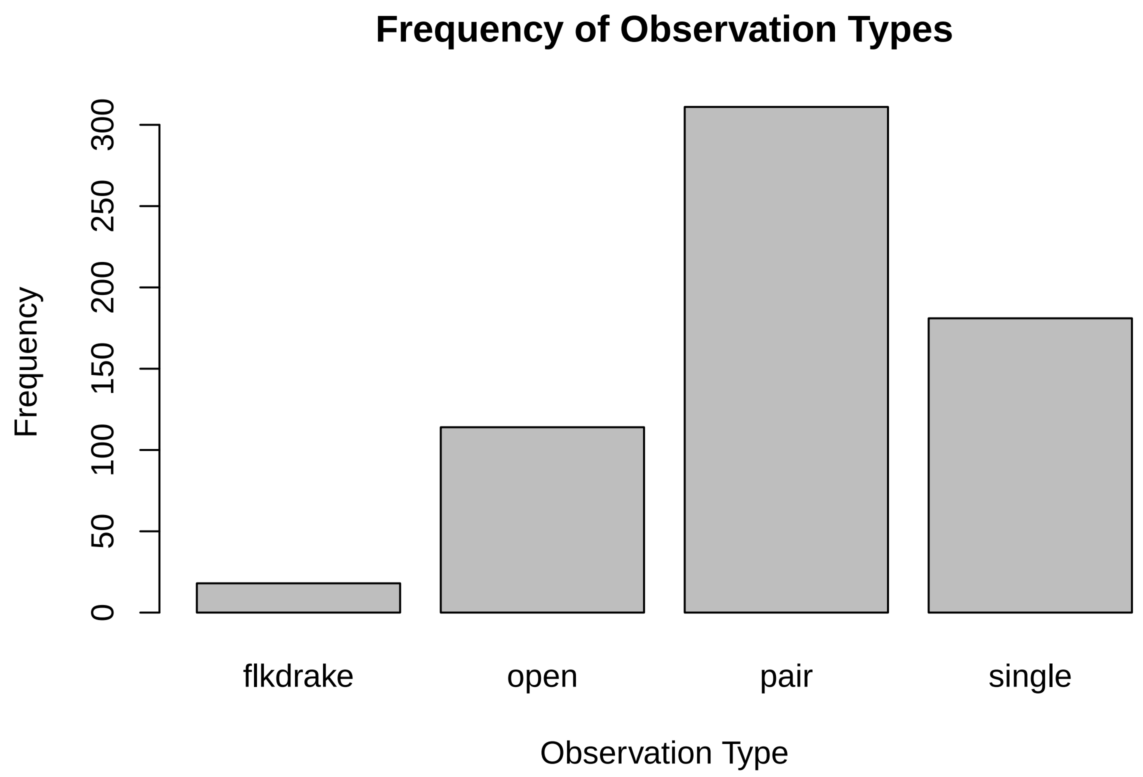
Hmmm. . . looks like more QA/QC problems. They seem pretty intuitive to fix except for one. The second case of "pair" on the x axis is actually "pair" (with a space on the end). This is a common problem with character data and one that R doesn't automatically know how to treat. We would have identified it using a unique(aerial$Obs_Type) call. So let's fix these.

```
aerial$Obs_Type[aerial$Obs_Type %in% c("2pair", "pair ")] = "pair"

aerial$Obs_Type[aerial$Obs_Type == "shingle"] = "single"

unique(aerial$Obs_Type)
```

```
## [1] "open"     "single"   "pair"     "flkdrake"
```

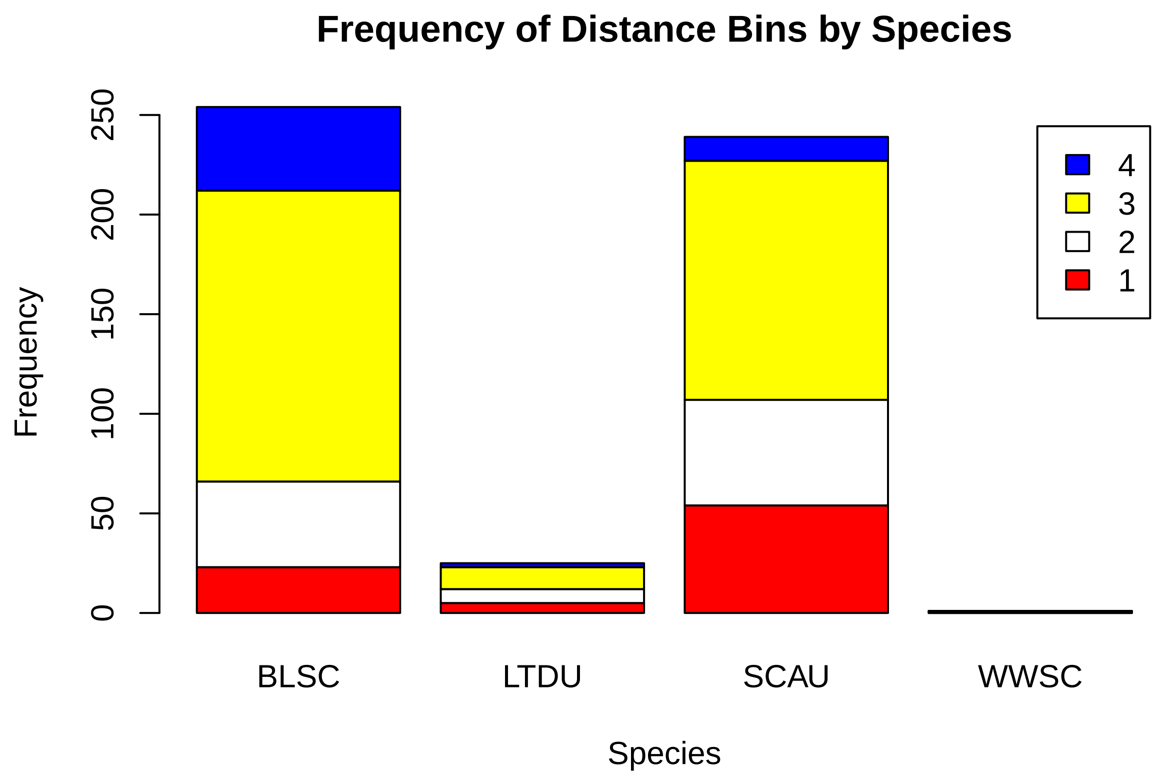
Ok, those look reasonable now. Again with the barplot.

```
counts=table(aerial$Obs_Type)

barplot(counts,
        xlab="Observation Type",
        ylab="Frequency",
        main="Frequency of Observation Types")
```



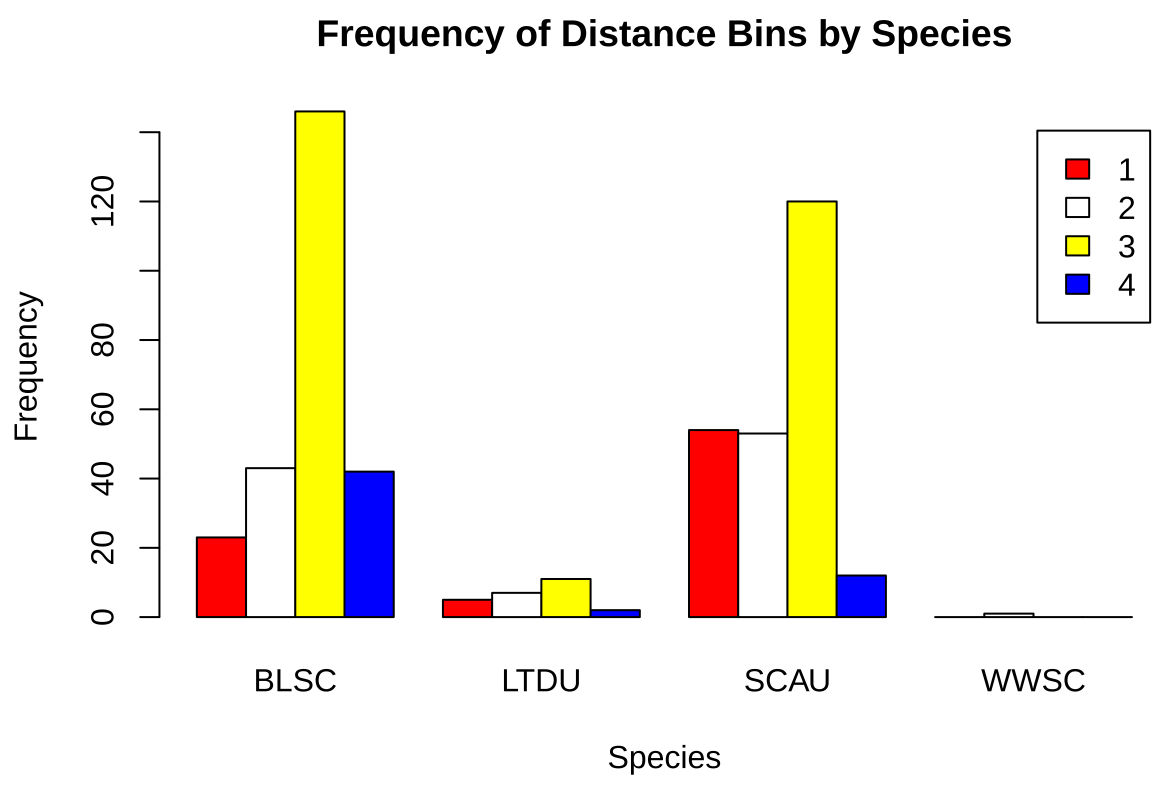
Looks better. We can even do 2-dimensional tables.

```
counts=table(aerial.tidy$Distance, aerial.tidy$Species)

barplot(counts,
        xlab="Species",
        ylab="Frequency",
        main="Frequency of Distance Bins by Species",
        legend=rownames(counts),
        col=c("red","white","yellow","blue"))
```



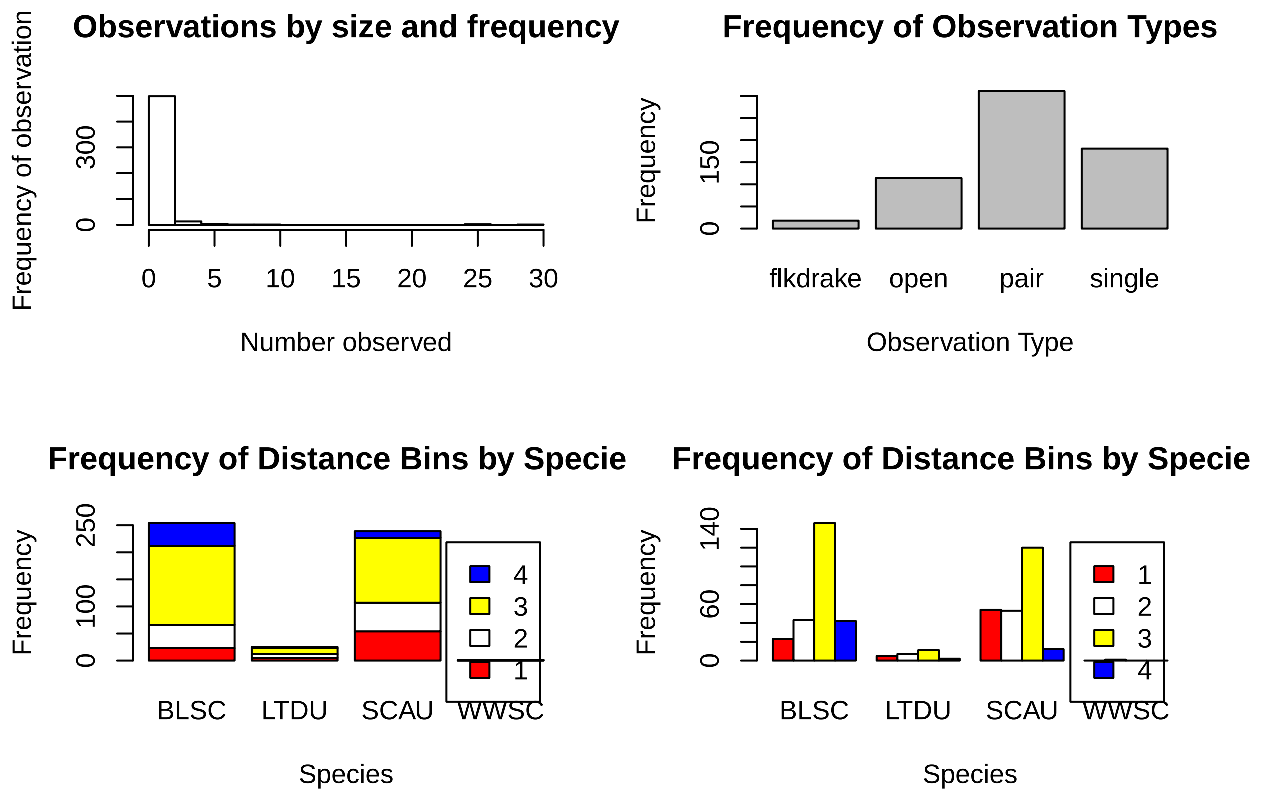
You can place them side by side with the argument beside = TRUE.

```
counts=table(aerial.tidy$Distance, aerial.tidy$Species)

barplot(counts,
        xlab="Species",
        ylab="Frequency",
        main="Frequency of Distance Bins by Species",
        legend=rownames(counts),
        col=c("red","white","yellow","blue"),
        beside=TRUE)
```



You can even configure the R plot window to show you multiple figures at once by changing up the graphical parameters.

```
par(mfrow = c(2,2)) #give me a 2x2 matrix graphical display

#R will fill the 4 spaces with my next 4 plot calls

hist(aerial.tidy$Num,
     xlab = "Number observed",
     ylab = "Frequency of observation",
     main = "Observations by size and frequency")


counts=table(aerial$Obs_Type)

barplot(counts,
        xlab="Observation Type",
        ylab="Frequency",
        main="Frequency of Observation Types")


counts=table(aerial.tidy$Distance, aerial.tidy$Species)

barplot(counts,
        xlab="Species",
        ylab="Frequency",
        main="Frequency of Distance Bins by Species",
        legend=rownames(counts),
        col=c("red","white","yellow","blue"))


counts=table(aerial.tidy$Distance, aerial.tidy$Species)

barplot(counts,
        xlab="Species",
        ylab="Frequency",
        main="Frequency of Distance Bins by Species",
        legend=rownames(counts),
        col=c("red","white","yellow","blue"),
        beside=TRUE)
```

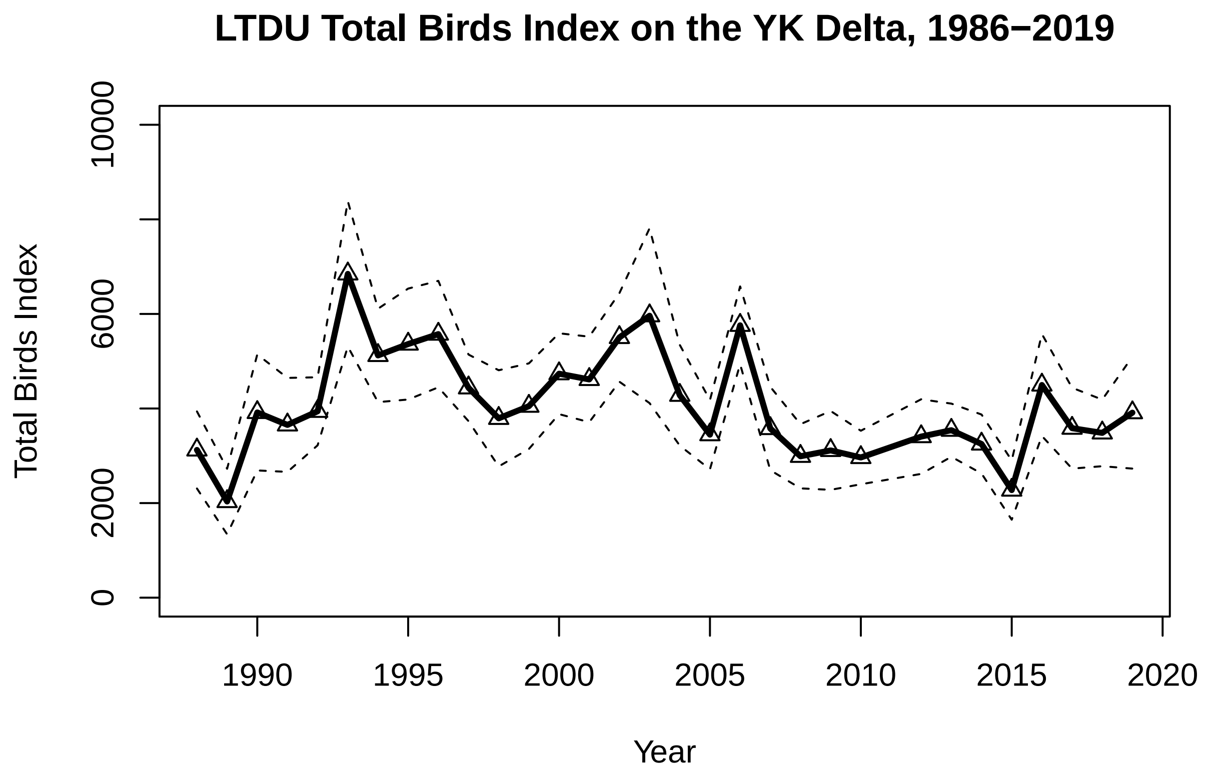


```
#don't forget to change it back!
par(mfrow=c(1,1))
```

The base functions plot(), points(), and lines() work splendidly for time series type data.

```
ltdu = YKDHistoric$combined %>%
  filter(Species == "LTDU")


plot(total~Year, data = ltdu,
```

```r
     type = 'l', #connect the dots
     lwd = 3, #thicken the line
     ylim = c(0,10000), #define the y range instead of letting R calculate it
     ylab = "Total Birds Index",
     main = "LTDU Total Birds Index on the YK Delta, 1986-2019")

lines((total-1.96*total.se)~Year, data = ltdu, lty = 2)

lines((total+1.96*total.se)~Year, data = ltdu, lty = 2)

points(total~Year, data = ltdu,
       pch = 2)  #plot character 2, triangles
```



Here we only have 1 call to a new plot object. The functions lines() and points() will place their output on top of the current plot (as long as they are in scale!). Some data types even have their own plot processes that run when you try to plot them. We will need a couple more packages, though, to deal with spatial data.

```r
install.packages(c("rgdal", "sp", "geojsonio", "ggplot2", "leaflet"))
```

```r
library(rgdal)
library(sp)
library(geojsonio)
```

```
## Warning: package 'geojsonio' was built under R version 3.6.3
```

