

Hydroinformatics at VT

JP Gannon

2023-02-06

Contents

1	Introduction	11
1.1	How to use these materials	12
1.2	Table of contents:	12
2	Intro to Plotting	15
2.1	Download and install tidyverse library	15
2.2	Reading data	16
2.3	Our first ggplot	17
2.4	Change point type	18
2.5	Set colors	19
2.6	Controlling color with a third variable and other functions	21
2.7	Plotting multiple groups	21
2.8	Facets	22
2.9	Two variable facetting	23
2.10	Boxplots	24
2.11	More about color, size, etc	25
2.12	Multiple geoms	26
3	R Tidyverse Programming Basics	29
3.1	Introduction	29
3.2	You can use R as a calculator	29
3.3	You can create new objects using <-	30
3.4	Using functions	31

3.5	Read in some data.	32
3.6	Wait, hold up. What is a tibble?	33
3.7	Data wrangling in dplyr	34
3.8	Filter	35
3.9	Arrange	36
3.10	Select	38
3.11	Mutate	39
3.12	Summarize	40
3.13	Multiple operations with pipes	41
3.14	Save your results to a new tibble	43
3.15	What about NAs?	43
3.16	What are some things you think I'll ask you to do for the activity next class?	44
4	ACTIVITY Intro Skills	45
4.1	Problem 1	45
4.2	Problem 2	45
4.3	Problem 3	45
4.4	Problem 4	46
4.5	Problem 5	46
4.6	Problem 6	46
5	Introduction to Basic Statistics	47
5.1	Reading for this section: Statistical Methods in Water Resources: Chapter 1	47
5.2	Questions for today:	47
5.3	What is the difference between a sample and a population.	50
5.4	Measuring our sample distribution: central tendency.	52
5.5	Measures of variability	55
5.6	What is a normal distribution and how can we determine if we have one?	59

CONTENTS	5
6 ACTIVITY Intro Stats	61
6.1 Problem 1	61
6.2 Problem 2	61
6.3 Problem 3	61
6.4 Problem 4	62
6.5 Problem 5	62
6.6 Problem 6	62
7 Joins, Pivots, and USGS dataRetrieval	63
7.1 Goals for today	63
7.2 Exploring what dataRetrieval can do.	64
7.3 Joins	65
7.4 Join example	69
7.5 Finding IDs to download USGS data	71
7.6 OK let's download some data!	72
7.7 Pivoting: wide and long data	75
7.8 Pivot Examples	75
8 ACTIVITY Joins Pivots dataRetrieval	83
8.1 Load the tidyverse, dataRetrieval, and patchwork packages.	83
8.2 Problem 1	83
8.3 Problem 2	83
8.4 Problem 3	84
8.5 Problem 4	84
8.6 Problem 5	84
8.7 Problem 6	84
8.8 Problem 7	84
9 ACTIVITY Summative 1	85
9.1 Problem 1	85
9.2 Problem 2	85
9.3 Problem 3	86

9.4 Problem 4	86
9.5 Problem 5	86
9.6 Problem 6	86
9.7 Problem 7	87
9.8 Problem 8	87
9.9 Problem 9	87
9.10 Problem 10	88
10 Flow Duration Curves	89
10.1 Get data	90
10.2 Review: describe the distribution	90
10.3 ECDFs	91
10.4 Calculate flow exceedence probabilities	92
10.5 Plot a Flow Duration Curve using the probabilities	93
10.6 Make an almost FDC with stat_ecdf	94
10.7 Example use of an FDC	95
10.8 Compare to a boxplot of the same data	96
10.9 Challenge: Examining flow regime change at the Grand Canyon .	97
11 Low Flow Analysis	101
11.1 What are low flow statistics?	101
11.2 Get data	102
11.3 Create the X days average flow record	103
11.4 Look at what a rolling mean does.	104
11.5 Calculate yearly minimums	105
11.6 Calculate return interval	106
11.7 Fit to Pearson Type III distribution	108
11.8 Distribution-free method	112
12 Flood Frequency Analysis and Creating Functions	113
12.1 Template Repository	113
12.2 Intro	113
12.3 Challenge: Create a function	122

13 Geospatial data in R - Vector	123
13.1 Goals	123
13.2 Intro to tmap	124
13.3 Data wrangling with tidyverse principles	133
13.4 Plot maps side by side	139
13.5 Built in styles, like themes in ggplot	140
13.6 Interactive Maps	141
14 Summative Assessment 2	143
14.1 Info for assessment	143
15 Geospatial R Raster - Hydro Analyses	145
15.1 Introduction	145
15.2 Read in DEM	146
15.3 Plot DEM	146
15.4 Generate a hillshade	147
15.5 Prepare DEM for Hydrology Analyses	148
15.6 Visualize filled sinks and breached depressions	150
15.7 D8 Flow Accumulation	151
15.8 D infinity flow accumulation	153
15.9 Topographic Wetness Index	155
15.10 Downslope TWI	156
15.11 Map Stream Network	156
15.12 Extract raster values to point locations	158
15.13 View raster data as a PDF or histogram	160
15.14 Subsetting a raster for visualization	161
15.15 Raster Math	162
15.16 Extra: plot topo characteristics against one another	163

16 Geospatial R Raster - Watershed Delineation	165
16.1 Introduction	165
16.2 The watershed delineation tool/process	166
16.3 Read in DEM	167
16.4 Generate a hillshade	168
16.5 Prepare DEM for Hydrology Analyses	168
16.6 Create flow accumulation and pointer grids	169
16.7 Setting pour points	169
16.8 Delineate watersheds	171
16.9 Convert watersheds to shapefiles	172
16.10 Extract data based on watershed outline	172
16.11 BONUS: Make a 3d map of your watershed with rayshader . . .	173
17 Intro to Modeling - Getting Started with HBV	177
17.1 Introduction	177
17.2 Creating the HBV model function	178
17.3 Read in Precip and Temp	179
17.4 Calculate PET	180
17.5 HBV Parameters	181
17.6 First model run	182
17.7 Import observed streamflow data	183
17.8 Compare observed and modeled discharge graphically	184
17.9 Compare observed and modelled discharge with interactive graph	185
17.10 Measure how well the model fits with NSE	185
17.11 Assess model fit with a different measure: Snow	186
17.12 Calibrate HBV manually	188
18 Intro to Modeling - Calibrate HBV	191
18.1 Introduction	191
18.2 Challenge: Write a for loop	194
18.3 Prep data for HBV	194
18.4 Calculate PET	196

CONTENTS	9
----------	---

18.5 Monte Carlo step 1: generate random parameter sets	196
18.6 Run the model for each parameter set	198
18.7 Find the best parameter set	200
18.8 Investigating a much bigger Monte Carlo	201

Chapter 1

Introduction

- 1.0.1 To help me keep get an idea of who is using this resource so I can improve it in the future, please consider filling out any or all of this survey: <https://forms.gle/6Zcntzvr1wZZUh6S7> Thanks!**

This bookdown contains the notes and most exercises for a course on data analysis techniques in hydrology using the programming language R. The material will be updated each time the course is taught. If new topics are added, the topics they replace will be left, in case they are useful to others.

I hope these materials can be a resource to those teaching themselves R for hydrologic analysis and/or for instructors who may want to use a lesson or two or the entire course. Each chapter in this bookdown is linked to a github repository where the code can be downloaded or copied to another github account.

If you have questions, suggestions, or would like activity answer keys, etc. please email me at jpgannon at vt.edu

The following resources, among others, were very helpful when compiling the chapters of this book. They are also linked in specific chapters, along with other resources. These are great resources if you want to dig deeper into topics covered in this bookdown.

R for Data Science: <https://r4ds.had.co.nz/>

Statistical Methods in Water Resources: <https://pubs.er.usgs.gov/publication/tm4A3>

Geocomputation with R: <https://geocompr.robinlovelace.net/>

1.1 How to use these materials

At the top of each chapter there is a link to a github repository. In each repository is the code that produces each chapter and a version where the code chunks within it are blank. These repositories are all template repositories, so you can easily copy them to your own github space by clicking *Use This Template* on the repo page.

In my class, I work through the each document, live coding with students following along. Typically I ask students to watch as I code and explain the chunk and then replicate it on their computer. Depending on the lesson, I will ask students to try some of the chunks before I show them the code as an in-class activity. Some chunks are explicitly designed for this purpose and are typically labeled a “challenge”.

Chapters called ACTIVITY are either homework or class-period-long in-class activities. The code chunks in these are therefore blank. If you would like a key for any of these, please just send me an email.

1.2 Table of contents:

- 2 **Intro to Plotting:** *Introduction to plotting with ggplot.*
- 3 **R Tidyverse Programming Basics:** *Introduction to basic R syntax and dplyr verbs.*
- 4 **ACTIVITY Intro Skills:** *Activity to practice basic plotting and programming.*
- 5 **Introduction to Basic Statistics:** *Introcutiong to basic ways to measure a data distribution.*
- 6 **ACTIVITY Intro Stats:** *Activity to practice basic statistics concepts.*
- 7 **Joins, Pivots, and USGS dataRetrieval:** *Joins and Pivots, using USGS dataRetrieval to generate examples.*
- 8 **ACTIVITY Joins Pivots dataRetrieval:** *Activity to practice Joins, Pivots, and dataRetrieval.*
- 9 **ACTIVITY Summative 1:** *First summative assessment/practice.*
- 10 **Flow Duration Curves:** *Building and exploring flow duration curves.*
- 11 **Low Flow Analysis:** *How to calculate low-flow statistics (ex: 7Q10, 1Q10).*
- 12 **Flood Frequency Analysis and Creating Functions:** *Flood frequency analysis and making your own functions.*
- 13 **Geospatial data in R - Vector:** *Intro to working with vector data in R*
- 14 **Summative Assessment 2:** *Summative Assessment #2*
- 15 **Geospatial R Raster - Hydro Analyses:** *Intro to working with raster data in R, hydrological raster analyses*
- 16 **Geospatial R Raster - Watershed Delineation:** *Watershed delineation and extracting DEM values by watershed*
- 17 **Intro to Modeling - Getting Started with HBV:** *Setting up a simple*

hydrologic model in R

18 **Intro to Modeling - Calibrate HBV:** *Using the Monte Carlo method to calibrate the HBV model, discuss equifinality*

Chapter 2

Intro to Plotting

Get this document and a version with empty code chunks at the template repository on github: <https://github.com/VT-Hydroinformatics/1-Intro-plotting-R>

2.1 Download and install tidyverse library

We will use the tidyverse a lot this semester. It is a suite of packages that handles plotting and data wrangling efficiently.

You only have to install the library once. You have to load it using the `library()` function each time you start an R session.

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

## Warning: package 'tidyverse' was built under R version 4.1.2

## -- Attaching packages ----- tidyverse 1.3.2 --
## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble   3.1.8     v dplyr    1.0.9
## v tidyr    1.2.0     v stringr  1.4.0
## v readr    2.1.2     vforcats  0.5.1

## Warning: package 'ggplot2' was built under R version 4.1.2

## Warning: package 'tibble' was built under R version 4.1.2

## Warning: package 'tidyr' was built under R version 4.1.2
```

```
## Warning: package 'readr' was built under R version 4.1.2

## Warning: package 'dplyr' was built under R version 4.1.2

## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

2.2 Reading data

The following lines will read in the data we will use for this exercise. Don't worry about this right now beyond running it, we will talk more about it later.

```
Pine <- read_csv("PINE_Jan-Mar_2010.csv")

## Rows: 2160 Columns: 8
## -- Column specification -----
## Delimiter: ","
## chr (2): StationID, surrogate
## dbl (5): cfs, year, quarter, month, day
## dttm (1): datetime
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

SNP <- read_csv("PINE_NFDR_Jan-Mar_2010.csv")

## Rows: 4320 Columns: 8
## -- Column specification -----
## Delimiter: ","
## chr (2): StationID, surrogate
## dbl (5): cfs, year, quarter, month, day
## dttm (1): datetime
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

RBI <- read_csv("Flashy_Dat_Subset.csv")

## Rows: 49 Columns: 26
## -- Column specification -----
```

```
## Delimiter: ","
## chr (4): STANAME, STATE, CLASS, AGGECOREGION
## dbl (22): site_no, RBI, RBIRank, DRAIN_SQKM, HUC02, LAT_GAGE, LNG_GAGE, PPTA...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Create the plotting space: `ggplot()`

Represent the data on the plotting space: `geom_point()`

- + Lets you know there is more coming

`ggplot(data = cars, aes(x = speed, y = dist)) +
 geom_point()`

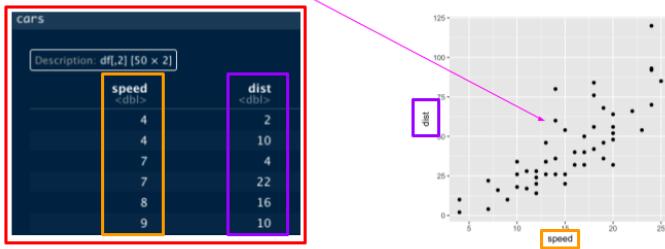
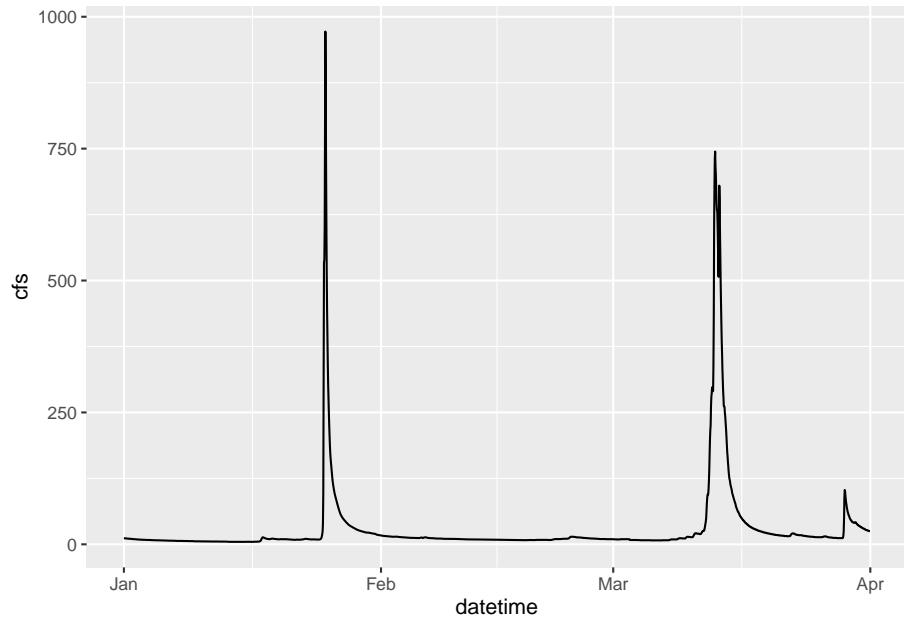


Figure 2.1: Basic ggplot syntax

2.3 Our first ggplot

Let's look at the Pine data, plotting streamflow (the `cfs` column) by the date (datetime column). We will show the time series as a line.

```
ggplot(data = Pine, aes(x = datetime, y = cfs))+  
  geom_line()
```



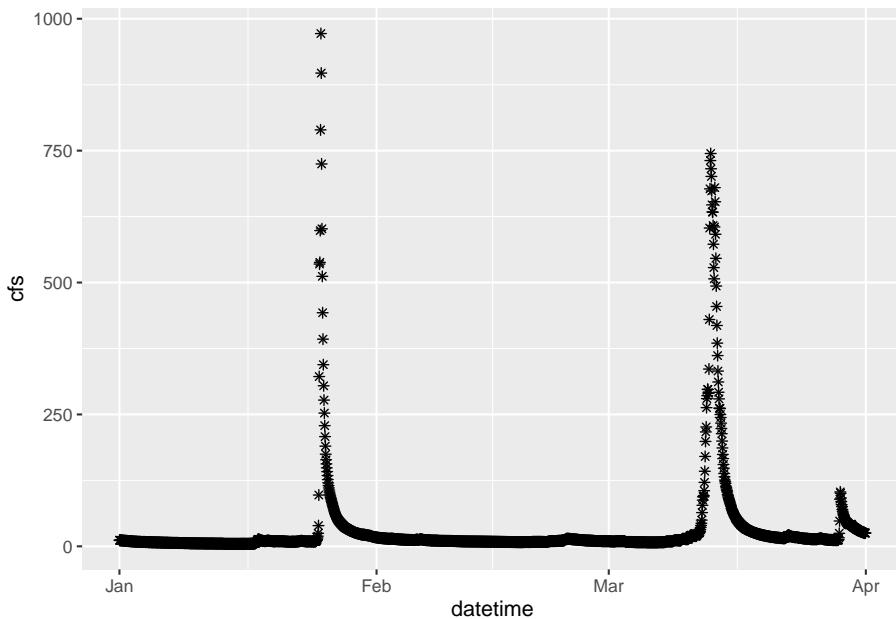
2.4 Change point type

Now let's make the same plot but show the data as points, using the pch parameter in geom_point() we can change the point type to any of the following:



Figure 2.2: pch options from R help file

```
ggplot(data = Pine, aes(x = datetime, y = cfs))+
  geom_point(pch = 8)
```



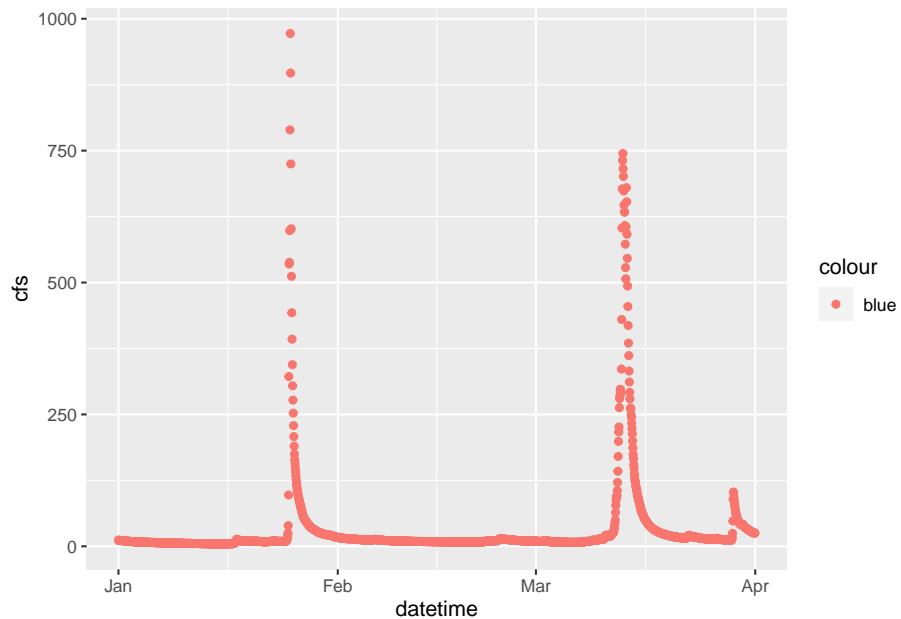
2.5 Set colors

We can also “easily” change the color. Easily is in quotes because this often trips people up. If you put color = “blue” in the aesthetic function, think about what that is telling ggplot. It says “control the color using”blue”“. That doesn’t make a whole lot of sense, so neither does the output... Try it.

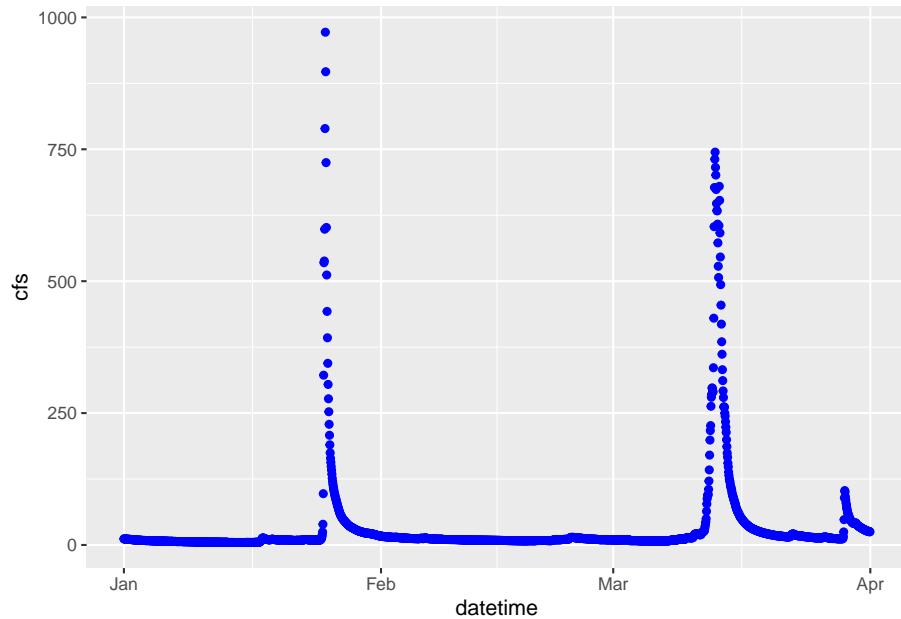
What happens is that if color = “blue” is in the aesthetic, you are telling R that the color used in the geom represents “blue”. This is very useful if you have multiple geoms in your plot, are coloring them differently, and are building a legend. But if you are just trying to color the points, it kind of feels like R is trolling you... doesn’t it?

Take the color = “blue” out of the aesthetic and you’re golden.