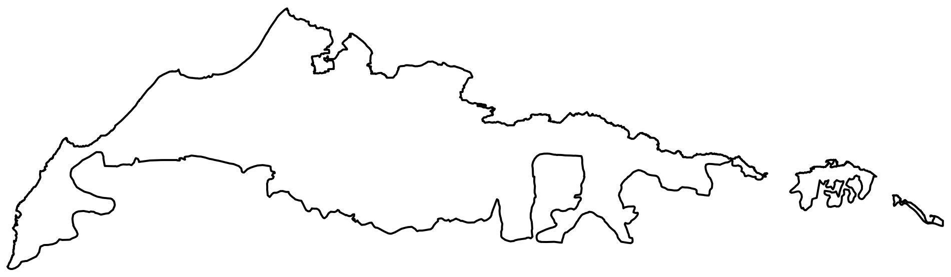
```r
library(ggplot2)
library(leaflet)
```

The following series of commands will be doing a few things:

1. Download the PlotData.zip container from my GitHub page
2. Place the file in your current working directory
3. Unzip the files from the container
4. Delete the .zip file
5. Read in the ACP ESRI shp file
6. Project the file in WGS84
7. Plot the polygons using base R plot()

```r
download.file("http://github.com/cfrost3/MBM_R_Short_Course/raw/master/PlotData/PlotData.zip",
              destfile = "PlotData.zip" , mode='wb')
unzip("PlotData.zip", exdir = "./Plot")
file.remove("PlotData.zip")
```

```r
map= readOGR("./Plot/ACP_2007to2019_StudyArea.shp",
  layer = "ACP_2007to2019_StudyArea",
  verbose=FALSE)

map.proj <- spTransform(map, "+proj=longlat +ellps=WGS84 +datum=WGS84")

plot(map.proj)
```

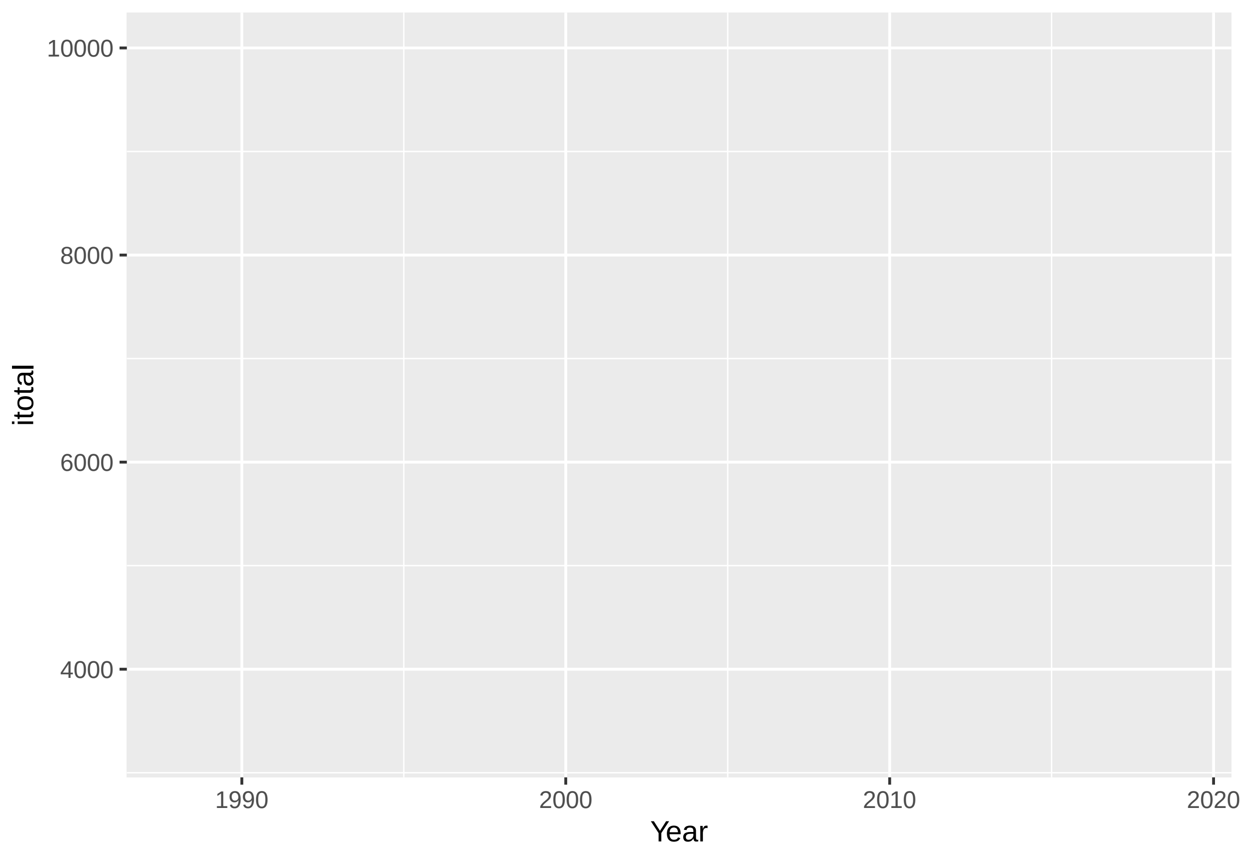


This is definitely the dry white toast of GIS mapping. We could make it work if we had to, but luckily we have plenty of added features in the contributed packages ggplot2 and leaflet to visualize our spatial (and time series, and virtually every other possible type of) data.

4.2.2 Plotting in Contributed Packages

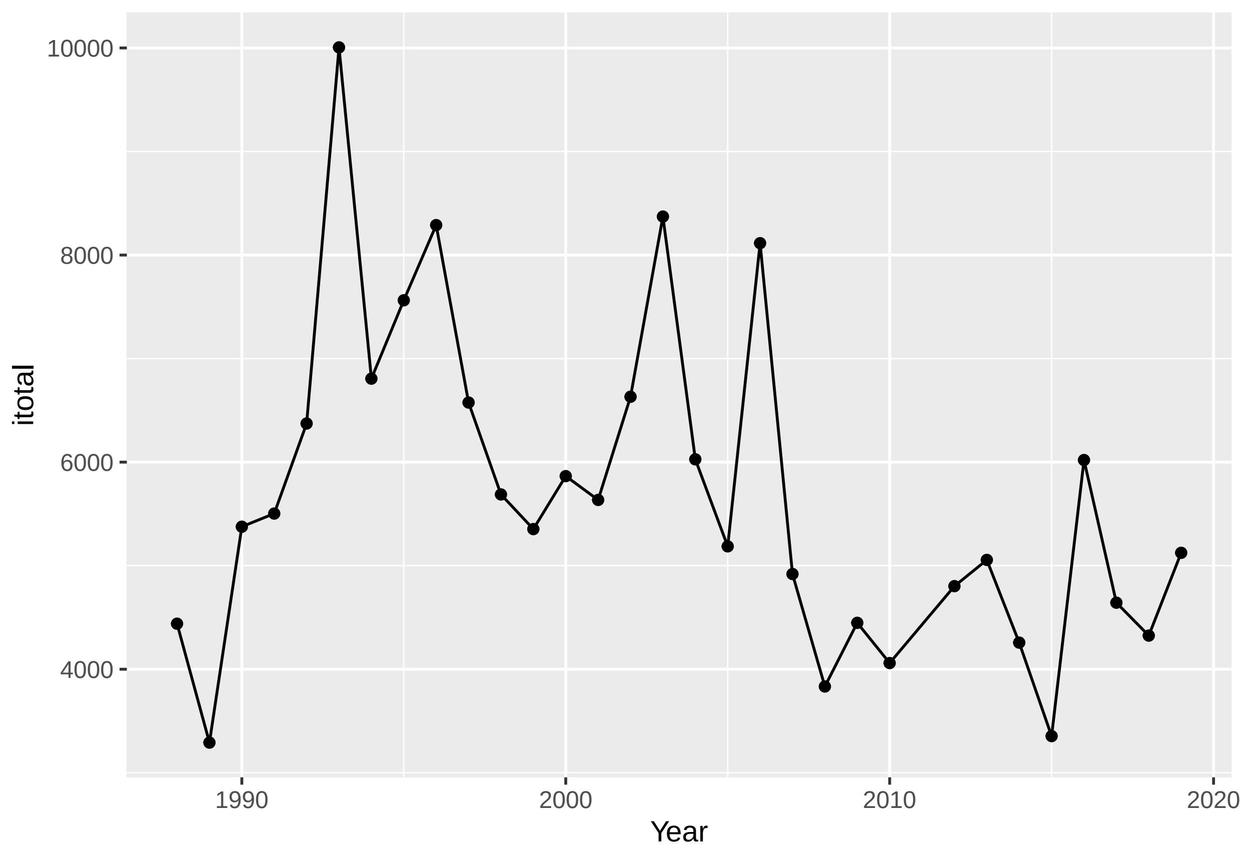
The best package (currently) to learn to plot in is ggplot2. It works with tidy data to produce publication-quality figures. The major difference in base R plotting and ggplot2 is that while in base R you are passing vectors, in ggplot2 you use whole data frames. Individual variables can be extracted from the data frame in various ways to add feature layers to your ggplot. We will start by recreating the long-tailed duck plot from earlier.

```r
YKDHistoric$combined %>%
  filter(Species == "LTDU") %>%
  ggplot(aes( x = Year, y = itotal))
```



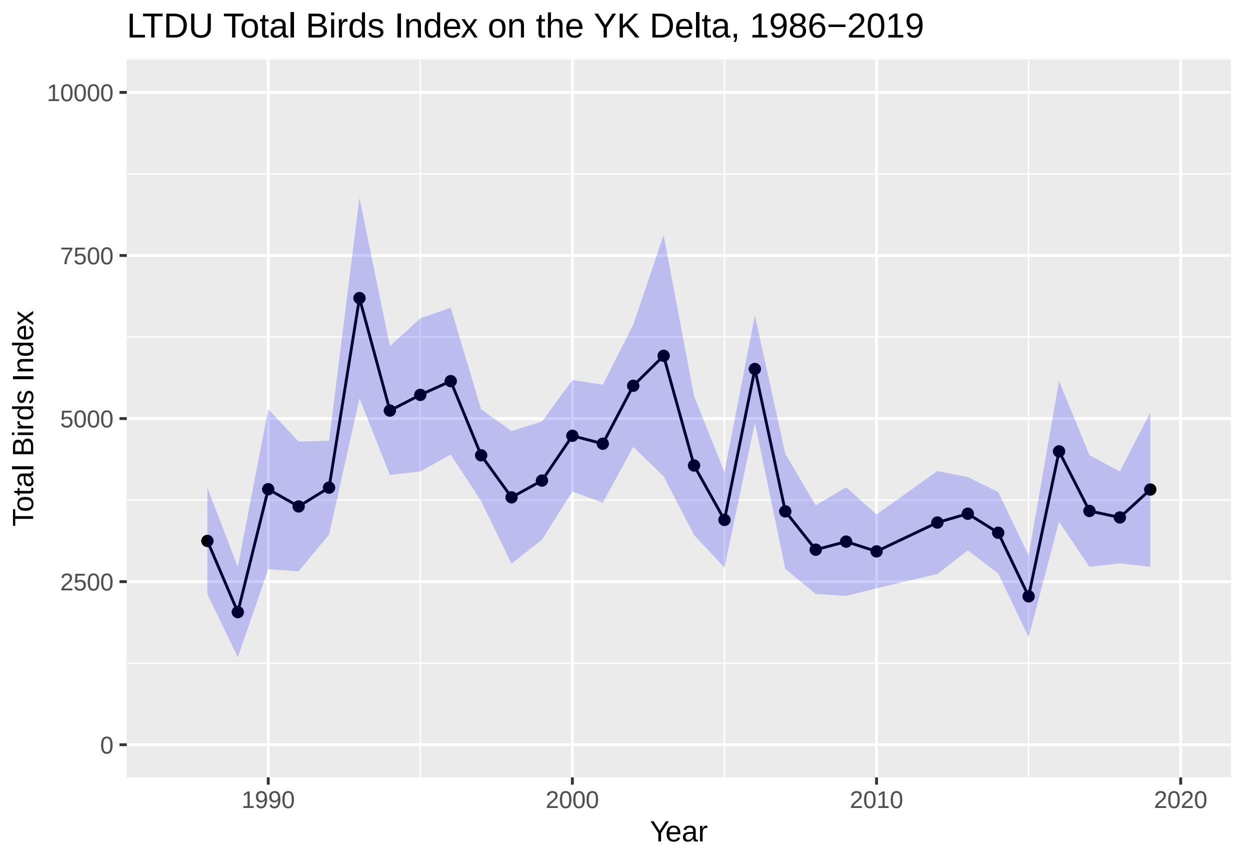
Mind-blowing! But really, what we've done here is establish our base layer and our aesthetics. Here, the x axis defined as Year, and the y axis as indicated total. We haven't told ggplot how to actual show us the data, but since the function is modular (like designing a modular home), we can start to tack on our layers.

```
YKDHistoric$combined %>%
  filter(Species == "LTDU") %>%
  ggplot(aes( x = Year, y = itotal)) +
    geom_point() +
    geom_line()
```



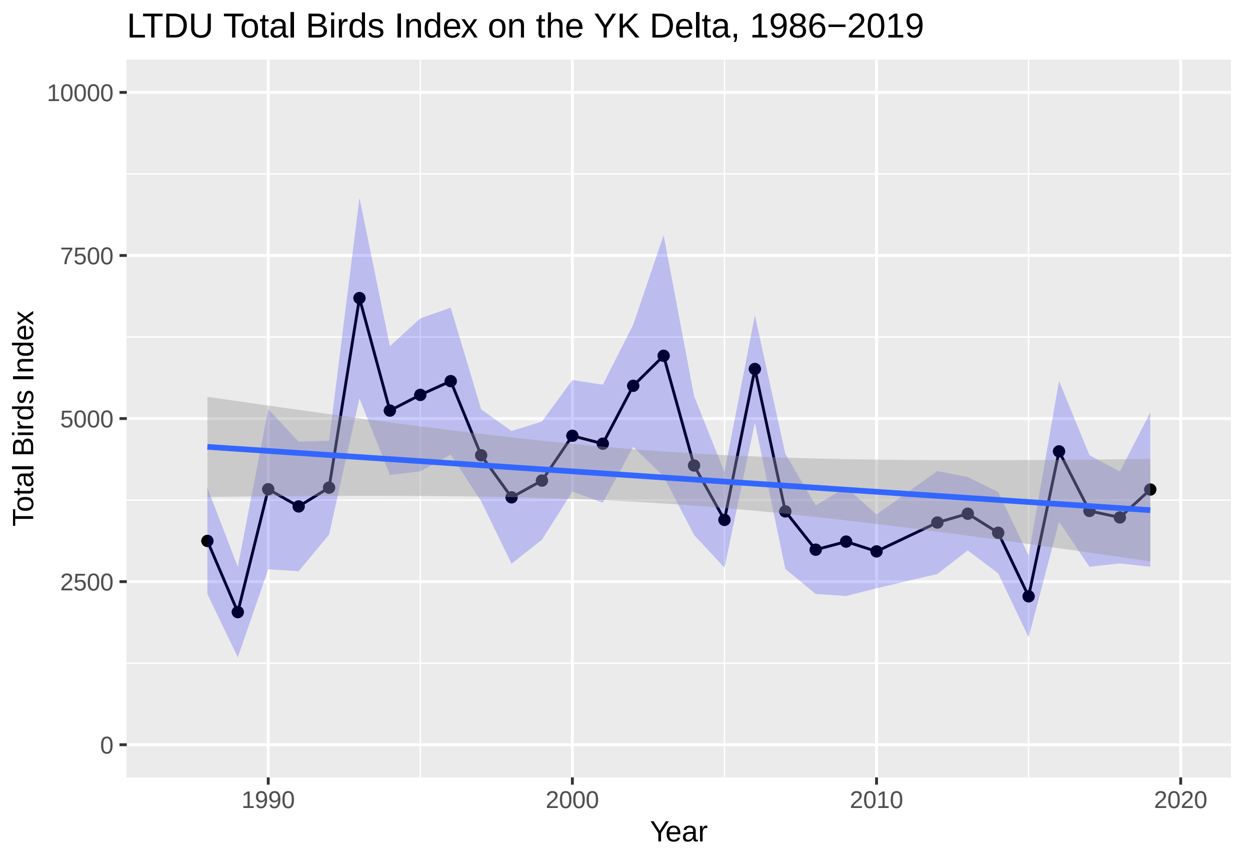
Here we told ggplot to now display our data as both points (geom_point) and lines (geom_line). Now we can fix the axes and get those confidence intervals on there. We will use a fill color and alpha value to set the transparency. We can also set the labels using labs() while we are at it (very intuitively).

```
duck.plot =
  YKDHistoric$combined %>%
  filter(Species == "LTDU") %>%
  ggplot(aes( x = Year, y = total)) +
    geom_point() +
    geom_line() +
    geom_ribbon(aes(ymin = total - 1.96 * total.se, ymax = total + 1.96 * total.se),
                fill = "blue",
                alpha = 0.2) +
    coord_cartesian(xlim=c(1987,2020), ylim=c(0, 10000)) +
    labs(title="LTDU Total Birds Index on the YK Delta, 1986-2019", x="Year", y="Total Birds Index")

print(duck.plot)
```



So far so good. Since a ggplot functions as an R object, we can store it in R (here we saved it as duck.plot) and add onto it with ease. The only difference is when we store the object it won't automatically plot, so we have to explicitly make it print(). Now let's imagine we want to throw a simple linear regression onto it.

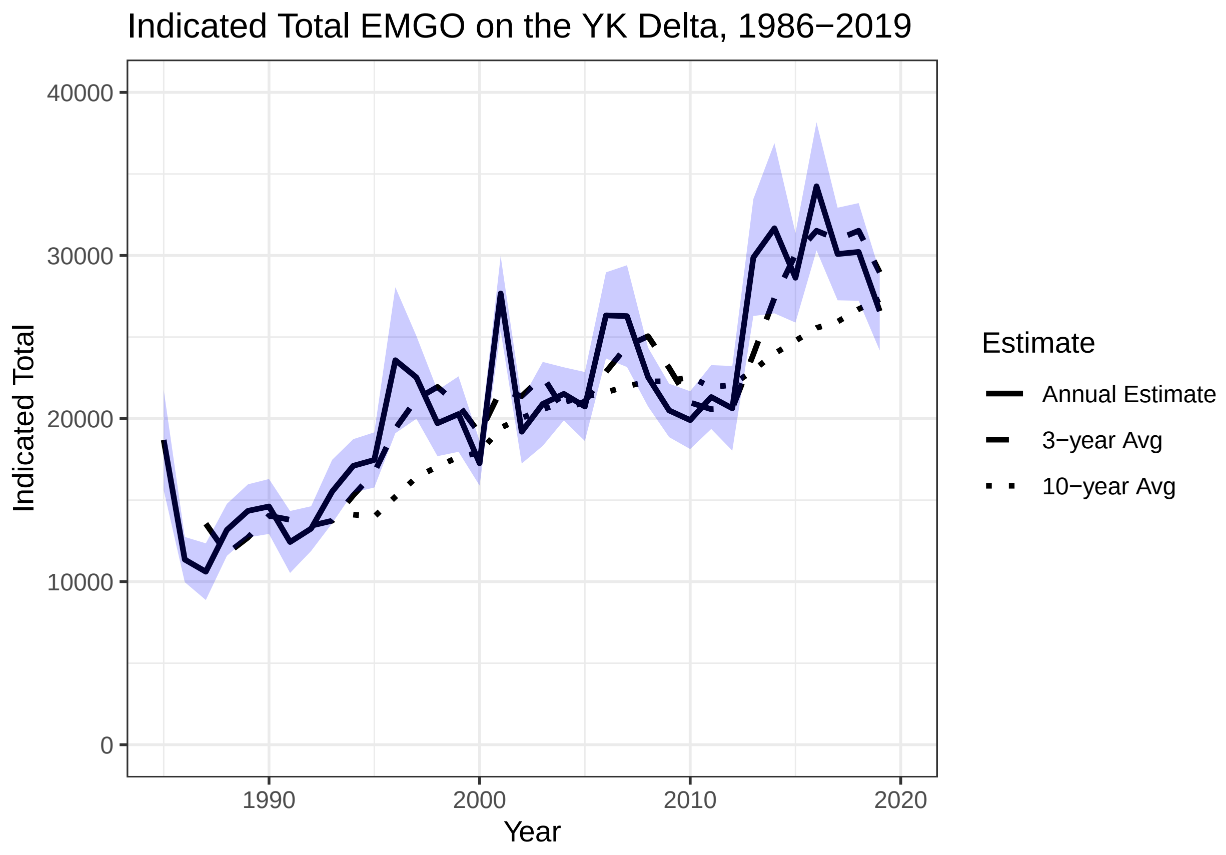
```
duck.plot + geom_smooth(method='lm')
```



Pretty slick. Let's see that killer table we made earlier now in plot form.

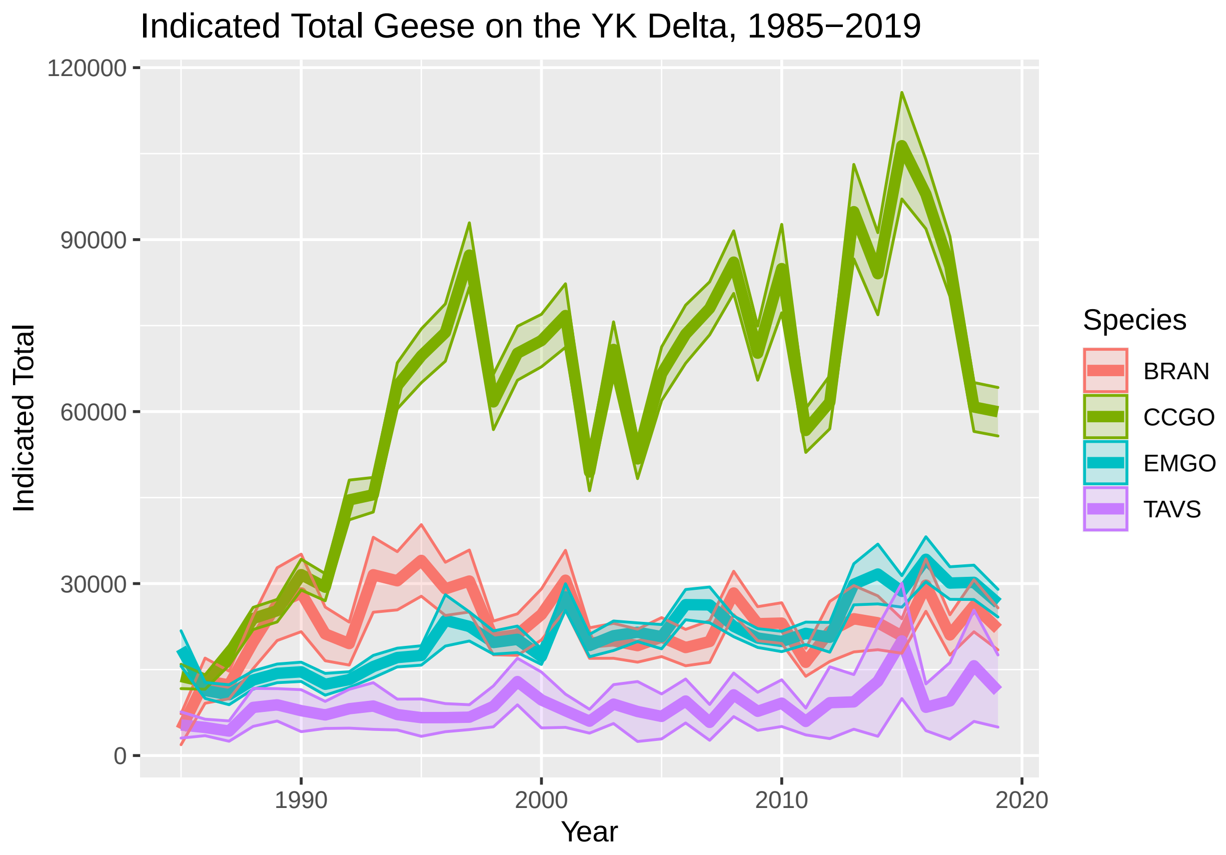
```
emgo =

  #tidy it up and pivot the estimates out

  YKGHistoric$combined %>%

  filter(Species=="EMGO") %>%

  select(Year, itotal, itotal.se) %>%

  mutate(avg3=rollapply(itotal,3,mean,align='right',fill=NA)) %>%

  mutate(avg10=rollapply(itotal,10,mean,align='right',fill=NA)) %>%

  pivot_longer(cols=c("itotal","avg3", "avg10"), names_to = "index", values_to = "estimate")

  #now plot it

  emgo.base =

  ggplot(emgo, aes(x = Year)) +

  #change the linetype (dash spacing) by index type
  geom_line(aes(y = estimate, linetype=index), size = 1) +

  #add the confidence intervals
  #note these are specific to itotal, so we have to filter what we pass

  geom_ribbon(data = filter(emgo, index=="itotal"), aes(ymin = estimate - 1.96 * itotal.se, ymax = esti
              fill = "blue",
              alpha = 0.2) +

  coord_cartesian(xlim=c(1985,2020), ylim=c(0, 40000)) +

  labs(title="Indicated Total EMGO on the YK Delta, 1986-2019", x="Year", y="Indicated Total") +

  #make a keen legend to sort out the mess

  scale_linetype_manual(name = "Estimate",
                        values = c(1,2,3),
                        labels = c("Annual Estimate", "3-year Avg", "10-year Avg"),
                        limits = c("itotal", "avg3", "avg10")) +

  #take the gray tiles off of the back

  theme_bw()


  print(emgo.base)
```

```
## Warning: Removed 11 row(s) containing missing values (geom_path).
```



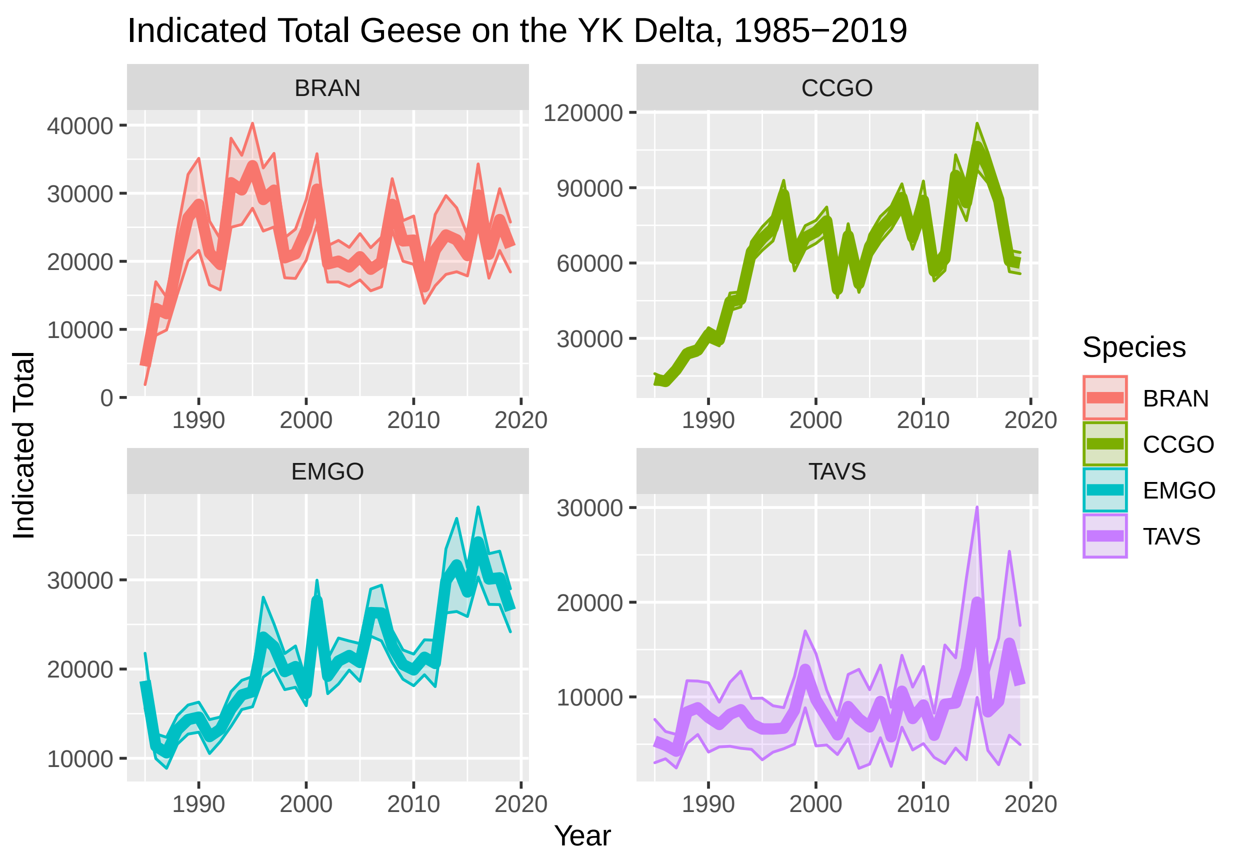
One powerful feature of ggplot is its ability to handle factors (or character strings that can be easily converted to factors). It can process a multi-species or multi-area data set and produce a string of plots in no time.

```
multi =


  YKGHistoric$combined %>%

  filter(Species %in% c("EMGO", "BRAN", "CCGO", "TAVS")) %>%

  select(Year, Species, itotal, itotal.se)


  multi.plot =
    ggplot(multi, aes(x = Year, y = itotal, color = Species)) +
    geom_line(size=2) +
    geom_ribbon(aes(ymin = itotal - 1.96 * itotal.se, ymax = itotal + 1.96 * itotal.se,
                    color=Species,
                    fill = Species),
                    alpha=.2)  +
    labs(title="Indicated Total Geese on the YK Delta, 1985-2019", x="Year", y="Indicated Total")


print(multi.plot)
```



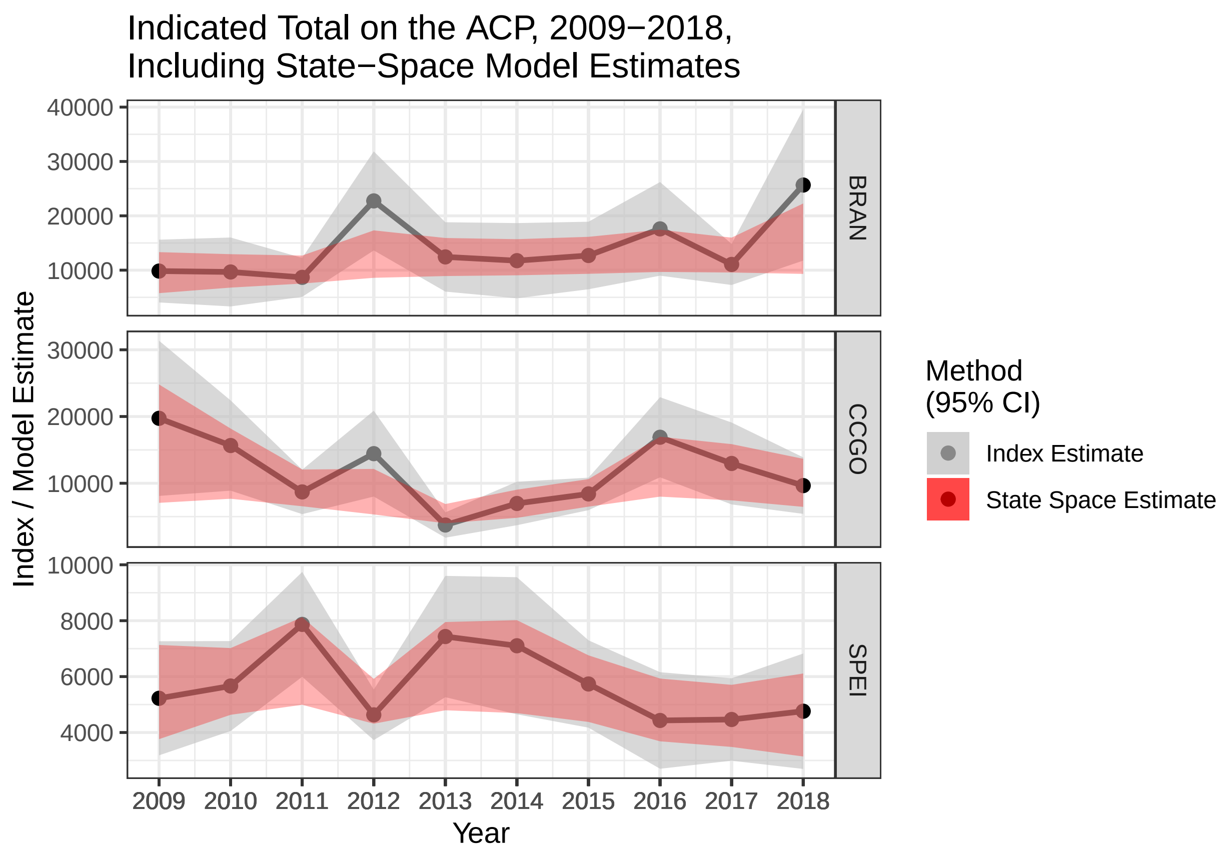
The neat part here is that when we have a factor (or a character that can be automatically converted to one, in this case) such as Species, we can explicitly tell ggplot to map it as an aesthetic in the aes() definition right away. Then later when we want to change colors, linetypes, sizes, etc., we just tell it to change per our aesthetic. We even get a legend made automatically. But what if we wanted to split these out by our Species factor, each to their own figure?

```
multi.plot +
  facet_wrap(~Species, scales = "free")
```



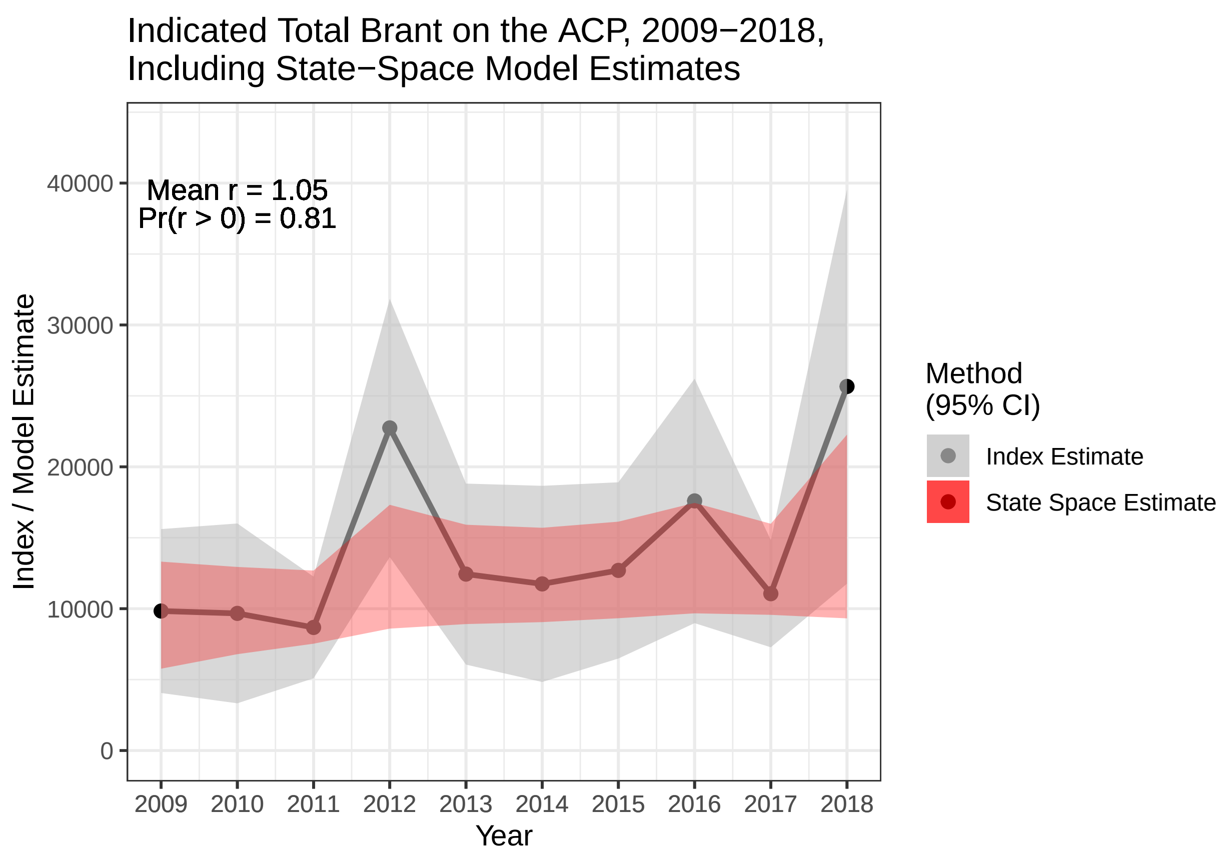
We told it to create separate figures (facets) wrapped logically on our screen based on how big or small we shape our plot window. These were separated by species (~Species), and we had to tell ggplot to allow the scales for each species to be "free" and not all based on the highest value of the most abundant species. The alternative (and default) is "fixed." Below we will use facet_grid to tell ggplot to change the number of rows to reflect the number of Species it counts. I also use a few more style elements here to expand the x axis values and produce the legend.

```
ssm = read.csv("./Plot/StateSpaceEst.csv",
               header = TRUE,
               stringsAsFactors = FALSE)

ssm = ssm %>%
  pivot_longer(cols=c("N", "index"),
               names_to = "method",
               values_to = "estimate")

ssm.plot=
  ggplot(ssm, aes(x = Year, y = estimate, fill = method)) +
  geom_line(data = ssm %>% filter(method == "index"), size=1) +
  geom_point(data = ssm %>% filter(method == "index"), size=2) +
  geom_ribbon(data = ssm %>% filter(method == "index"),
              aes(ymin = l95index, ymax = u95index),
                alpha = 0.6) +

  geom_ribbon(data = ssm %>% filter(method == "N"),
              aes(ymin = lower95, ymax = upper95),
                alpha = 0.3) +

  facet_grid(Species~., scales = "free") +


  labs(title="Indicated Total on the ACP, 2009-2018,\nIncluding State-Space Model Estimates",
        x="Year", y="Index / Model Estimate") +

  scale_fill_manual(name = "Method\n(95% CI)",
                    values=c("grey", "red") ,
                    labels = c("Index Estimate", "State Space Estimate")) +

  scale_x_continuous("Year", labels = as.character(ssm$Year), breaks = ssm$Year) +

  theme(axis.text.x = element_text(face="bold", color="black")) +

  theme_bw()


print(ssm.plot)
```