```
ggplot(data = Pine, aes(datetime, y = cfs, color = "blue"))+
  geom_point()
```



```
ggplot(data = Pine, aes(x = datetime, y = cfs))+  
  geom_point(color = "blue")
```



2.6 Controlling color with a third variable and other functions

Let's plot the data as a line again, but play with it a bit.

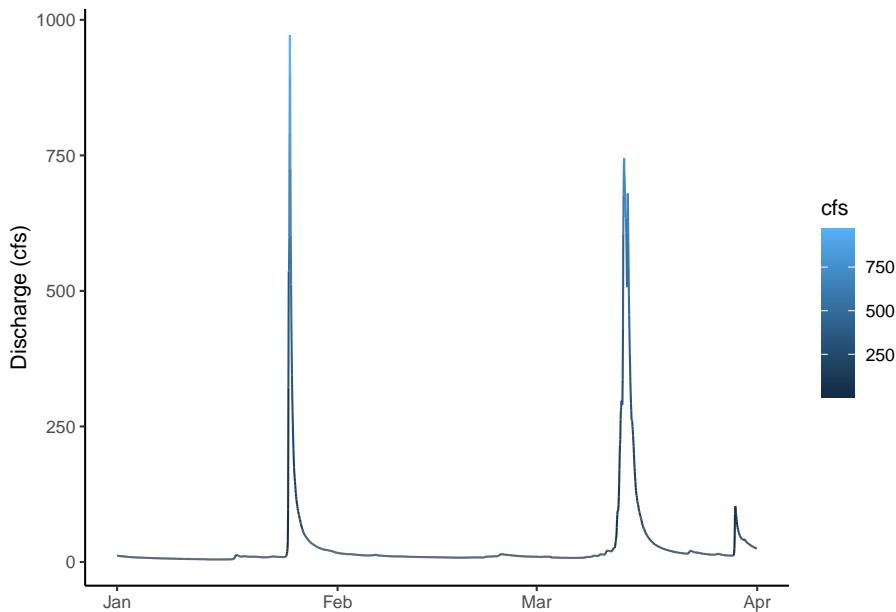
First: make the line blue

Second: change the theme

Third: change the axis labels

Fourth: color by discharge

```
ggplot(data = Pine, aes(x = datetime, y = cfs, color = cfs))+
  geom_line()+
  ylab("Discharge (cfs)")+
  xlab(element_blank())+
  theme_classic()
```



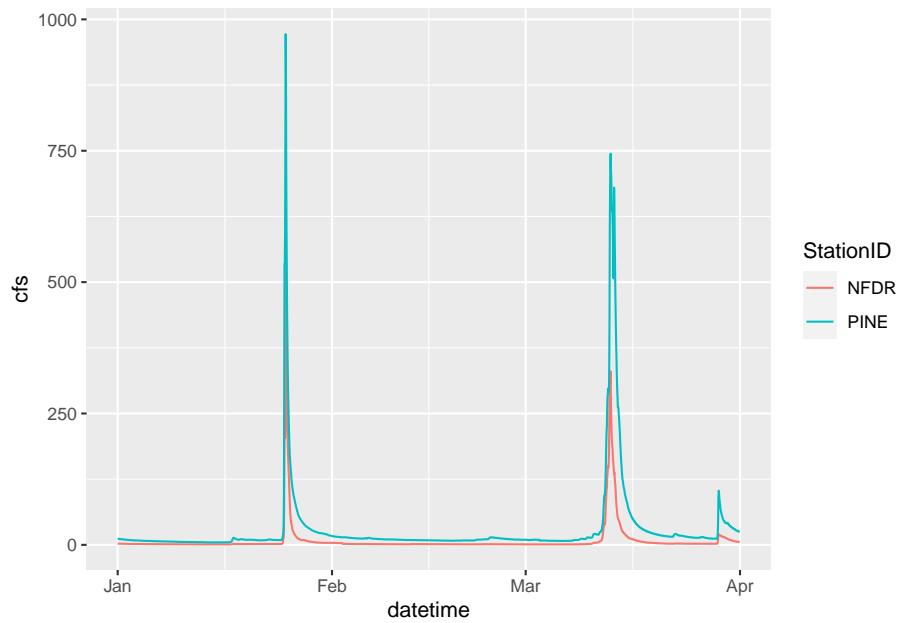
2.7 Plotting multiple groups

The SNP dataset has two different streams: Pine and NFDR

We can look at the two of those a couple of different ways

First, make two lines, colored by the stream by adding color = to your aesthetic.

```
ggplot(data = SNP, aes(x = datetime, y = cfs, color = StationID)) +
  geom_line()
```

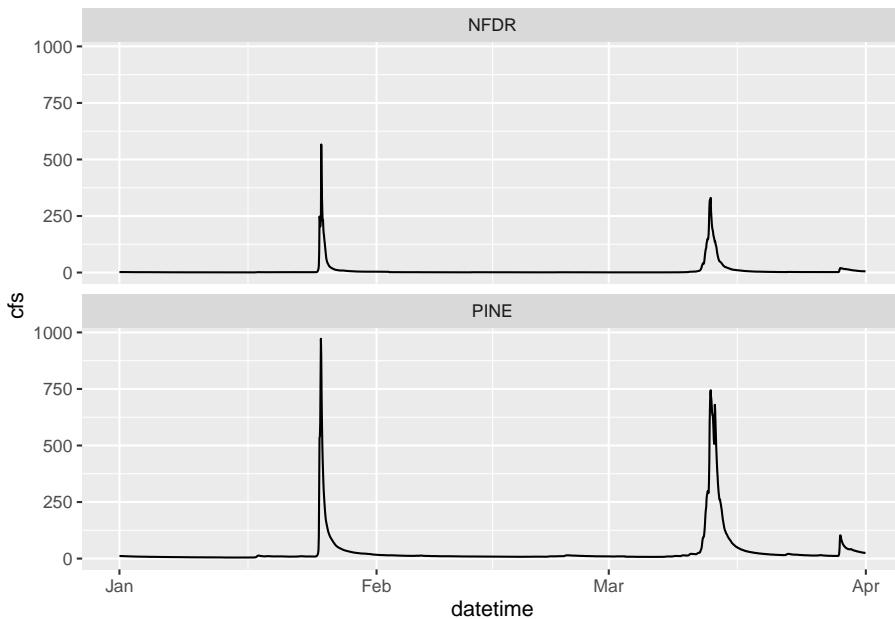


2.8 Facets

We can also use facets.

You must tell the `facet_wrap` what variable to use to make the separate panels (`facet =`). It'll decide how to orient them or you can tell it how. We want them to be on top of each other so we are going to tell it we want 2 rows by setting `nrow = 2`. Note that we have to put the column used to make the facets in quotes after `facets =`

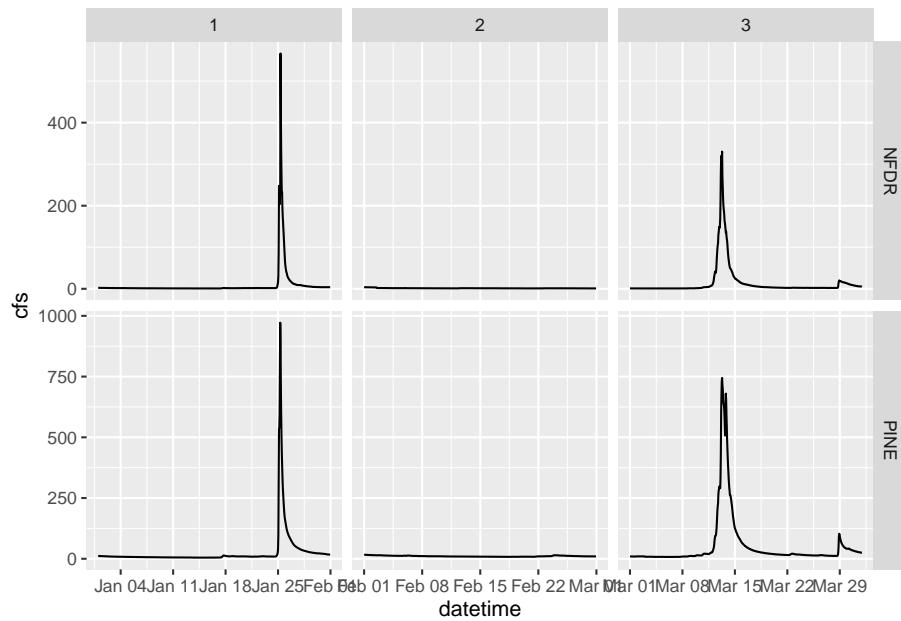
```
ggplot(data = SNP, aes(x = datetime, y = cfs)) +
  geom_line() +
  facet_wrap(facets = "StationID", nrow = 2)
```



2.9 Two variable faceting

You can also use `facet_grid()` to break your plots up into panels based on two variables. Below we will create a panel for each month in each watershed. Adding `scales = "free"` allows `facet_grid` to change the axes. By default, all axes will be the same. This is often what we want, so we can more easily compare magnitudes, but sometimes we are looking for patterns more, so we may want to let the axes have whatever range works for the individual plots.

```
ggplot(data = SNP, aes(x = datetime, y = cfs)) +
  geom_line() +
  facet_grid(StationID ~ month, scales = "free")
```

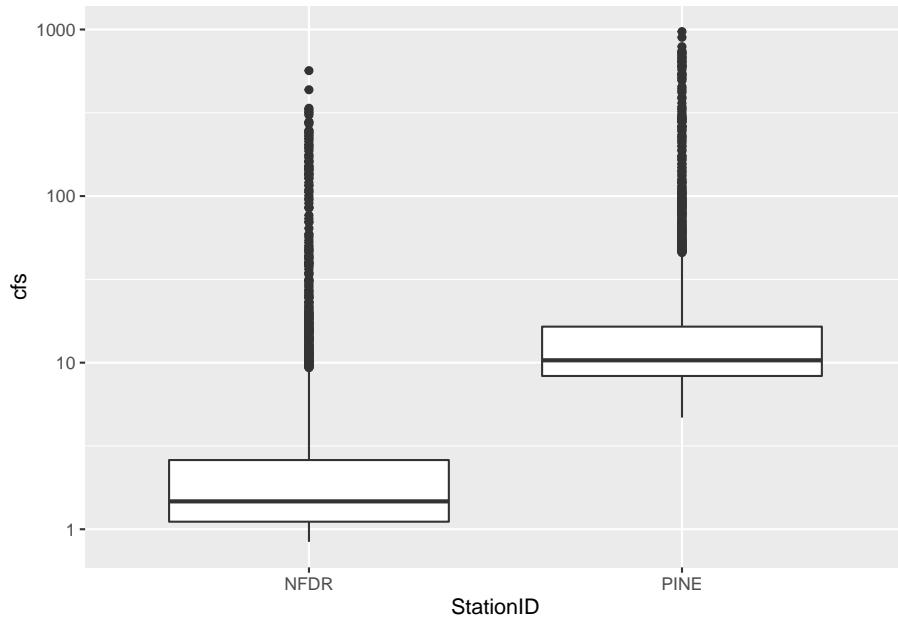


2.10 Boxplots

We can look at these data in other ways as well. A very useful way to look at the variation of two groups is to use a boxplot.

Because the data span several orders of magnitude, we will have to log the y axis to see the differences between the two streams. We do that by adding `scale_y_log10()`

```
ggplot(data = SNP, aes(x = StationID, y = cfs)) +
  stat_boxplot() +
  scale_y_log10()
```



2.11 More about color, size, etc

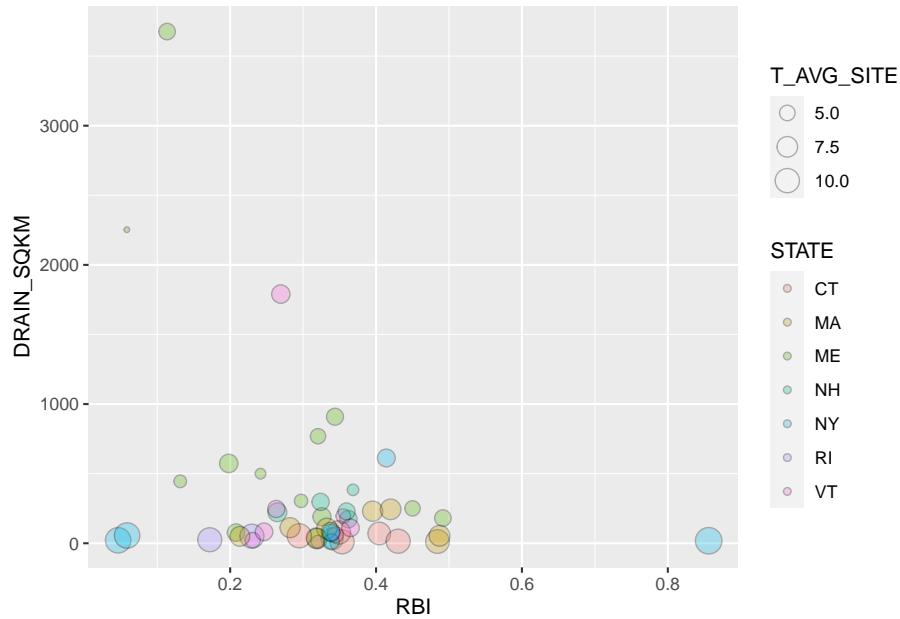
Let's play around a bit with controlling color, point size, etc with other data.

We can control the size of points by putting size = in the aes() and color by putting color =

If you use a point type that has a background, like #21, you can also set the background color using bg =

If points are too close together to see them all you can use a hollow point type or set the alpha lower so the points are transparent (alpha =)

```
ggplot(RBI, aes(RBI, DRAIN_SQKM, size = T_AVG_SITE, bg = STATE))+  
  geom_point(pch = 21, alpha = 0.3)
```

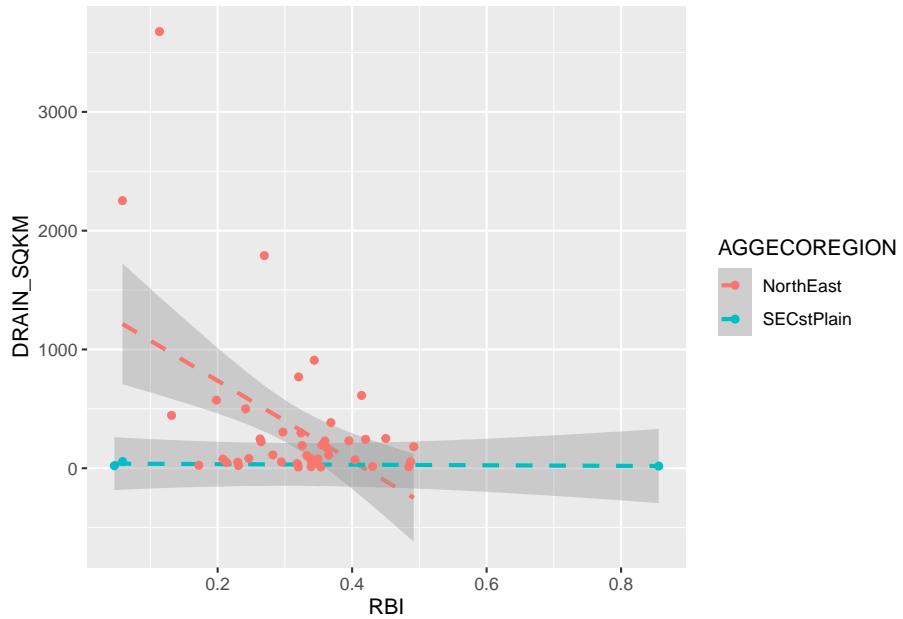


2.12 Multiple geoms

Finally: You can add multiple geoms to the same plot. Examples of when you might want to do this are when you are showing a line fit and want to show the points as well, or maybe showing a boxplot and want to show the data behind it. You simply add additional geom_... lines to add additional geoms.

```
ggplot(RBI, aes(RBI, DRAIN_SQKM, color = AGGECOREGION))+
  stat_smooth(method = "lm", linetype = 2)+
  geom_point()
```

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



Chapter 3

R Tidyverse Programming Basics

Get this document and a version with empty code chunks at the template repository on github: <https://github.com/VT-Hydroinformatics/2-Programming-Basics>

3.1 Introduction

We have messed around with plotting a bit and you've seen a little of what R can do. So now let's review or introduce you to some basics. Even if you have worked in R before, it is good to be remind of/practice with this stuff, so stay tuned in!

This exercise covers most of the same principles as two chapters in R for Data Science

Workflow: basics (<https://r4ds.had.co.nz/workflow-basics.html>)

Data transformation (<https://r4ds.had.co.nz/transform.html>)

3.2 You can use R as a calculator

If you just type numbers and operators in, R will spit out the results

```
1 + 2
```

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

3.3 You can create new objects using <-

Yea yea, = does the same thing. But use <-. We will call <- assignment or assignment operator. When we are coding in R we use <- to assign values to objects and = to set values for parameters in functions. Using <- helps us differentiate between the two. Norms for formatting are important because they help us understand what code is doing, especially when stuff gets complex.

Oh, one more thing: Surround operators with spaces. Don't code like a gorilla.

x <- 1 looks better than x<-1 and if you disagree you are wrong. :)

You can assign single numbers or entire chunks of data using <-

So if you had an object called my_data and wanted to copy it into my_new_data you could do:

```
my_new_data <- my_data
```

You can then recall/print the values in an object by just typing the name by itself.

In the code chunk below, assign a 3 to the object "y" and then print it out.

```
y <- 3
y
```

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

If you want to assign multiple values, you have to put them in the function c(). c means combine. R doesn't know what to do if you just give it a bunch of values with space or commas, but if you put them as arguments in the combine function, it'll make them into a vector.

Any time you need to use several values, even passing as an argument to a function, you have to put them in c() or it won't work.

```
a <- c(1,2,3,4)
a
```

```
## [1] 1 2 3 4
```

When you are creating objects, try to give them meaningful names so you can remember what they are. You can't have spaces or operators that mean something else as part of a name. And remember, everything is case sensitive.

Assign the value 5.4 to water_pH and then try to recall it by typing "water_ph"

```
water_pH <- 5.4
```

```
#water_ph
```

You can also set objects equal to strings, or values that have letters in them. To do this you just have to put the value in quotes, otherwise R will think it is an object name and tell you it doesn't exist.

Try: name <- "JP" and then name <- JP

What happens if you forget the ending parenthesis?

Try: name <- "JP

R can be cryptic with its error messages or other responses, but once you get used to them, you know exactly what is wrong when they pop up.

```
name <- "JP"
#name <- JP
```

3.4 Using functions

function_name(arg1 = val1, arg2 = val2)
equivalent: function_name(arg2 = val2, arg1 = val1)
equivalent: function_name(val1, val2)
NOT equivalent: function_name(val2, val1)

You PASS values to function arguments in parentheses after its (CASE SENSITIVE) name.

R knows what values correspond to what arguments by their order, or if you specify using names and =

As an example, let's try the seq() function, which creates a sequence of numbers.

```
seq(from = 1, to = 10, by = 1)
```

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

```
#or

seq(1, 10)

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

#or

seq(1, 10)

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

#what does this do
seq(10,1)

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

3.5 Read in some data.

For the following demonstration we will use the RBI data from a sample of USGS gages we used last class. First we will load the tidyverse library, everything we have done so far is in base R.

Important: `read_csv()` is the tidyverse csv reading function, the base R function is `read.csv()`. `read.csv()` will not read your data in as a tibble, which is the format used by tidyverse functions.

```
library(tidyverse)

rbi <- read_csv("Flashy_Dat_Subset.csv")

## Rows: 49 Columns: 26
## -- Column specification -----
## Delimiter: ","
## chr (4): STANAME, STATE, CLASS, AGGECOREGION
## dbl (22): site_no, RBI, RBIRank, DRAIN_SQKM, HUC02, LAT_GAGE, LNG_GAGE, PPTA...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

3.6 Wait, hold up. What is a tibble?

Good question. It's a fancy way to store data that works well with tidyverse functions. Let's look at the rbi tibble.

```
head(rbi)

## # A tibble: 6 x 26
##   site_no     RBI RBIRank STANAME      DRAIN~1 HUC02 LAT_G~2 LNG_G~3 STATE CLASS
##   <dbl>    <dbl>    <dbl> <chr>      <dbl> <dbl>    <dbl> <dbl> <chr> <chr>
## 1 1013500 0.0584      35 Fish River n~  2253.     1    47.2  -68.6 ME   Ref
## 2 1021480 0.208       300 Old Stream n~   76.7     1    44.9  -67.7 ME   Ref
## 3 1022500 0.198       286 Narraguagus ~  574.     1    44.6  -67.9 ME   Ref
## 4 1029200 0.132       183 Seboeis Rive~  445.     1    46.1  -68.6 ME   Ref
## 5 1030500 0.114       147 Mattawamkeag~ 3676.     1    45.5  -68.3 ME   Ref
## 6 1031300 0.297       489 Piscataquis ~  304.     1    45.3  -69.6 ME   Ref
## # ... with 16 more variables: AGGECOREGION <chr>, PPTAVG_BASIN <dbl>,
## #   PPTAVG_SITE <dbl>, T_AVG_BASIN <dbl>, T_AVG_SITE <dbl>, T_MAX_BASIN <dbl>,
## #   T_MAXSTD_BASIN <dbl>, T_MAX_SITE <dbl>, T_MIN_BASIN <dbl>,
## #   T_MINSTD_BASIN <dbl>, T_MIN_SITE <dbl>, PET <dbl>, SNOW_PCT_PRECIP <dbl>,
## #   PRECIP_SEAS_IND <dbl>, FLOWYRS_1990_2009 <dbl>, wy00_09 <dbl>, and
## #   abbreviated variable names 1: DRAIN_SQKM, 2: LAT_GAGE, 3: LNG_GAGE
## # i Use `colnames()` to see all variable names
```

Now read in the same data with `read.csv()` which will NOT read the data as a tibble. How is it different? Output each one in the Console.

Knowing the data type for each column is super helpful for a few reasons.... let's talk about them.

Types: int, dbl, fctr, char, logical

```
rbi_NT <- read.csv("Flashy_Dat_Subset.csv")

head(rbi_NT)

##   site_no     RBI RBIRank STANAME
## 1 1013500 0.05837454      35 Fish River near Fort Kent, Maine
## 2 1021480 0.20797008      300 Old Stream near Wesley, Maine
## 3 1022500 0.19805382      286 Narraguagus River at Cherryfield, Maine
## 4 1029200 0.13151299      183 Seboeis River near Shin Pond, Maine
## 5 1030500 0.11350485      147 Mattawamkeag River near Mattawamkeag, Maine
## 6 1031300 0.29718786      489 Piscataquis River at Blanchard, Maine
##   DRAIN_SQKM HUC02 LAT_GAGE LNG_GAGE STATE CLASS AGGECOREGION PPTAVG_BASIN
## 1      2252.7     1 47.23739 -68.58264    ME   Ref    NorthEast      97.42
```