Or, for a single species:

```
ssm.single = ssm %>% filter(Species == "BRAN")

ssm.single.plot =
  ggplot(data = ssm.single,
         aes(x = Year, y = estimate, fill = method)) +

  geom_line(data = ssm.single %>% filter(method == "index"), size=1) +

  geom_point(data = ssm.single %>% filter(method == "index"), size=2) +

  geom_ribbon(data = ssm.single %>% filter(method == "index"),
              aes(ymin = l95index, ymax = u95index),
                alpha = 0.6) +

  geom_ribbon(data = ssm.single %>% filter(method == "N"),
              aes(ymin = lower95, ymax = upper95),
                alpha = 0.3) +

  labs(title="Indicated Total Brant on the ACP, 2009-2018,\nIncluding State-Space Model Estimates",
       x="Year", y="Index / Model Estimate") +

  scale_fill_manual(name = "Method\n(95% CI)",
                    values=c("grey", "red") ,
                    labels = c("Index Estimate", "State Space Estimate")) +

  scale_x_continuous("Year",
                     labels = as.character(ssm.single$Year),
                     breaks = ssm.single$Year) +

  theme(axis.text.x = element_text(face="bold", color="black")) +

  coord_cartesian(ylim=c(0, 1.1 * max(ssm.single$u95index))) +

  geom_text(x=min(ssm.single$Year + 1), y=max(ssm.single$u95index),
          label=paste0("Mean r = ", round(1 + ssm.single$mean.r[1], 2))) +

  geom_text(x=min(ssm.single$Year + 1), y=.95*max(ssm.single$u95index),
          label=paste0("Pr(r > 0) = ", round(ssm.single$p.r, 2))) +

  theme_bw()

print(ssm.single.plot)
```



Note the use of geom_text to paste together a character string and place it at a given (x,y) on the figure. Also take note of the way you can dynamically set an axis to be a little wider or more narrow than the automatic setting. We can also quickly add the associated table.

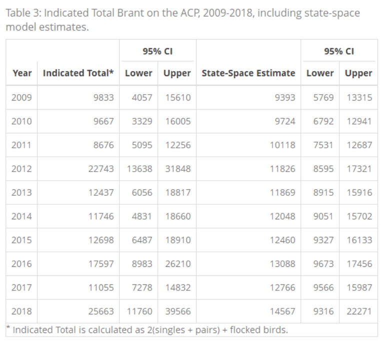


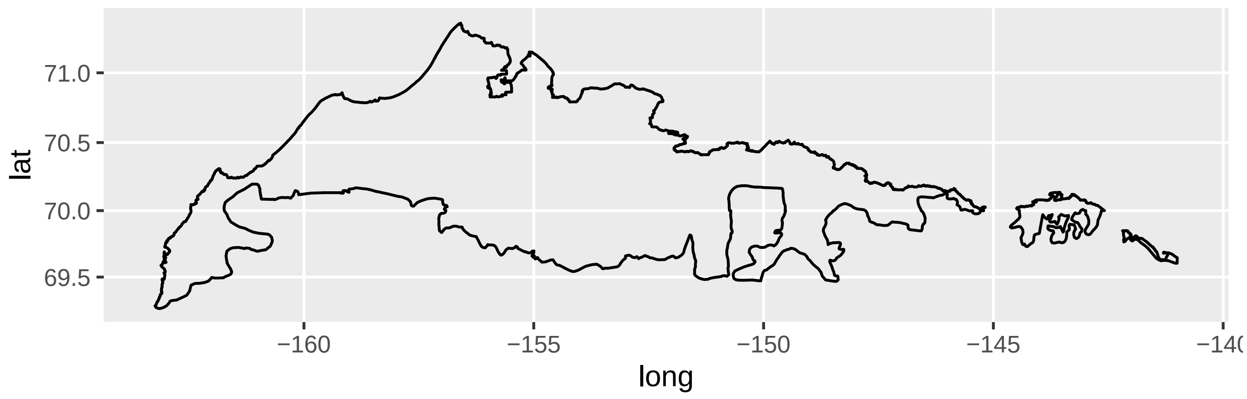
Table 3: Indicated Total Brant on the ACP, 2009-2018, including state-space model estimates.

		95% CI			95% CI	
Year	Indicated Total*	Lower	Upper	State-Space Estimate	Lower	Upper
2009	9833	4057	15610	9393	5769	13315
2010	9667	3329	16005	9724	6792	12941
2011	8676	5095	12256	10118	7531	12687
2012	22743	13638	31848	11826	8595	17321
2013	12437	6056	18817	11869	8915	15916
2014	11746	4831	18660	12048	9051	15702
2015	12698	6487	18910	12460	9327	16133
2016	17597	8983	26210	13088	9673	17456
2017	11055	7278	14832	12766	9566	15987
2018	25663	11760	39566	14567	9316	22271

* Indicated Total is calculated as 2(singles + pairs) + flocked birds.

Can ggplot plot spatial information? It can, but not necessarily easily. Remember when I said that ggplot takes a data frame rather than a vector? Well, it also takes a data frame rather than a spatial format. So to make it happen, we have to convert a shapefile or other spatial overlay into a data frame. It isn't too hairy and I've created a function (LoadMap) in AKaerial that will read in your spatial file and output it as a ggplot-able data frame. We will use it here and send it our shapefile.

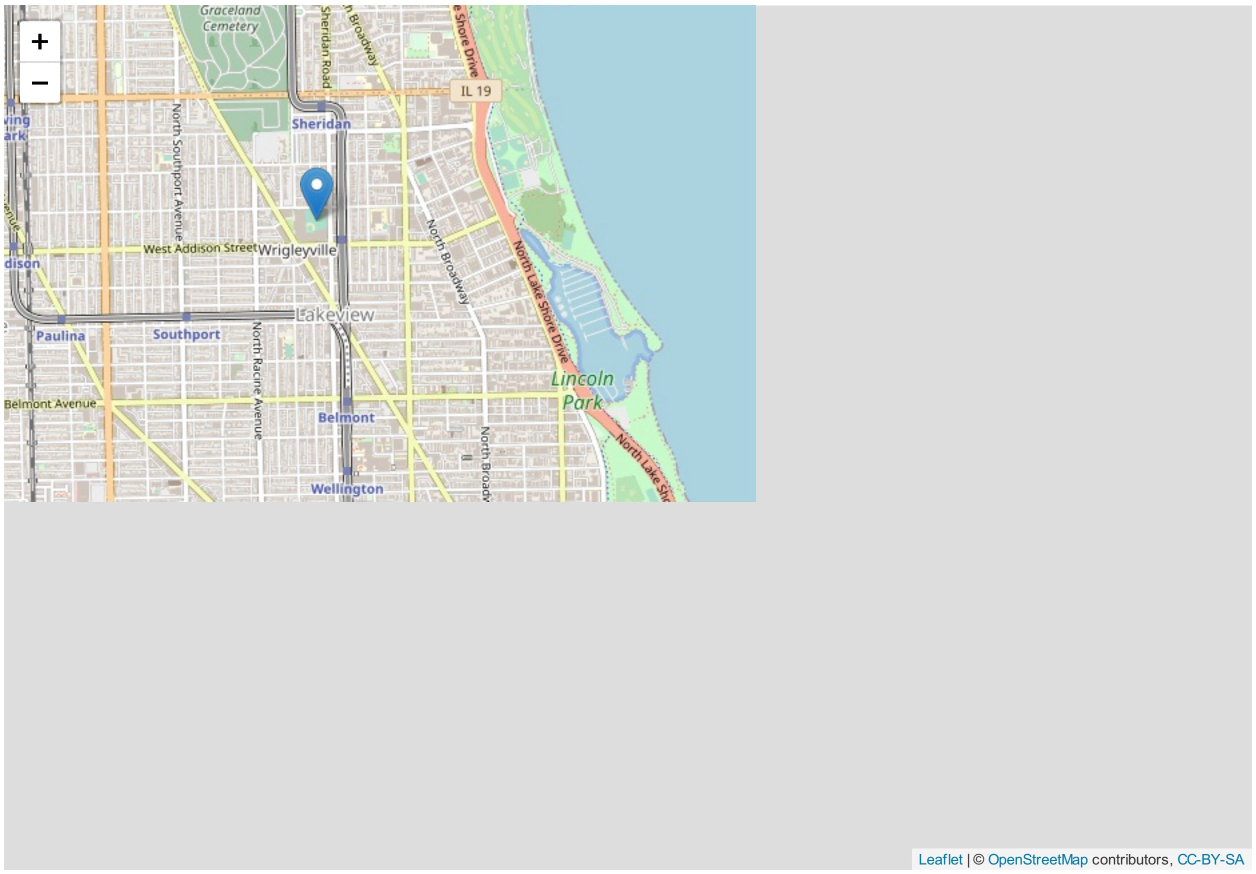
```r
map.df = AKaerial::LoadMap("./Plot/ACP_2007to2019_StudyArea.shp")

ggplot() +

      #geom_path here tells ggplot to connect the dots of my spatial file,
      #since polygons in their simplest form a just a series of vertices.

      geom_path(data=map.df, aes(long,lat,group=group)  ) +

      geom_path(color="black") +

      #now set a reasonable bounding box
      coord_map(xlim=c(min(map.df$long), max(map.df$long)),
                ylim=c(min(map.df$lat), max(map.df$lat)))
```



This isn't spectacular either, maybe a step up from base R spatial plots. We can add functionality to it and trace other polygons and fill in solid colors, but for spatial visualization, nothing really compares to leaflet (yet).

Leaflet is a powerful JavaScript library that creates user-friendly maps for almost any occasion. It isn't explicitly R code, so the package leaflet was created to access the JavaScript functionality through R. You can see how powerful it is with even a basic map.

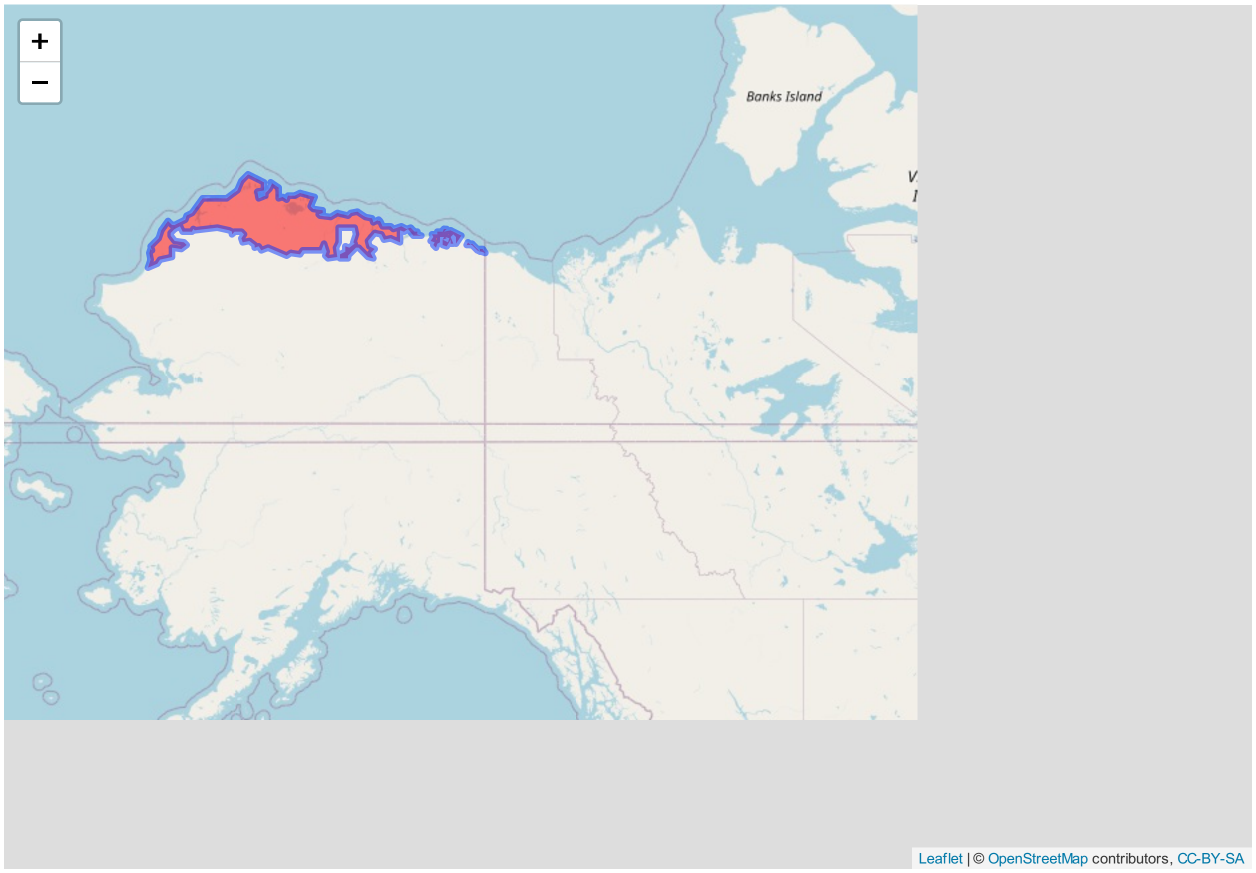
```r
leaflet() %>%
  addTiles() %>%  # use the default base map tiles
  addMarkers(lng=-87.6553, lat=41.9484,
             popup="Wrigley Field")
```



Since leaflet maps are html widgets, they won't appear well or function in this pdf document. On your computer screen, though, you can click, drag, and zoom. They also work great in interactive html documents (much like datatables and kables).

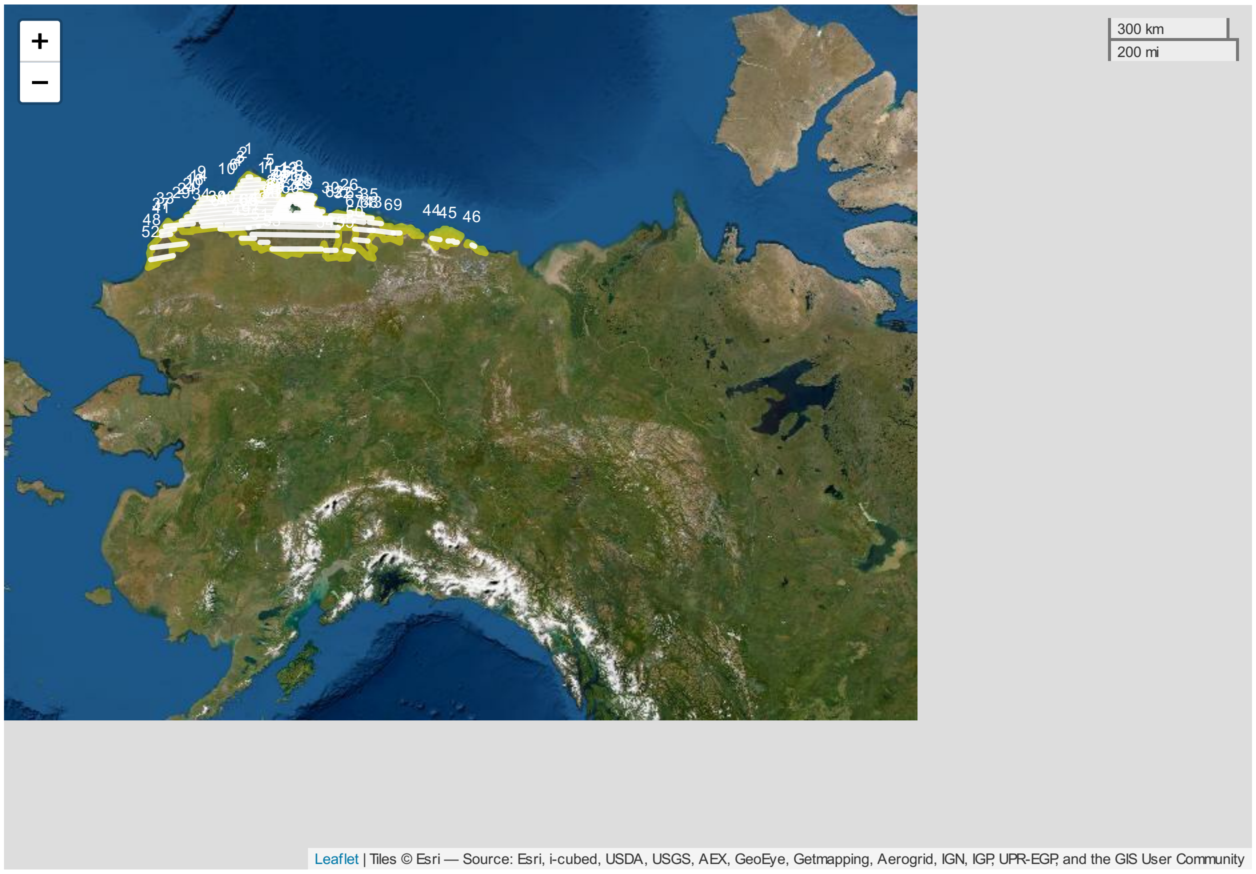
In addition to just displaying a place on a map, leaflet can take your spatial data files and make glorious visualizations with them.

```
leaflet() %>%
  addTiles() %>%
  addPolygons(data=map.proj,
              fillColor = "red",
              fillOpacity = .5)
```



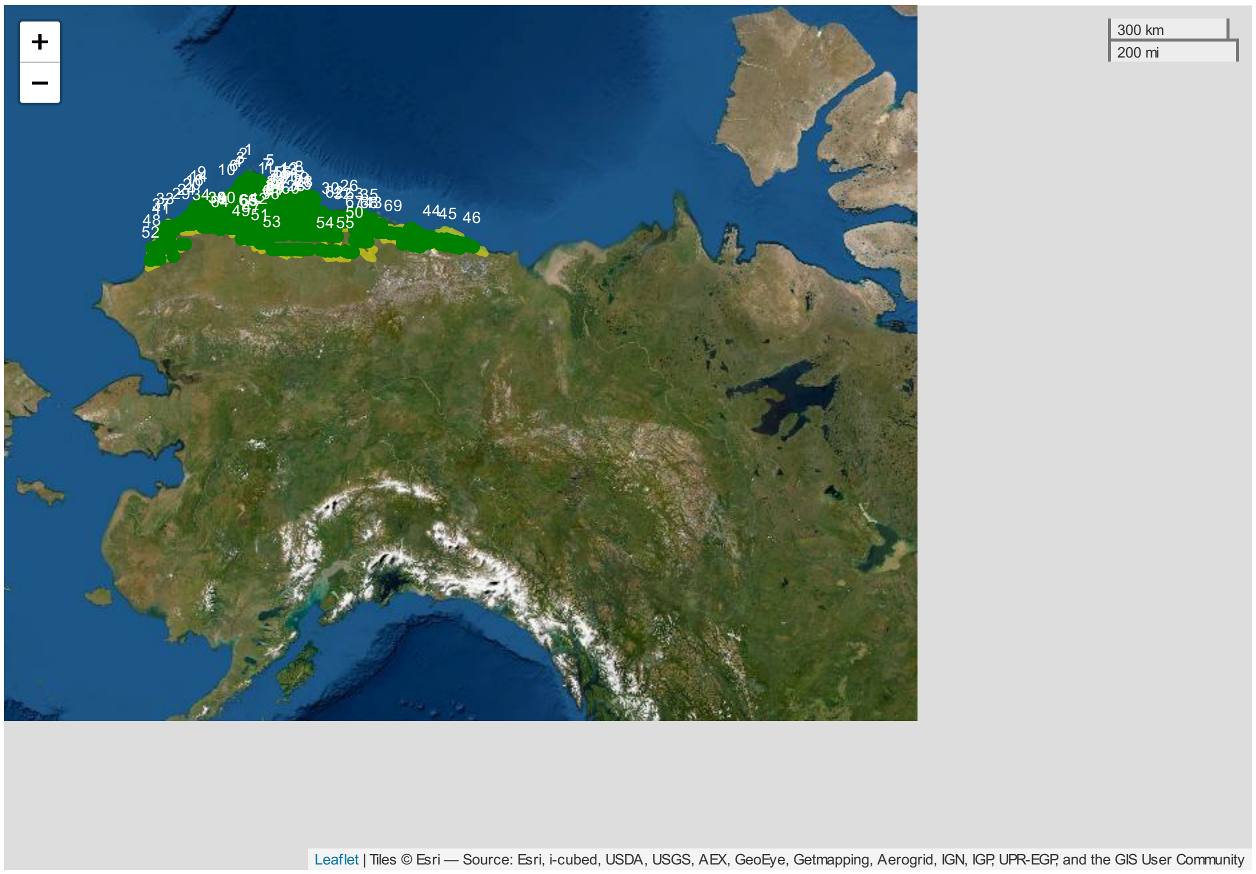
We can add almost any spatial files to it.

```
#read and project our lines shp file

lines= readOGR("./Plot/ACP_2019_Transects.shp",
  layer = "ACP_2019_Transects",
  verbose=FALSE)

lines.proj <- spTransform(lines, "+proj=longlat +ellps=WGS84 +datum=WGS84")

#map it

leaflet() %>%

  addPolygons(data=map.proj,
              color = "yellow",
              fill = FALSE,
              fillOpacity = .5) %>%

  #add our lines here
  addPolylines(data=lines.proj,
               color="white",
                 weight=4,
                 opacity=.9,
                 label=~TRANSID,
                 popup = paste("Transect: ", lines.proj$TRANSID, "<br>",
                                       "Length: ", lines.proj$LENGTH)) %>%

  #scale for...scale
  addScaleBar() %>%

  #cool satellite imagery base map
  addProviderTiles("Esri.WorldImagery") %>%

  #add labels
  addLabelOnlyMarkers(data = fortify(lines.proj) %>%
                          filter(order == 1) %>%
                          mutate(new.id = as.numeric(id) + 1),
                        label = ~as.character(new.id),
                        labelOptions = labelOptions(noHide = T,
                                                    direction = 'top',
                                                    textOnly = T,
                                                    style = list(
                                                      "color" = "white",
                                                      "font-size" = "12px")))
```


+
−
300 km
200 mi
Leaflet | Tiles © Esri — Source: Esri, i-cubed, USDA, USGS, AEX, GeoEye, Getmapping, Aerogrid, IGN, IGP, UPR-EGP, and the GIS User Community

We got a little fancy there by extracting the start point of each of our transect lines, then adding a text label to the point with the transect id. Now let's add some observation data:

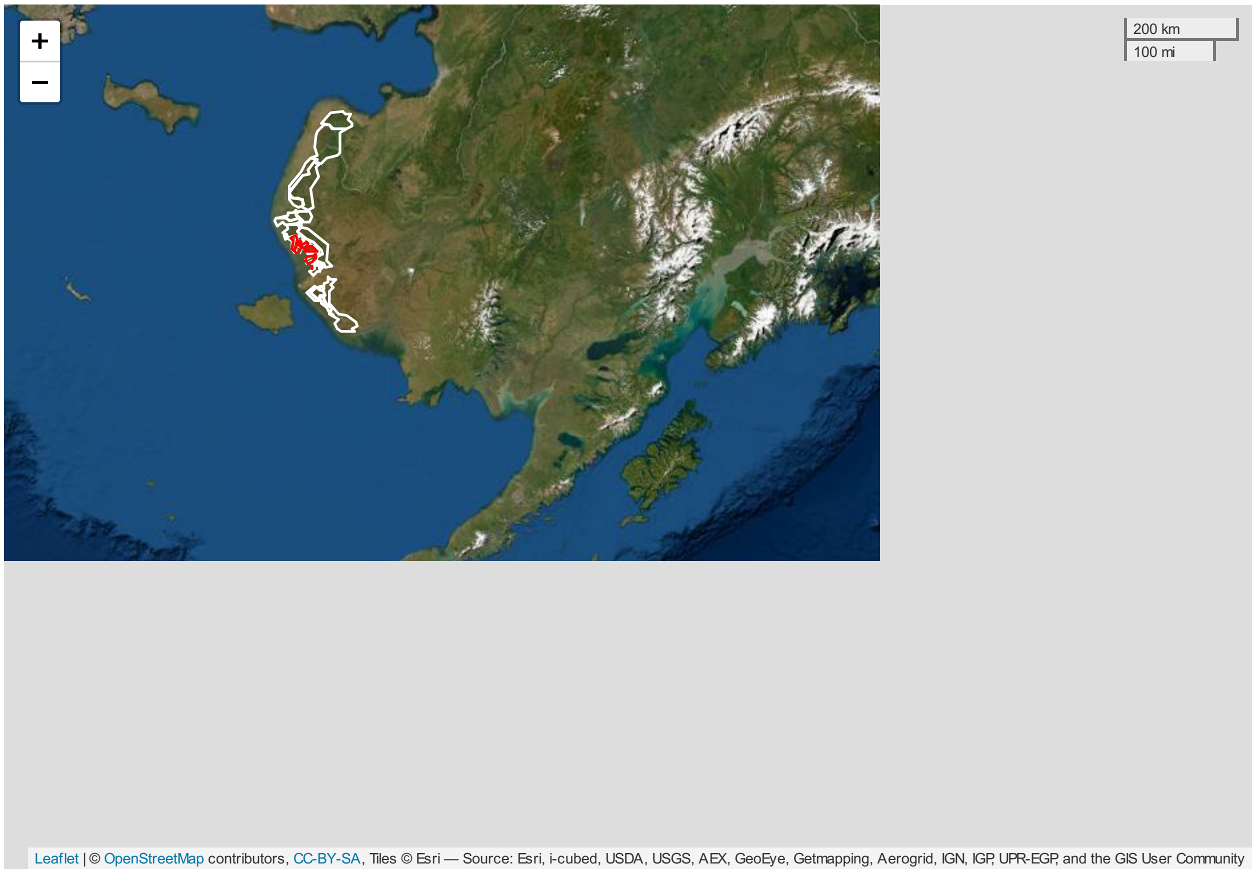
```
#read our observations

obs = read.csv("./Plot/ACP_2019_QCObs_HWilson.csv",
               header=TRUE,
               stringsAsFactors = FALSE)

#define the spatial x,y for plotting
coordinates(obs)=~Lon+Lat

#map it

leaflet() %>%

  addPolygons(data=map.proj,
              color = "yellow",
              fill = FALSE,
              fillOpacity = .5) %>%

  #add our lines here
  addPolylines(data=lines.proj,
               color="white",
                 weight=4,
                 opacity=.9,
                 label=~TRANSID,
                 popup = paste("Transect: ", lines.proj$TRANSID, "<br>",
                               "Length: ", lines.proj$LENGTH)) %>%

  #scale for...scale
  addScaleBar() %>%

  #cool satellite imagery base map
  addProviderTiles("Esri.WorldImagery") %>%

  #add labels
  addLabelOnlyMarkers(data = fortify(lines.proj) %>%
                        filter(order == 1) %>%
                        mutate(new.id = as.numeric(id) + 1),
                      label = ~as.character(new.id),
                      labelOptions = labelOptions(noHide = T,
                                                  direction = 'top',
                                                  textOnly = T,
                                                  style = list(
                                                    "color" = "white",
                                                    "font-size" = "12px"))) %>%

  #add the point data
  addCircleMarkers(data=obs,
                   radius = 5,
                   color = "green",
                   stroke = FALSE,
                   fillOpacity = 1,
                   popup= paste(obs$Observer, "<br>", obs$Species,
                                "<br>", obs$Obs_Type, "<br>", "n = ",obs$Num,
                                "<br>", "Transect ", obs$Transect, "<br>"))
```


+
−
300 km
200 mi
Leaflet | Tiles © Esri — Source: Esri, i-cubed, USDA, USGS, AEX, GeoEye, Getmapping, Aerogrid, IGN, IGP, UPR-EGP, and the GIS User Community

You can probably imagine many ways to use this type of map…at the very least, in QA/QC. For example, scroll to the east and look at all of the green points that fall out of the study area! In this case, though, that was actually by design for a corollary project on the refuge, but you can learn a lot from even a basic map. We can also accomplish all of this with GeoJSON files.

GeoJSON stands for Geographic Javascript Object Notation and is an encoding method for a variety of spatial data types. Here we will use some polygon GeoJSON containers from the Yukon-Kuskokwim Delta.

```r
ykdair <- geojson_read("./Plot/YKD_DesignStrata.geojson",  what = "sp")
ykdnest <-  geojson_read("./Plot/NestPlotStudyAreaBoundary.geojson",  what = "sp")

leaflet() %>%
  addTiles() %>%
  addPolygons(data=ykdair,
              fillColor="blue",
              fillOpacity=0,
              stroke=TRUE,
              color="white",
              opacity=1,
              weight=2) %>%
  addPolygons(data=ykdnest,
              fillOpacity=0,
              stroke=TRUE,
              color="red",
              opacity=1,
              weight=2) %>%
  addScaleBar() %>%
  addProviderTiles("Esri.WorldImagery")
```


+
−
200 km
100 mi
Leaflet | © OpenStreetMap contributors, CC-BY-SA, Tiles © Esri — Source: Esri, i-cubed, USDA, USGS, AEX, GeoEye, Getmapping, Aerogrid, IGN, IGP, UPR-EGP, and the GIS User Community

We read in 2 polygon layers describing the overall aerial survey coverage and subsetted nest plot project coverage. Zoom way in where the red and white overlap. Notice any potential QA/QC issues?

5.0.0 Working with Dates

One common thread to almost every project we have ever conducted in MBM is that we want to include some temporal aspect in the related data. This could be as simple as just a year, or as complicated as year-month-day-hours-minutes-seconds. Just like every other topic we've discussed, base R provides crude and sometimes clunky ways to deal with dates, and contributed packages enhance and refine specialized methods for manipulation of date-time data.

5.1.0 Dates and Times in Base R

The basic format for a date in R is YYYY-MM-DD. You can grab today's date or date and time using Sys.Date() or date() commands.

```
today = Sys.Date()

today
```

```
## [1] "2020-08-11"
```

```
str(today)
```

```
##  Date[1:1], format: "2020-08-11"
```

```
#the unit here is the day, so you can do basic calculations-

year.ago = Sys.Date() - 365

year.ago
```

```
## [1] "2019-08-12"
```

```
today - year.ago
```

```
## Time difference of 365 days
```

```
#this gives a longer representation, but as a character string, so no easy calculations

today.long = date()

str(today.long)
```

```
##  chr "Tue Aug 11 14:06:28 2020"
```

```
#today.long - 365 (this won't work!)
```

Base R has a Date format that can convert many types of dates into something you can manipulate with basic calculations. Let's create a fake temperature reading data set.

```
temps = data.frame ("Date"= c("2020-7-4", "2020-7-5", "2020-7-6", "2020-7"),
                    "Temp"= c(50, 51, 55, 50),
                    stringsAsFactors = FALSE)

str(temps)
```

```
## 'data.frame':    4 obs. of  2 variables:
##  $ Date: chr  "2020-7-4" "2020-7-5" "2020-7-6" "2020-7"
##  $ Temp: num  50 51 55 50
```

If we want R to be able to manipulate the Date column as a date, we can use as.Date() to change the data type. Notice that the 4th entry is not in our standard format; it is missing the month or day. Also the first 3 are not quite in the right format; they are in single digit month and day. When we call as.Date, it will know what to do with the single digits, but it will fill an NA for the one missing the month or day since it doesn't want to guess what you meant. We can fix this with a little paste command.

```
temps$Date[4] = paste(temps$Date[4], "7", sep = "-")

temps$Date = as.Date(temps$Date)

str(temps)
```

```
## 'data.frame':    4 obs. of  2 variables:
##  $ Date: Date, format: "2020-07-04" "2020-07-05" ...
##  $ Temp: num  50 51 55 50
```

```
range = max(temps$Date) - min(temps$Date)

range
```

```
## Time difference of 3 days
```

One nifty trick with dates, once they are recognized as Date data types, is that you can reformat them to almost any possible date format. This is done using format(), a generic function that reformats data types. If you pass it a date, it calls the funtion strptime() to reformat the date using acceptable values. You establish the format using a character string argument after your date argument. See ?strptime for all of the possibilities, but here are a few-

```
format(temps$Date, "%A")   #day
```

```
## [1] "Saturday" "Sunday"   "Monday"   "Tuesday"
```

```
format(temps$Date, "%a, %A") #abbreviated day, day
```

```
## [1] "Sat, Saturday" "Sun, Sunday"   "Mon, Monday"   "Tue, Tuesday"
```

```
format(temps$Date, "%a, %w") #abbreviated day, numeric day of the week
```

```
## [1] "Sat, 6" "Sun, 0" "Mon, 1" "Tue, 2"
```

```
format(temps$Date, "%W-%w")  #week of the year - numeric day of the week
```

```
## [1] "26-6" "26-0" "27-1" "27-2"
```

```
format(temps$Date, "%j")  #day of the year
```

```
## [1] "186" "187" "188" "189"
```

```
format(temps$Date, "%A, %B %d, %Y")  #day, character month, numeric day of month, 4 digit year
```

```
## [1] "Saturday, July 04, 2020" "Sunday, July 05, 2020"
## [3] "Monday, July 06, 2020"   "Tuesday, July 07, 2020"
```

You can also use the format specification in reverse to tell R what format your field is in that you want converted to a Date type, if you don't have the standard-

```
as.Date("2020, 200", format = "%Y, %j")
```

```
## [1] "2020-07-18"
```

5.2.0 Dates and Times in Contributed Packages

The main (useful) date-time contributed package in R is lubridate, so make sure you have it installed and loaded. We will also get back our fake aerial survey data to toy with.

```
install.packages("lubridate")
```

```
library(lubridate)
```

```
##
## Attaching package: 'lubridate'

## The following object is masked from 'package:base':
##
##     date
```

```
library(tidyverse)

aerial=read.csv(
  "https://raw.githubusercontent.com/cfrost3/MBM_R_Short_Course/master/BLSC_2018_RawObs_Fake.csv",
  header=TRUE,
  stringsAsFactors=FALSE)

aerial.tidy = aerial %>%
 filter(!(Species %in% c("START", "ENDPT"))) %>%  #remove start and end points
 select(Year, Month, Day, Species, Num)  #filter to just 5 columns for illustration


str(aerial.tidy)
```

```
## 'data.frame':    519 obs. of  5 variables:
##  $ Year   : int  2018 2018 2018 2018 2018 2018 2018 2018 2018 2018 ...
##  $ Month  : chr  "6" "6" "6" "6" ...
##  $ Day    : int  15 15 15 15 15 15 15 15 15 15 ...
##  $ Species: chr  "SCAU4" "LTDUF4" "BLSC4" "SCAU2" ...
##  $ Num    : chr  "1" "1" "1" "1" ...
```

```
table(aerial.tidy$Month)
```

```
##
##    6 June
##  516    3
```

Looks like Month is a little messy. It looks like we meant to record all observations in June, so let's just overwrite it.

```
aerial.tidy$Month = 6

str(aerial.tidy)
```

```
## 'data.frame':    519 obs. of  5 variables:
##  $ Year   : int  2018 2018 2018 2018 2018 2018 2018 2018 2018 2018 ...
##  $ Month  : num  6 6 6 6 6 6 6 6 6 6 ...
##  $ Day    : int  15 15 15 15 15 15 15 15 15 15 ...
##  $ Species: chr  "SCAU4" "LTDUF4" "BLSC4" "SCAU2" ...
##  $ Num    : chr  "1" "1" "1" "1" ...
```

As with other tidyverse elements, lubridate tries to make function calls more intuitive. They replace Sys.Date() with today() and date() with now() to get dates and date-times.

```
today()
```

```
## [1] "2020-08-11"
```

```
now()
```

```
## [1] "2020-08-11 14:06:28 AKDT"
```

Another handy treatment of dates in lubridate are a series of functions that simply take the letters m, d, and y, and rearrange them as needed to format your string into a date object-

```
ymd("2020-07-20")
```

```
## [1] "2020-07-20"
```

```
mdy("July 20th, 2020")
```

```
## [1] "2020-07-20"
```

```
dmy("20-Jul-2020")
```

```
## [1] "2020-07-20"
```

```
ymd(20200720)  #you can even pass all numbers without formatting
```

```
## [1] "2020-07-20"
```

In our aerial data set, and in many, many others, we like to split up the columns into year, month, and day. We can fix this using lubridate and the tidyverse.