```

## 2      76.7    1 44.93694 -67.73611   ME   Ref   NorthEast   115.39
## 3      573.6    1 44.60797 -67.93524   ME   Ref   NorthEast   120.07
## 4      444.9    1 46.14306 -68.63361   ME   Ref   NorthEast   102.19
## 5      3676.2    1 45.50097 -68.30596   ME   Ref   NorthEast   108.19
## 6      304.4    1 45.26722 -69.58389   ME   Ref   NorthEast   119.83
##   PPTAVG_SITE T_AVG_BASIN T_AVG_SITE T_MAX_BASIN T_MAXSTD_BASIN T_MAX_SITE
## 1      93.53     3.00     3.0      9.67      0.202     10.0
## 2     117.13     5.71     5.8     11.70      0.131     11.9
## 3     129.56     5.95     6.3     11.90      0.344     12.2
## 4     103.24     3.61     4.0      9.88      0.231     10.4
## 5     113.13     4.82     5.4     10.75      0.554     11.7
## 6     120.93     3.60     4.2      9.57      0.431     11.0
##   T_MIN_BASIN T_MINSTD_BASIN T_MIN_SITE   PET SNOW_PCT_PRECIP PRECIP_SEAS_IND
## 1      -2.49     0.269     -2.7 504.7      36.9      0.102
## 2      -0.85     0.123     -0.6 554.2      39.5      0.046
## 3       0.06     0.873      1.4 553.1      38.2      0.047
## 4      -2.13     0.216     -1.5 513.0      36.4      0.070
## 5      -1.49     0.251     -1.2 540.8      37.2      0.033
## 6      -2.46     0.268     -1.7 495.8      40.2      0.030
##   FLOWYRS_1990_2009 wy00_09
## 1            20     10
## 2            11     10
## 3            20     10
## 4            11     10
## 5            20     10
## 6            13     10

```

3.7 Data wrangling in dplyr

If you forget syntax or what the following functions do, here is an excellent cheat sheet: <https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>

We will demo five functions below:

- **filter()** - returns rows that meet specified conditions
- **arrange()** - reorders rows
- **select()** - pull out variables (columns)
- **mutate()** - create new variables (columns) or reformat existing ones
- **summarize()** - collapse groups of values into summary stats

With all of these, the first argument is the data and then the arguments after that specify what you want the function to do.

<code>filter(RiverDat, River == "New")</code>	<code>arrange(RiverDat, Flow)</code>	<code>select(RiverDat, River, Flow)</code>																																													
<p>RiverDat</p> <table border="1"> <thead> <tr> <th>River</th><th>Flow</th><th>Year</th></tr> </thead> <tbody> <tr><td>New</td><td>150</td><td>2020</td></tr> <tr><td>New</td><td>175</td><td>2021</td></tr> <tr><td>Roanoke</td><td>50</td><td>2020</td></tr> <tr><td>Roanoke</td><td>62</td><td>2021</td></tr> </tbody> </table>	River	Flow	Year	New	150	2020	New	175	2021	Roanoke	50	2020	Roanoke	62	2021	<p>RiverDat</p> <table border="1"> <thead> <tr> <th>River</th><th>Flow</th><th>Year</th></tr> </thead> <tbody> <tr><td>Roanoke</td><td>50</td><td>2020</td></tr> <tr><td>Roanoke</td><td>62</td><td>2021</td></tr> <tr><td>New</td><td>150</td><td>2020</td></tr> <tr><td>New</td><td>175</td><td>2021</td></tr> </tbody> </table>	River	Flow	Year	Roanoke	50	2020	Roanoke	62	2021	New	150	2020	New	175	2021	<p>RiverDat</p> <table border="1"> <thead> <tr> <th>River</th><th>Flow</th><th>Year</th></tr> </thead> <tbody> <tr><td>New</td><td>150</td><td>2020</td></tr> <tr><td>New</td><td>175</td><td>2021</td></tr> <tr><td>Roanoke</td><td>50</td><td>2020</td></tr> <tr><td>Roanoke</td><td>62</td><td>2021</td></tr> </tbody> </table>	River	Flow	Year	New	150	2020	New	175	2021	Roanoke	50	2020	Roanoke	62	2021
River	Flow	Year																																													
New	150	2020																																													
New	175	2021																																													
Roanoke	50	2020																																													
Roanoke	62	2021																																													
River	Flow	Year																																													
Roanoke	50	2020																																													
Roanoke	62	2021																																													
New	150	2020																																													
New	175	2021																																													
River	Flow	Year																																													
New	150	2020																																													
New	175	2021																																													
Roanoke	50	2020																																													
Roanoke	62	2021																																													
	<p><code>mutate(RiverDat, Flow10 = Flow *10)</code></p> <p>RiverDat</p> <table border="1"> <thead> <tr> <th>River</th><th>Flow</th><th>Year</th><th>Flow10</th></tr> </thead> <tbody> <tr><td>New</td><td>150</td><td>2020</td><td>1500</td></tr> <tr><td>New</td><td>175</td><td>2021</td><td>1750</td></tr> <tr><td>Roanoke</td><td>50</td><td>2020</td><td>500</td></tr> <tr><td>Roanoke</td><td>62</td><td>2021</td><td>620</td></tr> </tbody> </table>	River	Flow	Year	Flow10	New	150	2020	1500	New	175	2021	1750	Roanoke	50	2020	500	Roanoke	62	2021	620	<p><code>DatGrouped <- group_by(RiverDat, River)</code></p> <p><code>summarize(DatGrouped, AvgFlow = mean(Flow))</code></p> <p>RiverDat</p> <table border="1"> <thead> <tr> <th>River</th><th>AvgFlow</th></tr> </thead> <tbody> <tr><td>New</td><td>162.5</td></tr> <tr><td>Roanoke</td><td>56</td></tr> </tbody> </table>	River	AvgFlow	New	162.5	Roanoke	56																			
River	Flow	Year	Flow10																																												
New	150	2020	1500																																												
New	175	2021	1750																																												
Roanoke	50	2020	500																																												
Roanoke	62	2021	620																																												
River	AvgFlow																																														
New	162.5																																														
Roanoke	56																																														

3.8 Filter

Write an expression that returns data in rbi for the state of Maine (ME)

Operators:

`==` equal

`!=` not equal

`>=`, `<=` greater than or equal to, less than or equal to

`>`, `<` greater than or less than

`%in%` included in a list of values

`&` and

`|` or

```
filter(rbi, STATE == "ME")
```

```
## # A tibble: 13 x 26
##   site_no    RBI RBIRank STANAME      DRAIN~1 HUC02 LAT_G~2 LNG_G~3 STATE CLASS
##   <dbl> <dbl> <dbl> <chr>       <dbl> <dbl> <dbl> <dbl> <chr> <chr>
## 1 1013500 0.0584     35 Fish River ~  2253.     1   47.2  -68.6 ME   Ref 
## 2 1021480 0.208      300 Old Stream ~  76.7     1   44.9  -67.7 ME   Ref 
## 3 1022500 0.198      286 Narraguagus~  574.     1   44.6  -67.9 ME   Ref 
## 4 1029200 0.132      183 Seboeis Riv~  445.     1   46.1  -68.6 ME   Ref 
## 5 1030500 0.114      147 Mattawamkea~ 3676.    1   45.5  -68.3 ME   Ref 
## 6 1031300 0.297      489 Piscataquis~ 304.     1   45.3  -69.6 ME   Ref 
## 7 1031500 0.320      545 Piscataquis~ 769     1   45.2  -69.3 ME   Ref 
## 8 1037380 0.318      537 Ducktrap Ri~  39      1   44.3  -69.1 ME   Ref
```

```

##  9 1044550 0.242      360 Spencer Str~   500.      1    45.3   -70.2 ME  Ref
## 10 1047000 0.344      608 Carrabasset~  909.      1    44.9   -70.0 ME  Ref
## 11 1054200 0.492      805 Wild River ~  181      1    44.4   -71.0 ME  Ref
## 12 1055000 0.450      762 Swift River~  251.      1    44.6   -70.6 ME  Ref
## 13 1057000 0.326      561 Little Andr~  191.      1    44.3   -70.5 ME  Ref
## # ... with 16 more variables: AGGECOREGION <chr>, PPTAVG_BASIN <dbl>,
## #   PPTAVG_SITE <dbl>, T_AVG_BASIN <dbl>, T_AVG_SITE <dbl>, T_MAX_BASIN <dbl>,
## #   T_MAXSTD_BASIN <dbl>, T_MAX_SITE <dbl>, T_MIN_BASIN <dbl>,
## #   T_MINSTD_BASIN <dbl>, T_MIN_SITE <dbl>, PET <dbl>, SNOW_PCT_PRECIP <dbl>,
## #   PRECIP_SEAS_IND <dbl>, FLOWYRS_1990_2009 <dbl>, wy00_09 <dbl>, and
## #   abbreviated variable names 1: DRAIN_SQKM, 2: LAT_GAGE, 3: LNG_GAGE
## # i Use `colnames()` to see all variable names

```

3.8.1 Multiple conditions

How many gages are there in Maine with an rbi greater than 0.25

```
filter(rbi, STATE == "ME" & RBI > 0.25)
```

```

## # A tibble: 7 x 26
##   site_no   RBI RBIRank STANAME      DRAIN~1 HUC02 LAT_G~2 LNG_G~3 STATE CLASS
##   <dbl> <dbl> <dbl> <chr>      <dbl> <dbl> <dbl> <dbl> <chr> <chr>
## 1 1031300 0.297     489 Piscataquis R~   304.      1    45.3   -69.6 ME  Ref
## 2 1031500 0.320     545 Piscataquis R~   769      1    45.2   -69.3 ME  Ref
## 3 1037380 0.318     537 Ducktrap Rive~   39       1    44.3   -69.1 ME  Ref
## 4 1047000 0.344     608 Carrabassett ~  909.      1    44.9   -70.0 ME  Ref
## 5 1054200 0.492     805 Wild River at~  181      1    44.4   -71.0 ME  Ref
## 6 1055000 0.450     762 Swift River n~  251.      1    44.6   -70.6 ME  Ref
## 7 1057000 0.326     561 Little Andros~  191.      1    44.3   -70.5 ME  Ref
## # ... with 16 more variables: AGGECOREGION <chr>, PPTAVG_BASIN <dbl>,
## #   PPTAVG_SITE <dbl>, T_AVG_BASIN <dbl>, T_AVG_SITE <dbl>, T_MAX_BASIN <dbl>,
## #   T_MAXSTD_BASIN <dbl>, T_MAX_SITE <dbl>, T_MIN_BASIN <dbl>,
## #   T_MINSTD_BASIN <dbl>, T_MIN_SITE <dbl>, PET <dbl>, SNOW_PCT_PRECIP <dbl>,
## #   PRECIP_SEAS_IND <dbl>, FLOWYRS_1990_2009 <dbl>, wy00_09 <dbl>, and
## #   abbreviated variable names 1: DRAIN_SQKM, 2: LAT_GAGE, 3: LNG_GAGE
## # i Use `colnames()` to see all variable names

```

3.9 Arrange

Arrange sorts by a column in your dataset.

Sort the rbi data by the RBI column in ascending and then descending order

```

arrange(rbi, RBI)

## # A tibble: 49 x 26
##   site_no   RBI RBIrank STANAME      DRAIN~1 HUC02 LAT_G~2 LNG_G~3 STATE CLASS
##   <dbl>   <dbl>    <dbl> <chr>      <dbl> <dbl>    <dbl> <dbl> <chr> <chr>
## 1 1305500 0.0464     18 SWAN RIVER ~    21.3     2    40.8 -73.0  NY  Non-- 
## 2 1013500 0.0584     35 Fish River ~  2253.     1    47.2 -68.6  ME  Ref  
## 3 1306460 0.0587     37 CONNETQUOT ~  55.7      2    40.8 -73.2  NY  Non-- 
## 4 1030500 0.114      147 Mattawamke~ 3676.     1    45.5 -68.3  ME  Ref  
## 5 1029200 0.132      183 Seboeis Riv~ 445.      1    46.1 -68.6  ME  Ref  
## 6 1117468 0.172      244 BEAVER RIVE~ 25.3      1    41.5 -71.6  RI  Ref  
## 7 1022500 0.198      286 Narraguagus~ 574.      1    44.6 -67.9  ME  Ref  
## 8 1021480 0.208      300 Old Stream ~ 76.7      1    44.9 -67.7  ME  Ref  
## 9 1162500 0.213      311 PRIEST BROO~ 49.7      1    42.7 -72.1  MA  Ref  
## 10 1117370 0.230      338 QUEEN R AT ~ 50.5      1    41.5 -71.6  RI  Ref 
## # ... with 39 more rows, 16 more variables: AGGECOREGION <chr>,
## #   PPTAVG_BASIN <dbl>, PPTAVG_SITE <dbl>, T_AVG_BASIN <dbl>, T_AVG_SITE <dbl>,
## #   T_MAX_BASIN <dbl>, T_MAXSTD_BASIN <dbl>, T_MAX_SITE <dbl>,
## #   T_MIN_BASIN <dbl>, T_MINSTD_BASIN <dbl>, T_MIN_SITE <dbl>, PET <dbl>,
## #   SNOW_PCT_PRECIP <dbl>, PRECIP_SEAS_IND <dbl>, FLOWYRS_1990_2009 <dbl>,
## #   wy00_09 <dbl>, and abbreviated variable names 1: DRAIN_SQKM, 2: LAT_GAGE,
## #   3: LNG_GAGE
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names

arrange(rbi, desc(RBI))

## # A tibble: 49 x 26
##   site_no   RBI RBIrank STANAME      DRAIN~1 HUC02 LAT_G~2 LNG_G~3 STATE CLASS
##   <dbl>   <dbl>    <dbl> <chr>      <dbl> <dbl>    <dbl> <dbl> <chr> <chr>
## 1 1311500 0.856      1017 VALLEY STREA~ 18.1      2    40.7 -73.7  NY  Non-- 
## 2 1054200 0.492      805 Wild River a~ 181       1    44.4 -71.0  ME  Ref  
## 3 1187300 0.487      800 HUBBARD RIVE~ 53.9      1    42.0 -72.9  MA  Ref  
## 4 1105600 0.484      797 OLD SWAMP RI~ 12.7      1    42.2 -70.9  MA  Non-- 
## 5 1055000 0.450      762 Swift River ~ 251.      1    44.6 -70.6  ME  Ref  
## 6 1195100 0.430      744 INDIAN RIVER~ 14.8      1    41.3 -72.5  CT  Ref  
## 7 1181000 0.420      732 WEST BRANCH ~ 244.      1    42.2 -72.9  MA  Ref  
## 8 1350000 0.414      721 SCHOHARIE CR~ 612.      2    42.3 -74.4  NY  Ref  
## 9 1121000 0.404      710 MOUNT HOPE R~ 70.3      1    41.8 -72.2  CT  Ref  
## 10 1169000 0.395      688 NORTH RIVER ~ 231.      1    42.6 -72.7  MA  Ref 
## # ... with 39 more rows, 16 more variables: AGGECOREGION <chr>,
## #   PPTAVG_BASIN <dbl>, PPTAVG_SITE <dbl>, T_AVG_BASIN <dbl>, T_AVG_SITE <dbl>,
## #   T_MAX_BASIN <dbl>, T_MAXSTD_BASIN <dbl>, T_MAX_SITE <dbl>,
## #   T_MIN_BASIN <dbl>, T_MINSTD_BASIN <dbl>, T_MIN_SITE <dbl>, PET <dbl>,
## #   SNOW_PCT_PRECIP <dbl>, PRECIP_SEAS_IND <dbl>, FLOWYRS_1990_2009 <dbl>,

```

```
## #  wy00_09 <dbl>, and abbreviated variable names 1: DRAIN_SQKM, 2: LAT_GAGE,
## #  3: LNG_GAGE
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable names
```

3.10 Select

There are too many columns! You will often want to do this when you are manipulating the structure of your data and need to trim it down to only include what you will use.

Select Site name, state, and RBI from the rbi data

Note they come back in the order you put them in in the function, not the order they were in in the original data.

You can do a lot more with select, especially when you need to select a bunch of columns but don't want to type them all out. But we don't need to cover all that today. For a taste though, if you want to select a group of columns you can specify the first and last with a colon in between (first:last) and it'll return all of them. Select the rbi columns from site_no to DRAIN_SQKM.

```
select(rbi, STANAME, STATE, RBI)
```

```
## # A tibble: 49 x 3
##   STANAME                      STATE    RBI
##   <chr>                         <chr>  <dbl>
## 1 Fish River near Fort Kent, Maine ME      0.0584
## 2 Old Stream near Wesley, Maine  ME      0.208 
## 3 Narraguagus River at Cherryfield, Maine ME      0.198 
## 4 Seboeis River near Shin Pond, Maine ME      0.132 
## 5 Mattawamkeag River near Mattawamkeag, Maine ME      0.114 
## 6 Piscataquis River at Blanchard, Maine ME      0.297 
## 7 Piscataquis River near Dover-Foxcroft, Maine ME      0.320 
## 8 Ducktrap River near Lincolnville, Maine ME      0.318 
## 9 Spencer Stream near Grand Falls, Maine  ME      0.242 
## 10 Carrabassett River near North Anson, Maine ME      0.344
## # ... with 39 more rows
## # i Use `print(n = ...)` to see more rows
```

```
select(rbi, site_no:DRAIN_SQKM)
```

```
## # A tibble: 49 x 5
##   site_no      RBI RBIRank STANAME          DRAIN_S~1
##   <dbl>     <dbl>    <dbl> <chr>           <dbl>
## 1       1     0.0584      1 Fish River near Fort Kent, Maine
## 2       2     0.208       2 Old Stream near Wesley, Maine
## 3       3     0.198       3 Narraguagus River at Cherryfield, Maine
## 4       4     0.132       4 Seboeis River near Shin Pond, Maine
## 5       5     0.114       5 Mattawamkeag River near Mattawamkeag, Maine
## 6       6     0.297       6 Piscataquis River at Blanchard, Maine
## 7       7     0.320       7 Piscataquis River near Dover-Foxcroft, Maine
## 8       8     0.318       8 Ducktrap River near Lincolnville, Maine
## 9       9     0.242       9 Spencer Stream near Grand Falls, Maine
## 10      10     0.344      10 Carrabassett River near North Anson, Maine
```

```

## 1 1013500 0.0584      35 Fish River near Fort Kent, Maine          2253.
## 2 1021480 0.208       300 Old Stream near Wesley, Maine            76.7
## 3 1022500 0.198       286 Narraguagus River at Cherryfield, Maine    574.
## 4 1029200 0.132       183 Seboeis River near Shin Pond, Maine        445.
## 5 1030500 0.114       147 Mattawamkeag River near Mattawamkeag, Maine 3676.
## 6 1031300 0.297       489 Piscataquis River at Blanchard, Maine     304.
## 7 1031500 0.320       545 Piscataquis River near Dover-Foxcroft, Maine 769
## 8 1037380 0.318       537 Ducktrap River near Lincolnville, Maine   39
## 9 1044550 0.242       360 Spencer Stream near Grand Falls, Maine    500.
## 10 1047000 0.344      608 Carrabassett River near North Anson, Maine 909.
## # ... with 39 more rows, and abbreviated variable name 1: DRAIN_SQKM
## # i Use `print(n = ...)` to see more rows

```

3.11 Mutate

Use mutate to add new columns based on additional ones. Common uses are to create a column of data in different units, or to calculate something based on two columns. You can also use it to just update a column, by naming the new column the same as the original one (but be careful because you'll lose the original one!). I commonly use this when I am changing the datatype of a column, say from a character to a factor or a string to a date.

Create a new column in rbi called T_RANGE by subtracting T_MIN_SITE from T_MAX_SITE

```
mutate(rbi, T_RANGE = T_MAX_SITE - T_MIN_SITE)
```

```

## # A tibble: 49 x 27
##   site_no RBI RBIRank STANAME      DRAIN~1 HUC02 LAT_G~2 LNG_G~3 STATE CLASS
##   <dbl> <dbl> <dbl> <chr>      <dbl> <dbl> <dbl> <dbl> <chr> <chr>
## 1 1013500 0.0584      35 Fish River ~ 2253.    1    47.2  -68.6 ME  Ref
## 2 1021480 0.208       300 Old Stream ~ 76.7     1    44.9  -67.7 ME  Ref
## 3 1022500 0.198       286 Narraguagus~ 574.     1    44.6  -67.9 ME  Ref
## 4 1029200 0.132       183 Seboeis Riv~ 445.     1    46.1  -68.6 ME  Ref
## 5 1030500 0.114       147 Mattawamkeag~ 3676.    1    45.5  -68.3 ME  Ref
## 6 1031300 0.297       489 Piscataquis~ 304.     1    45.3  -69.6 ME  Ref
## 7 1031500 0.320       545 Piscataquis~ 769.     1    45.2  -69.3 ME  Ref
## 8 1037380 0.318       537 Ducktrap Ri~ 39       1    44.3  -69.1 ME  Ref
## 9 1044550 0.242       360 Spencer Str~ 500.     1    45.3  -70.2 ME  Ref
## 10 1047000 0.344      608 Carrabassett~ 909.    1    44.9  -70.0 ME  Ref
## # ... with 39 more rows, 17 more variables: AGGECOREGION <chr>,
## #   PPTAVG_BASIN <dbl>, PPTAVG_SITE <dbl>, T_AVG_BASIN <dbl>, T_AVG_SITE <dbl>,
## #   T_MAX_BASIN <dbl>, T_MAXSTD_BASIN <dbl>, T_MAX_SITE <dbl>,
## #   T_MIN_BASIN <dbl>, T_MINSTD_BASIN <dbl>, T_MIN_SITE <dbl>, PET <dbl>,

```

```
## #  SNOW_PCT_PRECIP <dbl>, PRECIP_SEAS_IND <dbl>, FLOWYRS_1990_2009 <dbl>,
## #  wy00_09 <dbl>, T_RANGE <dbl>, and abbreviated variable names 1: DRAIN_SQKM,
## #  2: LAT_GAGE, 3: LNG_GAGE
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable nam
```

When downloading data from the USGS through R, you have to enter the gage ID as a character, even though they are all made up of numbers. So to practice doing this, update the site_no column to be a character datatype

```
mutate(rbi, site_no = as.character(site_no))
```

```
## # A tibble: 49 x 26
##   site_no     RBI RBIRank STANAME      DRAIN~1 HUC02 LAT_G~2 LNG_G~3 STATE CLASS
##   <chr>     <dbl> <dbl> <chr>      <dbl> <dbl> <dbl> <dbl> <chr> <chr>
## 1 1013500 0.0584     35 Fish River ~ 2253.    1    47.2 -68.6 ME  Ref
## 2 1021480 0.208      300 Old Stream ~ 76.7     1    44.9 -67.7 ME  Ref
## 3 1022500 0.198      286 Narraguagus~ 574.     1    44.6 -67.9 ME  Ref
## 4 1029200 0.132      183 Seboeis Riv~ 445.     1    46.1 -68.6 ME  Ref
## 5 1030500 0.114      147 Mattawamkea~ 3676.    1    45.5 -68.3 ME  Ref
## 6 1031300 0.297      489 Piscataquis~ 304.     1    45.3 -69.6 ME  Ref
## 7 1031500 0.320      545 Piscataquis~ 769      1    45.2 -69.3 ME  Ref
## 8 1037380 0.318      537 Ducktrap Ri~ 39       1    44.3 -69.1 ME  Ref
## 9 1044550 0.242      360 Spencer Str~ 500.     1    45.3 -70.2 ME  Ref
## 10 1047000 0.344     608 Carrabasset~ 909.     1    44.9 -70.0 ME  Ref
## # ... with 39 more rows, 16 more variables: AGGECOREGION <chr>,
## #   PPTAVG_BASIN <dbl>, PPTAVG_SITE <dbl>, T_AVG_BASIN <dbl>, T_AVG_SITE <dbl>,
## #   T_MAX_BASIN <dbl>, T_MAXSTD_BASIN <dbl>, T_MAX_SITE <dbl>,
## #   T_MIN_BASIN <dbl>, T_MINSTD_BASIN <dbl>, T_MIN_SITE <dbl>, PET <dbl>,
## #   SNOW_PCT_PRECIP <dbl>, PRECIP_SEAS_IND <dbl>, FLOWYRS_1990_2009 <dbl>,
## #   wy00_09 <dbl>, and abbreviated variable names 1: DRAIN_SQKM, 2: LAT_GAGE,
## #   3: LNG_GAGE
## # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable nam
```

3.12 Summarize

Summarize will perform an operation on all of your data, or groups if you assign groups.

Use summarize to compute the mean, min, and max rbi