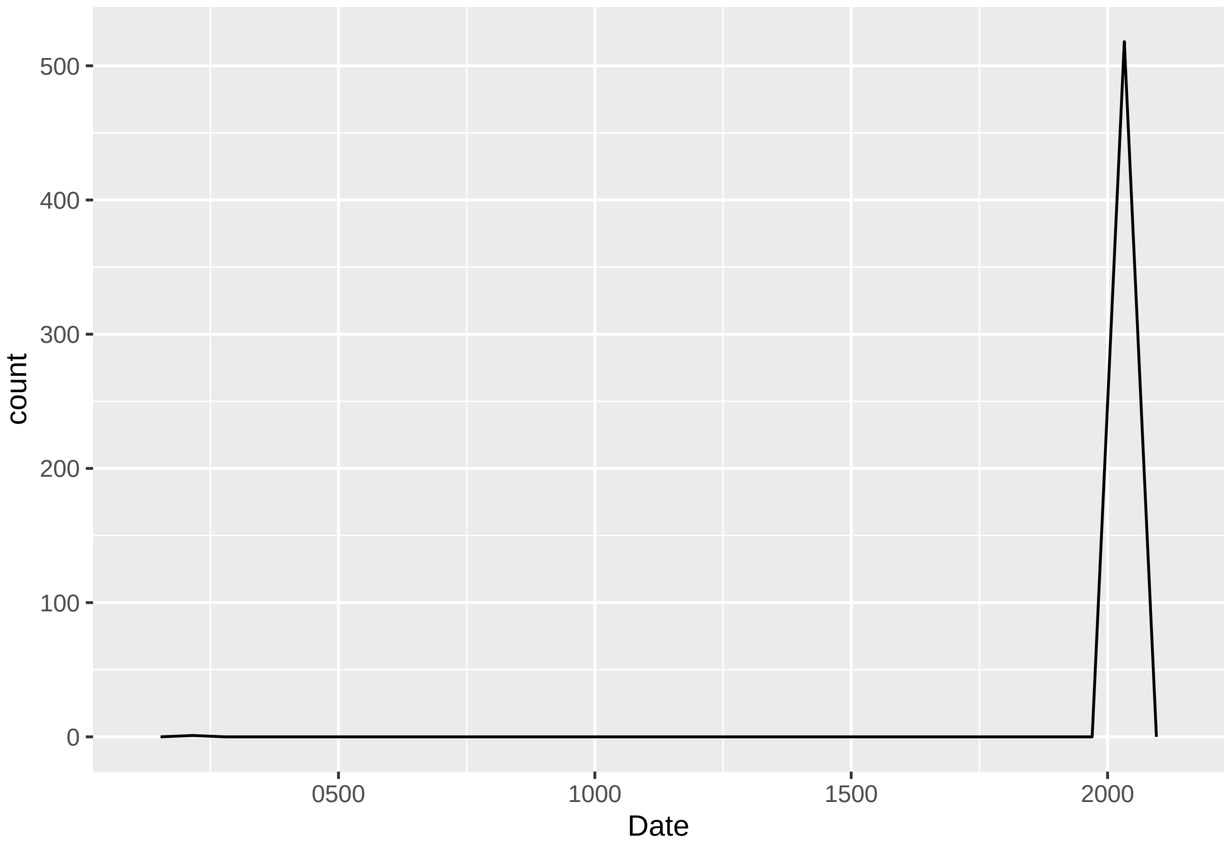
```
aerial.tidy = aerial.tidy %>%

  mutate(Date = make_date(year = Year, month = Month, day = Day))


head(aerial.tidy)
```

```
##   Year Month Day Species Num       Date
## 1 2018     6  15   SCAU4   1 2018-06-15
## 2 2018     6  15  LTDUF4   1 2018-06-15
## 3 2018     6  15   BLSC4   1 2018-06-15
## 4 2018     6  15   SCAU2   1 2018-06-15
## 5 2018     6  15   BLSC4   1 2018-06-15
## 6 2018     6  15   BLSC4   1 2018-06-15
```

Now we can plot it out using ggplot-

```
aerial.tidy %>%
  ggplot(aes(Date)) +
  geom_freqpoly()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



As usual, this uncovers another QA/QC problem.

```
unique(aerial.tidy$Date)
```

```
##  [1] "2018-06-15" "2018-06-18" "2018-06-19" "0201-06-19" "2018-06-20"
##  [6] "2018-06-22" "2018-06-23" "2018-06-24" "2018-06-25" "2018-06-26"
## [11] "2018-06-27"
```

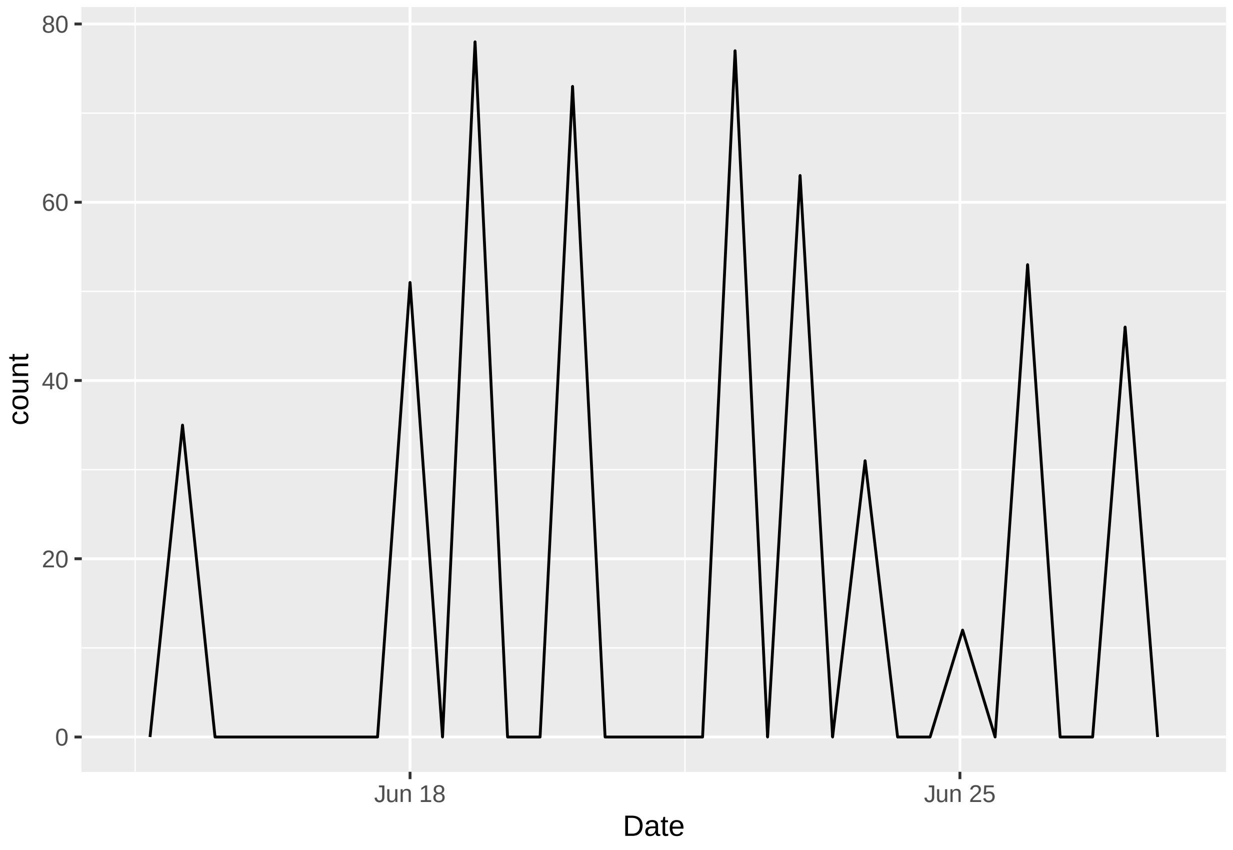
```
unique(aerial.tidy$Year)
```

```
## [1] 2018  201
```

Looks like we have a Year entered wrong. We can fix it and try again to plot.

```
aerial.tidy$Year = 2018

aerial.tidy %>%

  mutate(Date = make_date(year = Year, month = Month, day = Day)) %>%

  ggplot(aes(Date)) +

  geom_freqpoly()
```

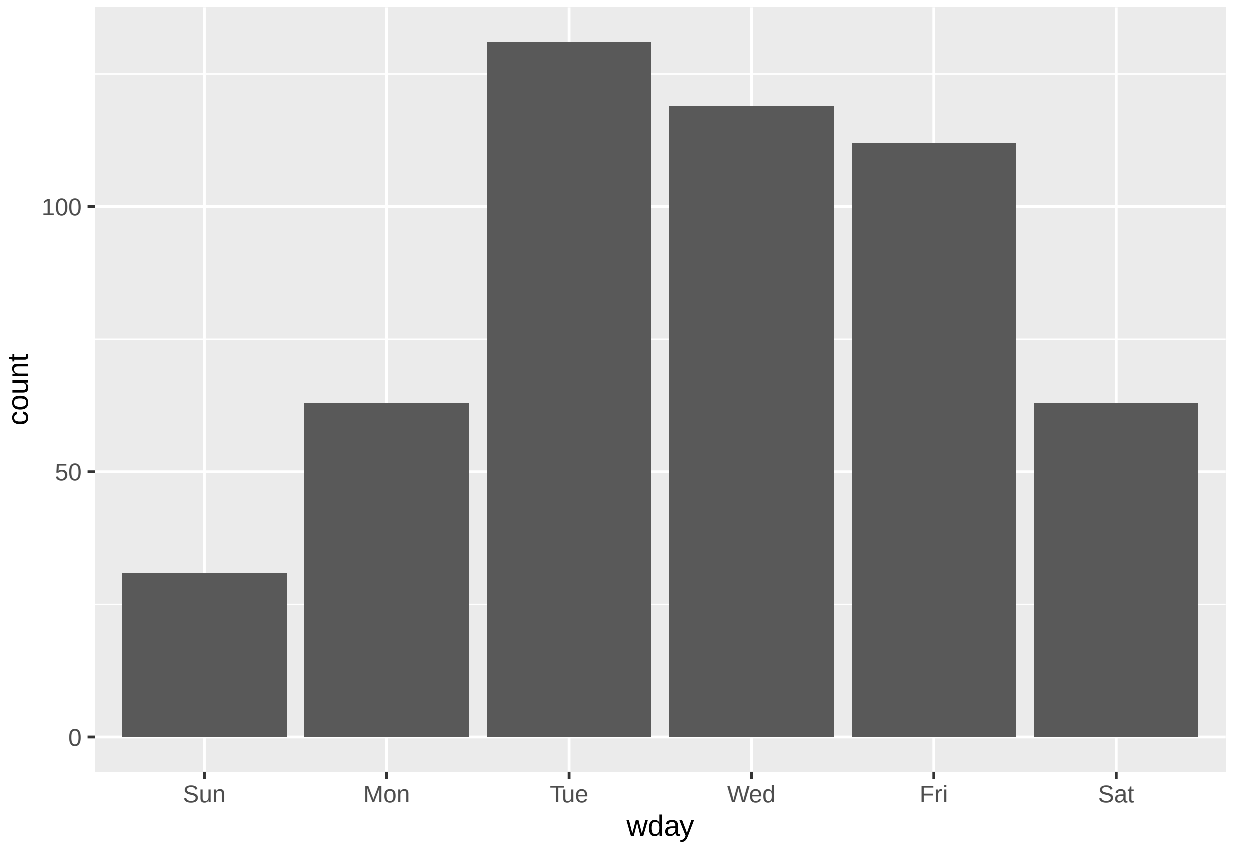
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



The lubridate package also has many functions that mimic the format() function mentioned earlier. For example, wday().

```
aerial.tidy %>%

  mutate(Date = make_date(year = Year, month = Month, day = Day)) %>%
```

```r
mutate(wday = wday(Date, label = TRUE)) %>%

ggplot(aes(x = wday)) +

  geom_bar()
```



In addition to date information, lubridate provides similar functions to deal with times, but we would use those so infrequently that I won't go over them here. For more information, look up the package lubridate.

6.0.0 Creating Functions, Conditional Statements, and Looping

Functions, conditional programming, and looping are concepts that are rooted deeply in generations of computer programming techniques. They can each be implemented separately, but, as illustrated below, often meld to enhance efficiency, cohesiveness, and structure of your code.

6.0.1 Functions

Functions are used by programmers for 2 reasons:

Reusability - If something is going to be done more than once, a function can be used to increase efficiency. In R, this works both within and across packages. I may write a function in a package pertaining to data analysis that performs an algorithm common to a given data type, but I might call a function from another package that is designed to generically tidy up data before performing the algorithm. Both cases are examples of the power of reusing functions.

To provide a meaningful abstraction - In programming speak (and in general), abstraction refers to hiding or marginalizing (abstracting) the details and inner workings of a process. A well-named and well-written function will hide the details of a process and provide a user with a clean way to accomplish a task. For comparison, think of driving a car. You don't need to be a mechanic to drive a car. The car itself is a meaningful abstraction for the process of combustion-based locomotion. The user can get in and drive with very basic input. Similarly, a function can be built to accomplish an incredibly complex task with very little input from the user.

A function has 4 primary components:

1. Name - Functions should have (mostly) unique and appropriate names. They should be short, but not too short. Descriptive, but not too descriptive. You wouldn't want to call a function ThisIsHowToAddOneOrMoreNumbersTogether when you could call it Sum and deliver the same information to the user.

2. Purpose - What is your intent? Maybe a function already exists to do it. What do you actually want it to do?

3. Arguments - Arguments are the inputs to a function. You can pass almost anything to a function, from a simple variable, to a list of multiple data types. In some cases you will use arguments as surrogates for "settings" within a function. For example, you might use the argument plot.type = "histogram" to have your function produce a histogram, but also allow plot.type = "density" to produce a density plot using the same function.

4. Return Value - Functions commonly (but not always) return values after running. In R, functions can return any R object. If you want to return a string and a numeric as 2 objects, they must be returned as a list. If you don't provide a return statement at the end of your function, R will automatically return the last calculation as the return value.

You will usually find these 4 components clearly defined in the help file for a given function.

```r
library(AKaerial)

?DataSelect
```

Here in the function `DataSelect()` from package `AKaerial`, we have the name that describes briefly what we could expect the function to do- select some data. In the Description and Details sections we see the purpose of the function- combine data layers into one summarized list. Under Arguments we see what needs to be provided for the function to work- `area`, `data.path`, `strata.path`, `transect.path`, and `threshold`. Finally, under Value we see what we can expect back- a list of 4 elements.

Does this function pass our usefulness test? Is it sufficiently reusable? Yes, we can take the spatial and data components of a project and combine them into an analysis file for any combination of design files. We expect to have to do this a lot. Does it provide meaningful abstraction? Yes, it performs dozens of transformations behind the scenes that users don't really need to worry about to get a useful file for analysis. Could they still trace the transformations? Yes, but this is probably only necessary if they want to change them someday. For normal use, abstraction works great.

Now let's create some functions from scratch. Note the syntax below closely. The name of the function needs to be provided and defined as a function. You need parentheses to define the set of arguments, even if there are none, followed by a set of curly braces that will hold the body of the function.

```r
GreetMe = function(){
 print("Hello, Chuck")
}

GreetMe()
```

```
## [1] "Hello, Chuck"
```

This is about as basic as it gets. And this doesn't really pass our function test since it isn't really a meaningful abstraction. But for illustration, we define `GreetMe` as a function with no arguments that simply tells me hello. So let's add an argument telling it who we want to greet.

```
GreetMe = function( who = "Chuck" ){

  print(paste("Hello,", who))

  }

GreetMe()
```

```
## [1] "Hello, Chuck"
```

```
GreetMe(who = "the rest of you")
```

```
## [1] "Hello, the rest of you"
```

So our function now wants the user to supply who to greet. If they don't supply it, I've told it to default to Chuck. Then all the function does is paste the name onto our greeting and print the result. But what if we wanted to allow the user to change the greeting as well as the recipient?

```
GreetMe = function( what = "Hey there,",
                    who = "Chuck" ){

  print(paste(what, who))

  }

GreetMe()
```

```
## [1] "Hey there, Chuck"
```

```
GreetMe(who = "the rest of you")
```

```
## [1] "Hey there, the rest of you"
```

```
GreetMe(what = "Hey! I can't believe it is really")
```

```
## [1] "Hey! I can't believe it is really Chuck"
```

```
GreetMe(what = "But who are", who = "the rest of you?")
```

```
## [1] "But who are the rest of you?"
```

In this example, we supplied defaults just in case our user didn't specify `what` or `who`. We can also store the return value as an R object if we want. Remember that unless we specify a return statement in the function, it defaults to the final computation.

```r
greeting = GreetMe(what = "But who are", who = "the rest of you?")
```

```
## [1] "But who are the rest of you?"
```

```r
greeting
```

```
## [1] "But who are the rest of you?"
```

Alternatively,

```r
GreetMe = function( what = "Hey there,",
                    who = "Chuck" ){

  what.who = paste(what, who)

  return(what.who)

  }

greeting = GreetMe(what = "But who are", who = "the rest of you?")

greeting
```

```
## [1] "But who are the rest of you?"
```

One subtle difference here is that we replaced the `print()` statement with the object `what.who` definition. This results in the function running silently. It no longer prints our greeting every run. We had to call our greeting object explicitly. Now let's imagine we want both the pasted version and the unpasted inputs.

```r
GreetMe = function( what = "Hey there,",
                    who = "Chuck" ){

  what.who = paste(what, who)

  return(list("what.who" = what.who,
              "what" = what,
              "who" = who))

  }

greeting = GreetMe(what = "But who are", who = "the rest of you?")

greeting
```

```
## $what.who
## [1] "But who are the rest of you?"
##
## $what
## [1] "But who are"
##
## $who
## [1] "the rest of you?"
```

Now our greeting object is returned as a list. To access the components we have our old standby method $, or with bracket notation for list elements (note the double brackets for the list instead of single for matrix elements).

```
greeting$what.who
```

```
## [1] "But who are the rest of you?"
```

```
greeting[[1]]
```

```
## [1] "But who are the rest of you?"
```

6.0.2 Conditional Statements

A conditional statement (often called an if-else statement) is one that controls the flow of a function or block of code by using the result of a boolean test. Boolean is another term for a logical data type in R, or one that can take the values `TRUE` or `FALSE`. An if-else statement in R takes the general form:

`if (<condition TRUE>) {<do this>}`

or:

`if (<condition TRUE) {<do this>} else {<do this if FALSE>}`

or even:

`if (<condition TRUE) {<do this>} else if (<second condition TRUE>) { <do this>} else {<do this if all others FALSE>}`

You can piggyback as many if - else if - else pieces as you need. But if you find yourself doing a bunch of them, there is probably a better way.

Conditional statements rely on relational operators (also called relationals, relational tests, comparators). We have used several of them already in the code above.

The 6 relational operators in R are:

`x < y` (x is less than y)
`x > y` (x is greater than y)
`x <= y` (x is less than or equal to y)
`x >= y` (x is greater than or equal to y)
`x == y` (x is equal to y)
`x != y` (x is not equal to y)

Let's test them out.

```
5 < 4
```

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

```
"a" < "b"
```

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

```r
c(1,2,5) >= c(0,3,5)
```

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

```r
1 == 2
```

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

```r
0 == FALSE
```

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

```r
1 != 2
```

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

```r
c(1,2,5) != c(0,3,5)
```

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

In addition to relational operators, R has 3 primary logical operators. Logical operators provide tests for "and, or, and not." They are coded as:

`&` - element by element conditional AND. Both or all statements must be TRUE to return TRUE.

`|` - element by element conditional OR. One statement must be TRUE to return TRUE.

`!` - element by element conditional NOT. Returns the opposite logical test for an element.

There are also vector versions of `&&` and `||` that will use the first element of each vector and return a single value of TRUE or FALSE for the entire test. You probably won't need these and we won't talk about them here. We've also used the `%in%` operator previously. It is a special case operator that tests an element to see if it occurs in a vector of elements. Since this is just an extension of the OR operator, we don't consider it to be a basic logical operator.

Let's test the others.

```r
#all values greater than 0 will return a logical TRUE

5 & 4
```

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

```r
#here the 0 is a FALSE, so both conditions aren't met
5 & 0
```

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

```r
#but here we just need one of them to be TRUE to return a TRUE

5 | 0
```

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

```r
#opposite logic for NOT
!TRUE
```

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

```r
#element by element AND
c(1,2,5) & c(0,3,5)
```

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

```r
#element by element OR
c(FALSE,2,3.9) | c(0,3,5)
```

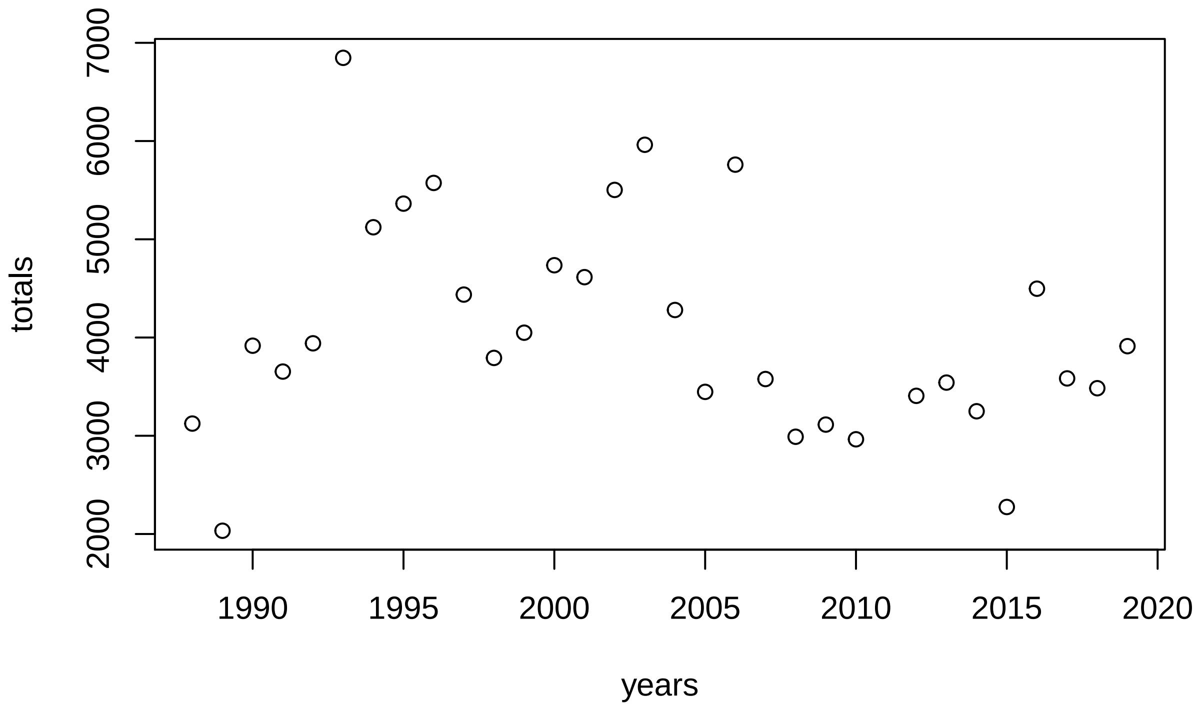
```
## [1] FALSE  TRUE  TRUE
```

```r
a = 5

(a > 3) & (a < 10)
```

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

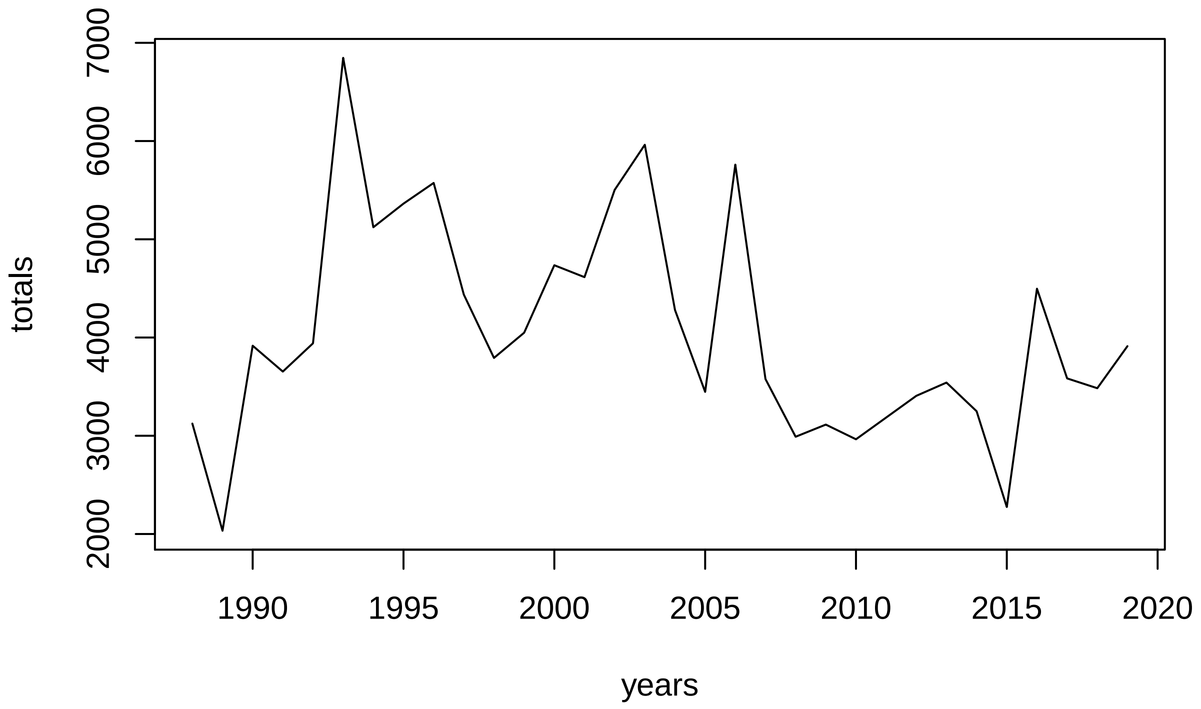
And now we can combine them into our if-else statements and functions.

```r
library(tidyverse)

PlotDucks = function (years, totals, type = "points"){

  if (type == "line") {

    plot(years,totals, type = "l")

  } else {plot(years, totals)}



}


ltdu = YKDHistoric$combined %>%
  filter(Species == "LTDU")

PlotDucks(years = ltdu$Year, totals = ltdu$total)
```



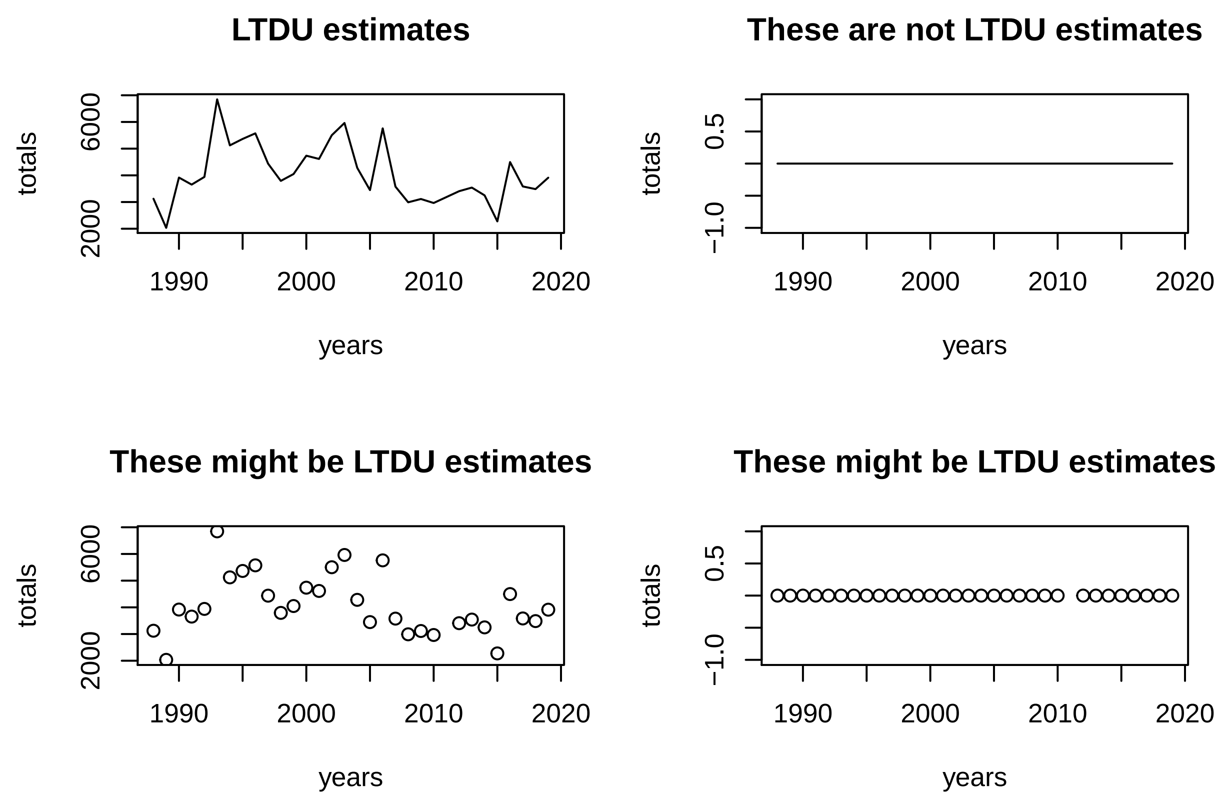
So here we used the argument “type” as a kind of setting. The user passes in the years, the totals, and then whether they want points or lines. Our function tests their input and responds accordingly.

```
PlotDucks(years = ltdu$Year, totals = ltdu$total, type = "line")
```



We can alter the function a little to show the use of the AND operator.

```r
PlotDucks = function (years, totals, species, type = "points"){

  if (type == "line" & species == "LTDU") {

    plot(years,totals, type = "l", main = "LTDU estimates")

  } else if (type == "line"){

    plot(years,totals, type = "l", main = "These are not LTDU estimates")

  } else {plot(years, totals, main = "These might be LTDU estimates")}

}


ltdu = YKDHistoric$combined %>%
  filter(Species == "LTDU")

par(mfrow=c(2,2))

PlotDucks(years = ltdu$Year, totals = ltdu$total, species = "LTDU", type = "line")

PlotDucks(years = ltdu$Year, totals = ltdu$total * 0, species = "not LTDU", type = "line")
PlotDucks(years = ltdu$Year, totals = ltdu$total, species = "LTDU")

PlotDucks(years = ltdu$Year, totals = ltdu$total * 0, species = "not LTDU")
```



```r
par(mfrow=c(1,1))
```

If we trace through the conditional formatting, R will run the first matching condition and exit out of the if-else. So by the end we get a catch-all that plots points and says they “might be” LTDU because we’ve only eliminated LTDU line plot, line plots for every other species. We don’t test the species and it is caught in our final else condition no matter what it is.

6.0.3 Looping

Loops are a special class of conditional statements that are designed to replicate a process either over a sequence of values (`for` loops) or indefinitely until a condition is met (`while` or `repeat` loops). The latter are used more in application programming than data programming, so we won’t talk about them much. For loops are far and away the most common loop in R programming, though they aren’t always necessary. The general structure of a `for` loop is:

```
for (<value> in <sequence>) {<do stuff>}
```

Here is a basic example. We will take values of i between 1 and 10 and print the squared value.

```r
for (i in 1:10) {

  print(i^2)

}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
## [1] 81
## [1] 100
```

In this case, i started at the first value of our sequence, was squared and output, reached the end, and iterated back until it reached the final value of the sequence. We can add some nonsense complexity to it do demonstrate how we can skip or break the loop.

```r
for (i in 1:10) {

  #we don't want to deal with 5 so we use next to skip it.
  #this jumps to the end of the loop and starts over with the next value.

  if(i == 5){next}

  print(i^2)

}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 36
## [1] 49
## [1] 64
## [1] 81
## [1] 100
```

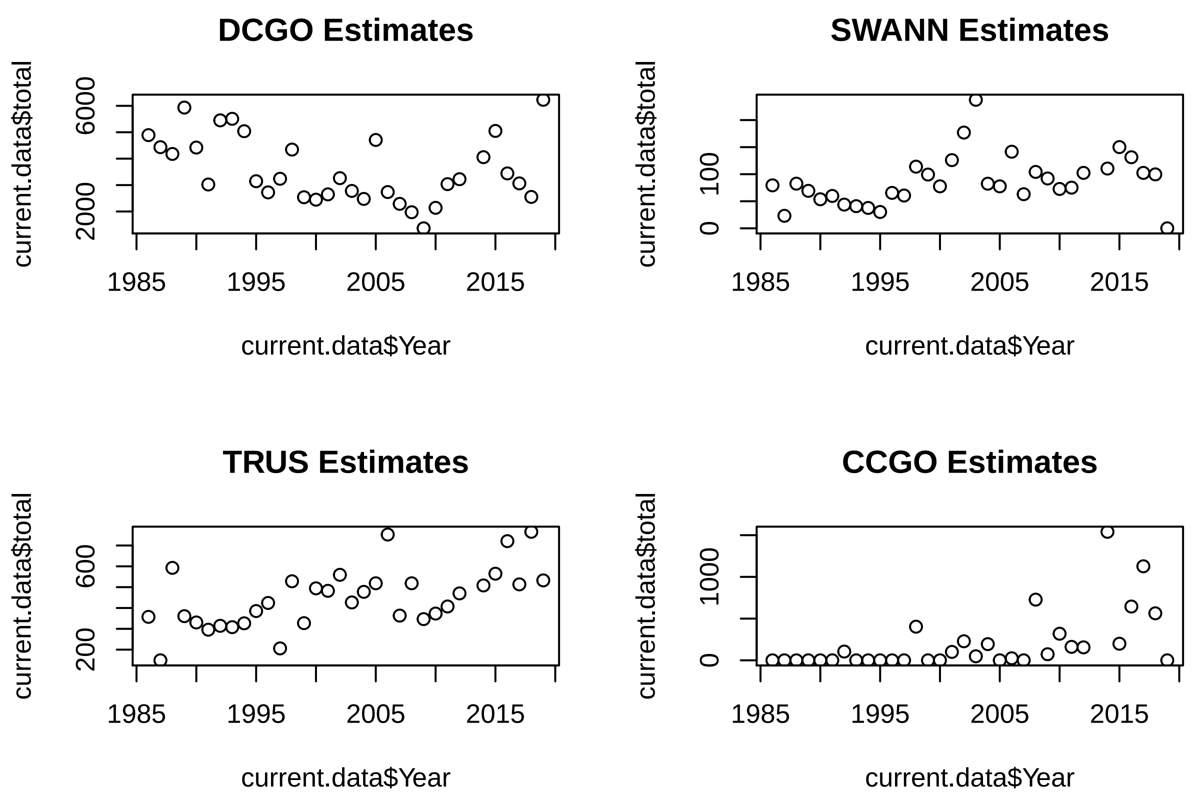
```r
for (i in 1:10) {

  #we don't want to continue squaring past 50.
  #this test will break (end) the loop entirely when triggered.

  if(i^2 > 50){break}

  print(i^2)