```
summarize(rbi, meanrbi = mean(RBI), maxrbi = max(RBI), minrbi = min(RBI))
```

```
## # A tibble: 1 x 3
```

```
##   meanrbi maxrbi minrbi
##      <dbl>  <dbl>  <dbl>
## 1    0.316  0.856  0.0464
```

Now use the group function to group by state and then summarize in the same way as above

```
rbistate <- group_by(rbi, STATE)
summarize(rbistate, meanrbi = mean(RBI), maxrbi = max(RBI), minrbi = min(RBI))
```

```
## # A tibble: 7 x 4
##   STATE meanrbi maxrbi minrbi
##   <chr>   <dbl>  <dbl>  <dbl>
## 1 CT     0.366  0.430  0.295
## 2 MA     0.367  0.487  0.213
## 3 ME     0.269  0.492  0.0584
## 4 NH     0.336  0.368  0.265
## 5 NY     0.342  0.856  0.0464
## 6 RI     0.201  0.230  0.172
## 7 VT     0.299  0.365  0.231
```

3.13 Multiple operations with pipes

The pipe operator `%>%` allows you to perform multiple operations in a sequence without saving intermediate steps. Not only is this more efficient, but structuring operations with pipes is also more intuitive than nesting functions within functions (the other way you can do multiple operations).

3.13.1 Let's say we want to tell R to make a PB&J sandwich by using the `pbbread()`, `jbread()`, and `joinslices()` functions and the data “ingredients”. If we do this saving each step if would look like this:

```
sando <- pbbread(ingredients)
```

```
sando <- jbread(sando)
```

```
sando <- joinslices(sando)
```

3.13.2 If we nest the functions together we get this

```
joinslice(jbread(pbbread(ingredients)))
```

Efficient... but tough to read/interpret

3.13.3 Using the pipe it would look like this

```
ingredients %>%
  pbbread() %>%
  jbread() %>%
  joinslices()
```

Much easier to follow!

3.13.4 When you use the pipe, it basically takes whatever came out of the first function and puts it into the data argument for the next one

so rbi %>% group_by(STATE) is the same as group_by(rbi, STATE)

Take the groupby and summarize code from above and perform the operation using the pipe

```
rbi %>%
  group_by(STATE) %>%
  summarize(meanrbi = mean(RBI), maxrbi = max(RBI), minrbi = min(RBI))
```

```
## # A tibble: 7 x 4
##   STATE meanrbi maxrbi minrbi
##   <chr>    <dbl>  <dbl>  <dbl>
## 1 CT      0.366  0.430  0.295
## 2 MA      0.367  0.487  0.213
## 3 ME      0.269  0.492  0.0584
## 4 NH      0.336  0.368  0.265
## 5 NY      0.342  0.856  0.0464
## 6 RI      0.201  0.230  0.172
## 7 VT      0.299  0.365  0.231
```

3.14 Save your results to a new tibble

We have just been writing everything to the screen so we can see what we are doing... In order to save anything we do with these functions to work with it later, we just have to use the assignment operator (<-) to store the data.

One kind of awesome thing about the assignment operator is that it works both ways...

`x <- 3` and `3 -> x` do the same thing (WHAT?!)

So you can do the assignment at the beginning of the end of your dplyr workings, whatever you like best.

Use the assignment operator to save the summary table you just made.

```
stateRBIs <- rbi %>%
  group_by(STATE) %>%
  summarize(meanrbi = mean(RBI), maxrbi = max(RBI), minrbi = min(RBI))

# Notice when you do this it doesn't output the result...
# You can see what you did by clickon in stateRBIs in your environment panel
# or just type stateRBIs

stateRBIs

## # A tibble: 7 x 4
##   STATE meanrbi maxrbi minrbi
##   <chr>    <dbl>   <dbl>   <dbl>
## 1 CT      0.366   0.430  0.295
## 2 MA      0.367   0.487  0.213
## 3 ME      0.269   0.492  0.0584
## 4 NH      0.336   0.368  0.265
## 5 NY      0.342   0.856  0.0464
## 6 RI      0.201   0.230  0.172
## 7 VT      0.299   0.365  0.231
```

3.15 What about NAs?

We will talk more about this when we discuss stats, but some operations will fail if there are NA's in the data. If appropriate, you can tell functions like `mean()` to ignore NAs. You can also use `drop_na()` if you're working with a tibble. But be aware if you use that and save the result, `drop_na()` gets rid of the whole row, not just the NA. Because what would you replace it with.... an NA?

```
x <- c(1,2,3,4,NA)
mean(x, na.rm = TRUE)
```

```
## [1] 2.5
```

3.16 What are some things you think I'll ask you to do for the activity next class?

Chapter 4

ACTIVITY Intro Skills

Get this document at the template repository on github: <https://github.com/VT-Hydroinformatics/3-Activity-Intro-Skills>

4.1 Problem 1

Load the tidyverse and lubridate libraries.

Read in the PINE_NFDR_Jan-Mar_2010 csv using `read_csv()`

Make a plot with the date on the x axis, discharge on the y axis. Show the discharge of the two watersheds as a line, coloring by watershed (StationID)

4.2 Problem 2

Make a boxplot to compare the discharge of Pine to NFDR for February 2010.

Hint: use the pipe operator and the `filter()` function.

Hint2: when you filter dates, you have to let R know you're giving it a date. You can do this by using the `mdy()` function from lubridate.

4.3 Problem 3

Read in the Flashy Dat Subset file.

For only sites in ME, NH, and VT: Plot PET (Potential Evapotranspiration) on the X axis and RBI (flashiness index) on the Y axis. Color the points based on what state they are in. Use the classic ggplot theme.

4.4 Problem 4

We want to look at the amount of snow for each site in the flashy dataset. Problem is, we are only given the average amount of total precip (PPTAVG_BASIN) and the percentage of snow (SNOW_PCT_PRECIP).

Create a new column in the dataset called SNOW_AVG_BASIN and make it equal to the average total precip times the percentage of snow (careful with the percentage number).

Make a barplot showing the amount of snow for each site in Maine. Put station name on the x axis and snow amount on the y. You have to add something to geom_bar() to use it for a 2 variable plot... check out the ggplot cheatsheet or do a quick internet search.

The x axis of the resulting plot looks terrible! Can you figure out how to rotate the X axis labels so we can read them?

4.5 Problem 5

Create a new tibble that contains the min, max, and mean PET for each state. Sort the tibble by mean PET from high to low. Give your columns meaningful names within the summarize function or using rename().

Be sure your code outputs the tibble.

4.6 Problem 6

Take the tibble from problem 5. Create a new column that is the Range of the PET (max PET - min PET). Then get rid of the max PET and min PET columns so the tibble just has columns for State, mean PET, and PET range.

Be sure your code outputs the tibble.

Chapter 5

Introduction to Basic Statistics

Get this document and a version with empty code chunks at the template repository on github: <https://github.com/VT-Hydroinformatics/4-Intro-Stats>

```
library(tidyverse)
library(patchwork)

theme_set(theme_classic())
```

5.1 Reading for this section: Statistical Methods in Water Resources: Chapter 1

<https://pubs.usgs.gov/tm/04/a03/tm4a3.pdf>

5.2 Questions for today:

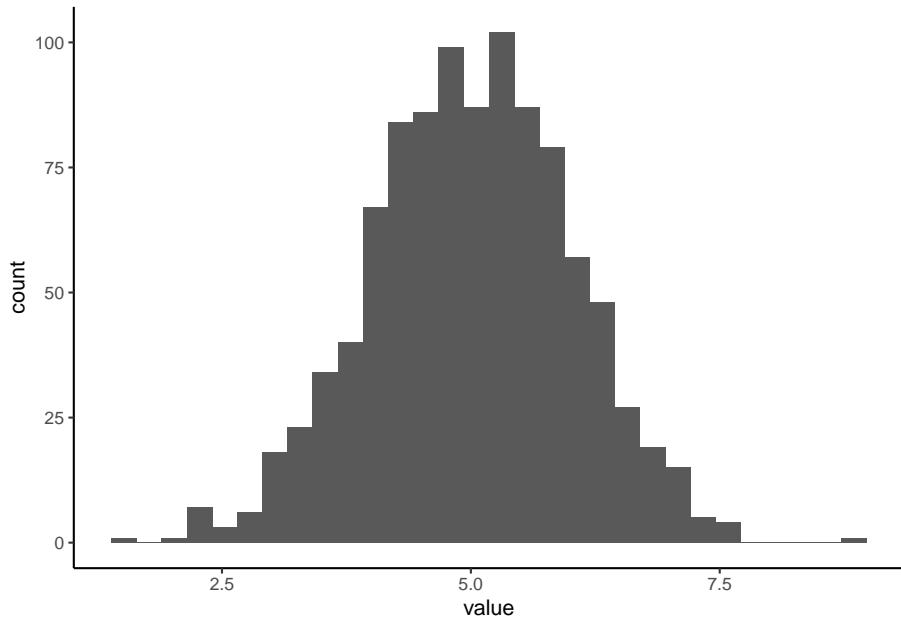
- *What is the difference between a sample and a population?*
- *How do we look at the distribution of data in a sample*
- *How do we measure aspects of a distribution*
- *What is a normal distribution?*

First let's generate some synthetic data and talk about how to visualize it.

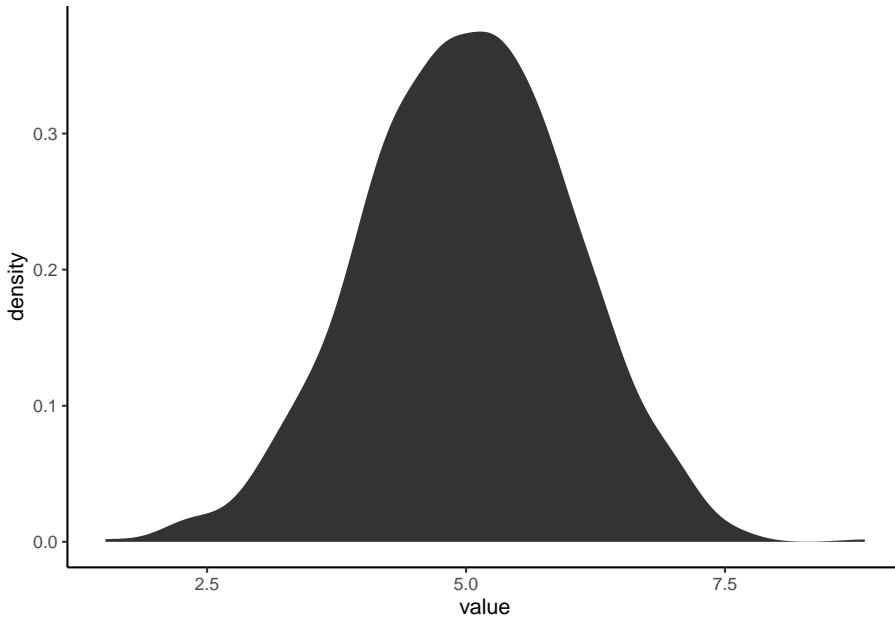
```
#generate a normal distribution
ExNorm <- rnorm(1000, mean = 5) %>%
  as_tibble()

#look at distributions
#histogram
ExNorm %>%
  ggplot(aes(value)) +
  geom_histogram()

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



```
#pdf
ExNorm %>%
  ggplot(aes(value)) +
  stat_density()
```



#Let's generate a plot that makes comparing these two easier

5.2.1 Stack plots to compare histogram and pdf

We will save each plot as ggplot object and then output them using the patchwork package (loaded in the setup chunk).