}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
```

One common use of next would be to catch and skip NA values with `is.na()`. There aren't many reasons why you would want to `break` your loops. These are more appropriate in application programming than in data analysis and manipulation.

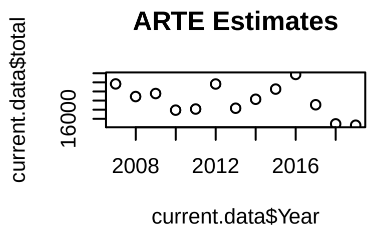
Let's do something a little more useful with our for loop inside a function. Imagine we want to take a table of estimates and plot them out by species no matter how many species we have. We can do it with a for loop inside a function.

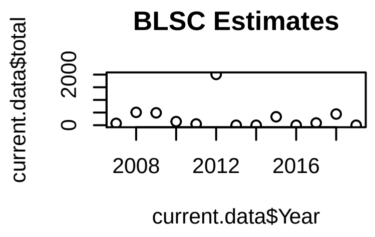
```r
#we assume here that we know the structure of our data and that the columns exist

PlotAll = function(data){

  #make a list of all known species
  species.list = as.character(unique(data$Species))

  #iterate through the length of our species list
  for (i in 1:length(species.list)){

    current.species = species.list[i]

    #filter our data to our current species
    current.data = data %>% filter(Species == current.species)

    plot(current.data$Year, current.data$total,
         main = paste(current.species, "Estimates"))


  }

}


#we cheat a little here since I know there are 4 species to plot
par(mfrow=c(2,2))

PlotAll(CRDHistoric$combined)
```

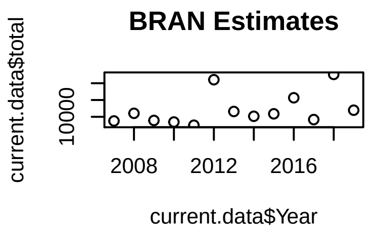


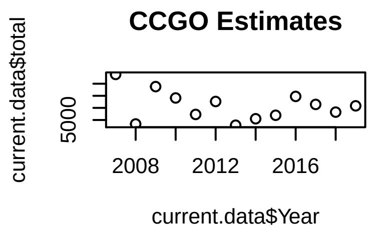
If we wrote our function correct, we should be able to apply it to other, larger data tables.

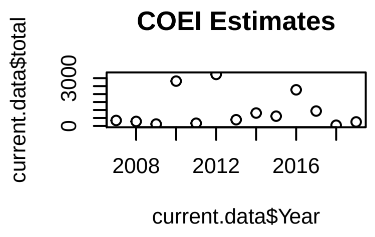
```r
par(mfrow=c(3,3))

PlotAll(ACPHistoric$combined)
```

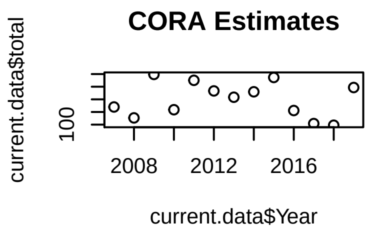

ARTE Estimates
current.data$total
16000
2008
2012
2016
current.data$Year

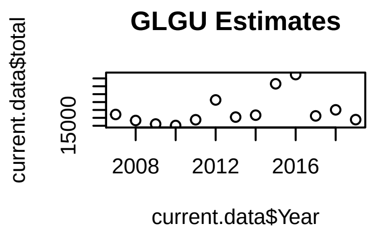

BLSC Estimates
current.data$total
0
2000
2008
2012
2016
current.data$Year

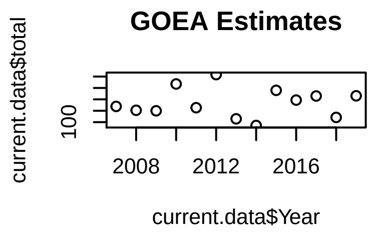

BRAN Estimates
current.data$total
10000
2008
2012
2016
current.data$Year

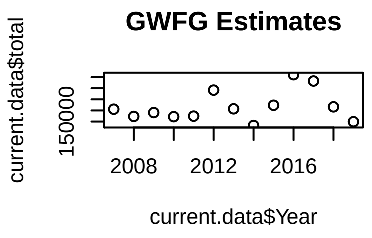

CCGO Estimates
current.data$total
5000
2008
2012
2016
current.data$Year

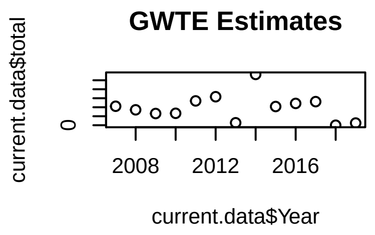

COEI Estimates
current.data$total
0
3000
2008
2012
2016
current.data$Year

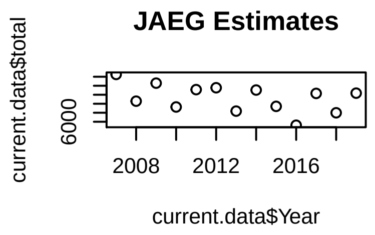

CORA Estimates
current.data$total
100
2008
2012
2016
current.data$Year

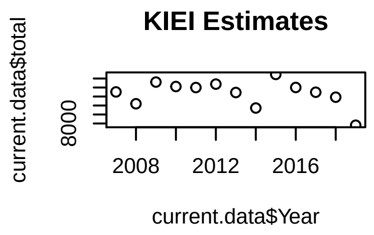

GLGU Estimates
current.data$total
15000
2008
2012
2016
current.data$Year

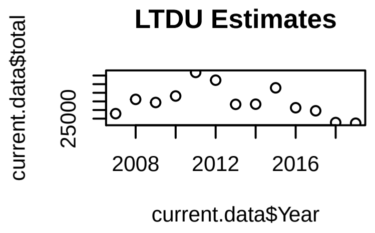

GOEA Estimates
current.data$total
100
2008
2012
2016
current.data$Year

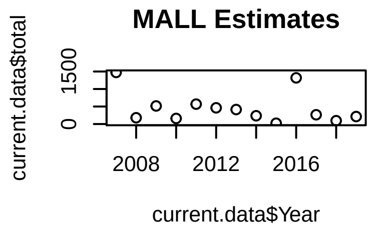

GWFG Estimates
current.data$total
150000
2008
2012
2016
current.data$Year

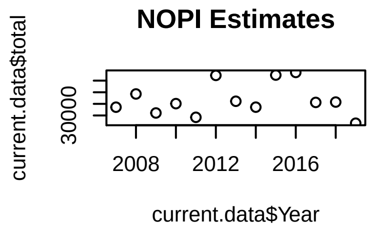

GWTE Estimates
current.data$total
0
2008
2012
2016
current.data$Year

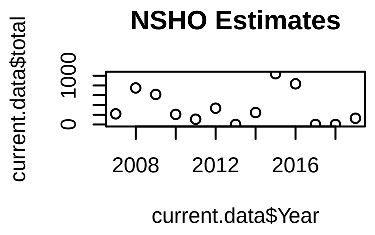

JAEG Estimates
current.data$total
6000
2008
2012
2016
current.data$Year

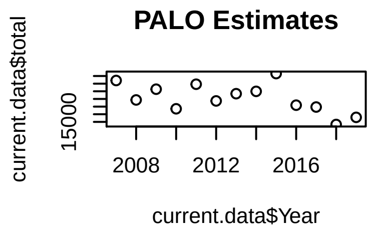

KIEI Estimates
current.data$total
8000
2008
2012
2016
current.data$Year

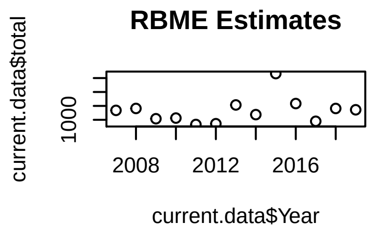

LTDU Estimates
current.data$total
25000
2008
2012
2016
current.data$Year

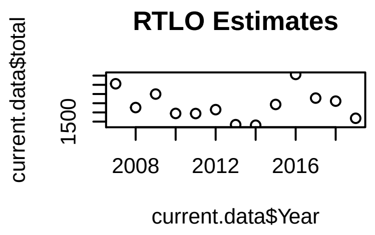

MALL Estimates
current.data$total
0
1500
2008
2012
2016
current.data$Year

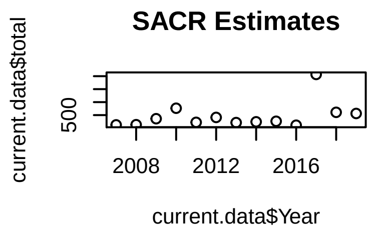

NOPI Estimates
current.data$total
30000
2008
2012
2016
current.data$Year

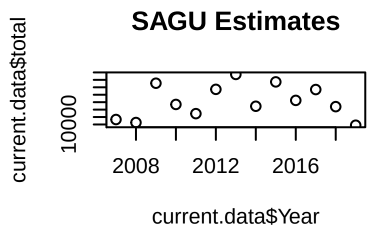

NSHO Estimates
current.data$total
0
1000
2008
2012
2016
current.data$Year

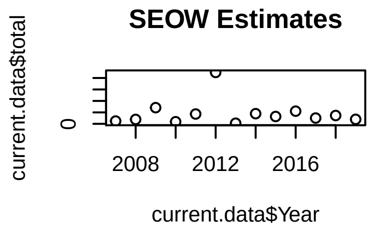

PALO Estimates
current.data$total
15000
2008
2012
2016
current.data$Year

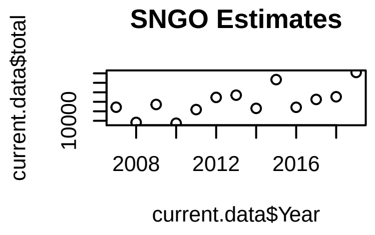

RBME Estimates
current.data$total
1000
2008
2012
2016
current.data$Year

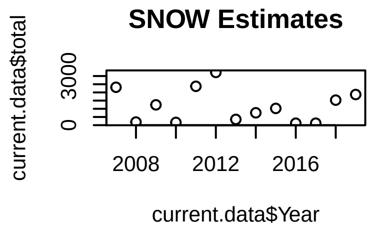

RTLO Estimates
current.data$total
1500
2008
2012
2016
current.data$Year

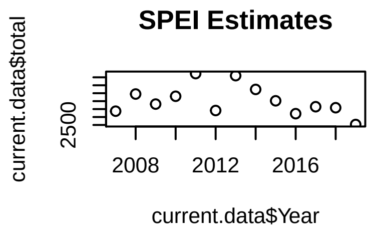

SACR Estimates
current.data$total
500
2008
2012
2016
current.data$Year

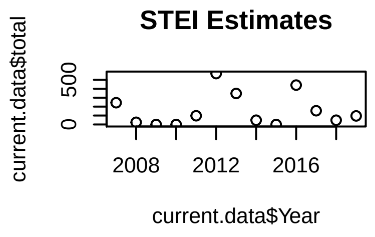

SAGU Estimates
current.data$total
10000
2008
2012
2016
current.data$Year

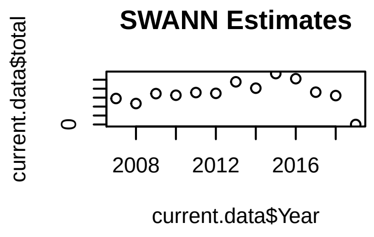

SEOW Estimates
current.data$total
0
2008
2012
2016
current.data$Year

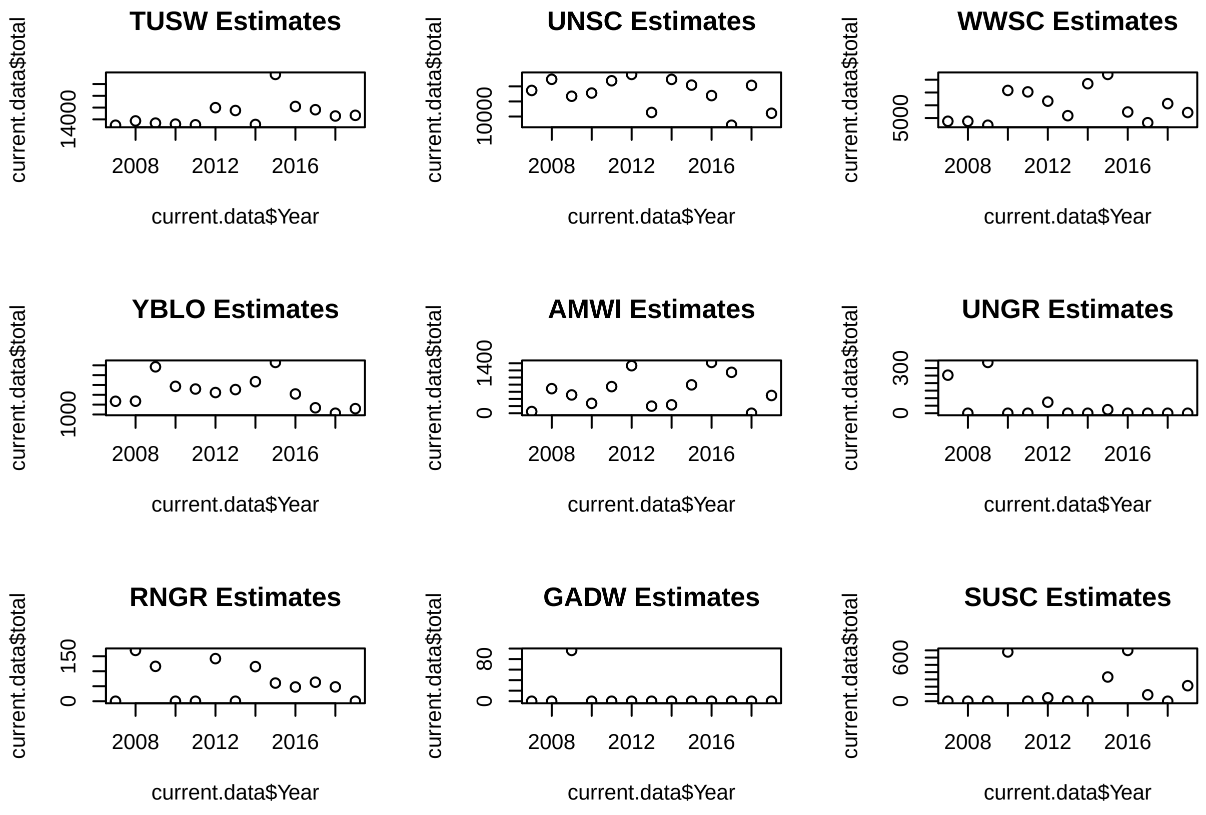

SNGO Estimates
current.data$total
10000
2008
2012
2016
current.data$Year


SNOW Estimates
current.data$total
0
3000
2008
2012
2016
current.data$Year

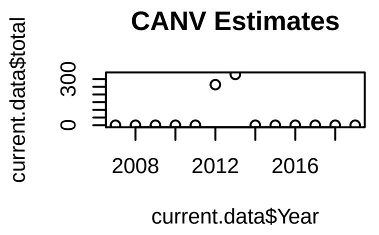

SPEI Estimates
current.data$total
2500
2008
2012
2016
current.data$Year

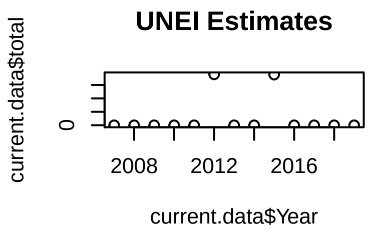

STEI Estimates
current.data$total
0
500
2008
2012
2016
current.data$Year

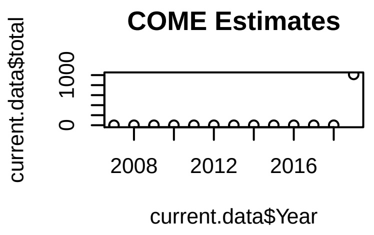

SWANN Estimates
current.data$total
0
2008
2012
2016
current.data$Year



```r
par(mfrow=c(1,1))
```







You can nest a mind-boggling number of for loops together as well. Although if you find yourself doing this, there is probably a function or other method that would be cleaner and easier. Our example will create an empty list of project areas and the last 3 years to store some future values in.

```
area=character()

year=integer()

area.list = c("ACP", "CRD", "YKD")

for (i in 1:length(area.list)) {

  for (j in 2018:2020) {

    area = rbind(area, area.list[i])
    year = rbind(year, j)

  }

}

area.year = data.frame(area = area, year = year, row.names = c(1:length(area)))


area.year
```

```
##   area year
## 1  ACP 2018
## 2  ACP 2019
## 3  ACP 2020
## 4  CRD 2018
## 5  CRD 2019
## 6  CRD 2020
## 7  YKD 2018
## 8  YKD 2019
## 9  YKD 2020
```

And finally, add a species in there.

```
area=character()

year=integer()

species=character()

area.list = c("ACP", "CRD", "YKD")

species.list = c("LTDU", "STEI")

for (i in 1:length(area.list)) {

  for (j in 1:length(species.list)){

      for (k in 2018:2020) {

        area = rbind(area, area.list[i])
        species = rbind(species, species.list[j])
        year = rbind(year, k)

    }
  }
}

area.species.year = data.frame(area = area,
                               species = species,
                               year = year,
                               row.names = c(1:length(area)))


area.species.year
```

```
##    area species year
## 1   ACP    LTDU 2018
## 2   ACP    LTDU 2019
## 3   ACP    LTDU 2020
## 4   ACP    STEI 2018
## 5   ACP    STEI 2019
## 6   ACP    STEI 2020
## 7   CRD    LTDU 2018
## 8   CRD    LTDU 2019
## 9   CRD    LTDU 2020
## 10  CRD    STEI 2018
## 11  CRD    STEI 2019
## 12  CRD    STEI 2020
## 13  YKD    LTDU 2018
## 14  YKD    LTDU 2019
## 15  YKD    LTDU 2020
## 16  YKD    STEI 2018
## 17  YKD    STEI 2019
## 18  YKD    STEI 2020
```

But, as I mentioned with nested for loops, there is probably a quicker way. And with automatic vectorization in R, even the simpler for loops are recreated easily.

```r
i = c(1:10)
j = i^2

j
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

```r
expand.grid(area.list, species.list, c(2018:2020))
```

```
##    Var1 Var2 Var3
## 1   ACP LTDU 2018
## 2   CRD LTDU 2018
## 3   YKD LTDU 2018
## 4   ACP STEI 2018
## 5   CRD STEI 2018
## 6   YKD STEI 2018
## 7   ACP LTDU 2019
## 8   CRD LTDU 2019
## 9   YKD LTDU 2019
## 10  ACP STEI 2019
## 11  CRD STEI 2019
## 12  YKD STEI 2019
## 13  ACP LTDU 2020
## 14  CRD LTDU 2020
## 15  YKD LTDU 2020
## 16  ACP STEI 2020
## 17  CRD STEI 2020
## 18  YKD STEI 2020
```