What is the difference between a histogram and a pdf?

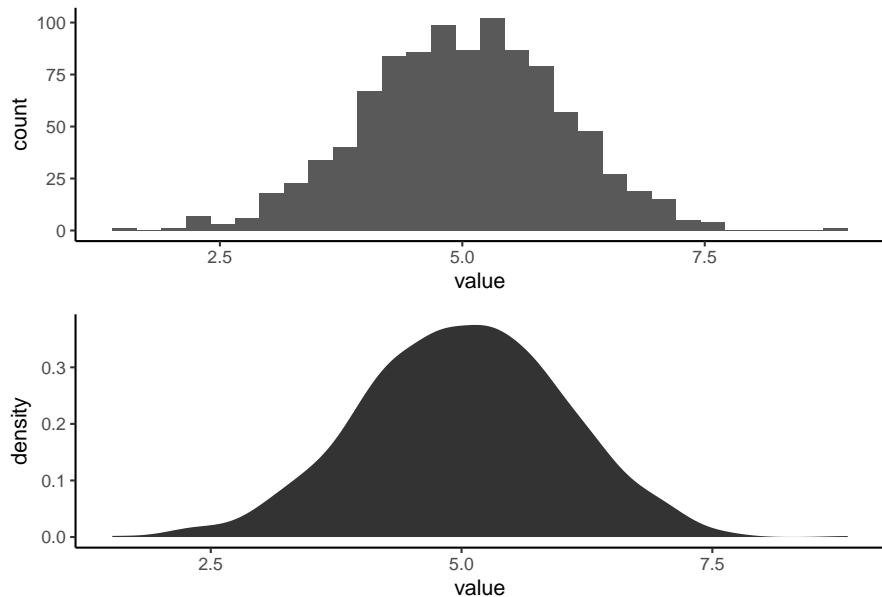
What features of the histogram are preserved? Which are lost?

```
#histogram
exhist <- ExNorm %>%
  ggplot(aes(value)) +
  geom_histogram()

#pdf
expdf <- ExNorm %>%
  ggplot(aes(value)) +
  stat_density()

#put the plots side by side with + or on top of each other with /
exhist/expdf

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



5.3 What is the difference between a sample and a population.

Simply put: a population is the thing you are trying to measure. A sample is the data you measure in an effort to measure the population. A sample is a subset of a population.

Let's write some code for an example:

We will create a POPULATION that is a large set of numbers. Think of this as the concentration of Calcium in every bit of water in a lake. Then we will create a SAMPLE by randomly grabbing values from the POPULATION. This simulates us going around in a boat and taking grab samples in an effort to figure out the concentration of calcium in the lake.

We can then run this code a bunch of times, you'll get a different sample each time. You can also take a smaller or larger number of samples by changing "size" in the `sample()` function.

- How does your sample distribution look similar or different from the population?
- Why does the sample change every time you run it?
- What happens as you increase or decrease the number of samples?
- What happens if you set the number of samples to the size of the population?

5.3. WHAT IS THE DIFFERENCE BETWEEN A SAMPLE AND A POPULATION.51

```
all_the_water <- rnorm(10000, mean = 6) %>% as_tibble()

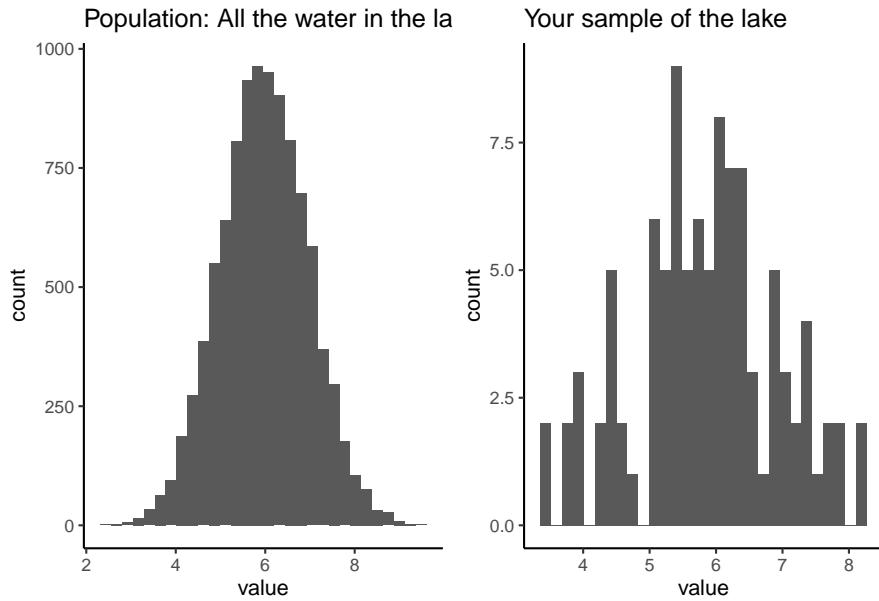
sample_of_water <- sample(all_the_water$value, size = 100, replace = FALSE) %>% as_tibble()

population_hist <- all_the_water %>%
  ggplot(aes(value)) +
  geom_histogram() +
  ggtitle("Population: All the water in the lake")

sample_hist <- sample_of_water %>%
  ggplot(aes(value)) +
  geom_histogram() +
  ggtitle("Your sample of the lake")

population_hist + sample_hist

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



5.4 Measuring our sample distribution: central tendency.

When we take a sample of a population, there are a few things we will want to measure about the distribution of values: where is the middle, how variable is it, and is it skewed to one side or another?

The first of these, “where is the middle?” is addressed with measures of central tendency. We will discuss three possible ways to measure this. The mean, median, and weighted mean.

To explain the importance of choosing between the mean and median, we will first import some discharge data. Read in the PINE discharge data.

```
pineQ <- read_csv("PINE_Jan-Mar_2010.csv")
## Rows: 2160 Columns: 8
## -- Column specification -----
## Delimiter: ","
## chr (2): StationID, surrogate
## dbl (5): cfs, year, quarter, month, day
## dttm (1): datetime
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

To find the mean (average), you just sum up all the values in your sample and divide by the number of values.

To find the median, you put the values IN ORDER, and choose the middle value. The middle value is the one where there are the same number of values higher than that value as there are values lower than it.

Because it uses the order of the values rather than just the values themselves, the median is resistant to skewed distributions. This means it is less effected by very large or very small values compared to most values in the sample data.

Let’s look at our normal distribution from earlier (ExNorm) compared to the Pine watershed discharge (pineQ)

Note that distributions like pineQ, that are positively skewed, are very common in environmental data.

```
#Calculate mean and median for cfs in pineQ and values in ExNorm
pineMean <- mean(pineQ$cfs)
pineMedian <- median(pineQ$cfs)
```

5.4. MEASURING OUR SAMPLE DISTRIBUTION: CENTRAL TENDENCY.53

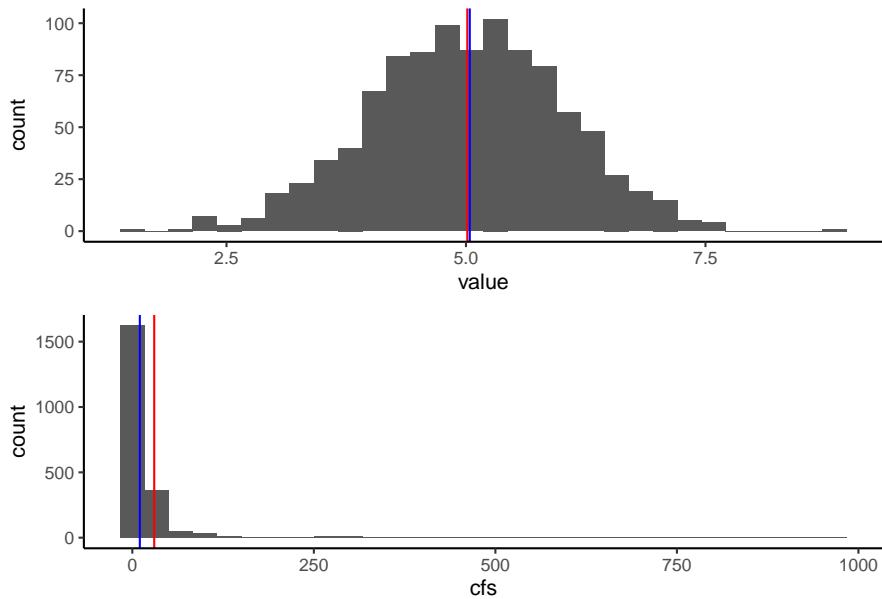
```
xmean <- mean(ExNorm$value)
xmedian <- median(ExNorm$value)

#plot mean and median on the ExNorm distribution
Ex <- ExNorm %>% ggplot(aes(value)) +
  geom_histogram()+
  geom_vline(xintercept = xmean, color = "red")+
  geom_vline(xintercept = xmedian, color = "blue")

#plot mean and median on the pineQ discharge histogram
PineP <- pineQ %>% ggplot(aes(cfs)) +
  geom_histogram()+
  geom_vline(xintercept = pineMean, color = "red")+
  geom_vline(xintercept = pineMedian, color = "blue")

Ex / PineP

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



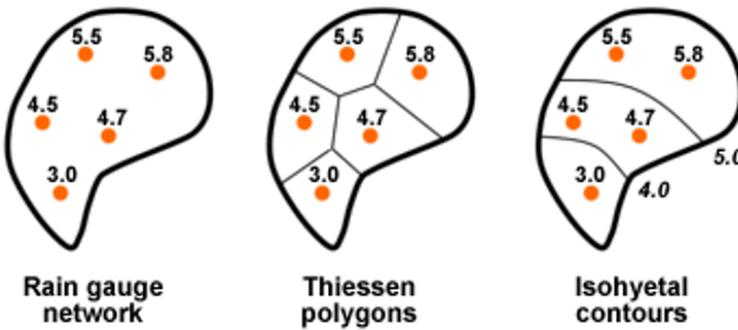
5.4.1 So what's a weighted average?

When you compute a standard mean or median, you are giving equal weight to each measurement. Adding up all the values in a sample and dividing by the number of samples is the same as multiplying each value by $1/\#$ of samples. For instance if you had ten samples, to calculate the mean you would add them up and divide by 10. This is the same as multiplying each value by $1/10$ and then adding them up. Each value is equally weighted at $1/10$.

There are certain situations in which this is not the ideal way to calculate an average. A common one in hydrology is that you have samples that are supposed to represent different portions of an area. One sample may be taken to measure a forest type that takes up 100 ha of a watershed while another sample represents a forest type that only takes up 4 ha. You may not want to simply average those values!

Another example is precipitation gages. In the image below, you see there are 5 rain gages. To get a precipitation number for the watershed, we could just average them, or we could assume they represent an area of the watershed and then weight their values by the area they represent. One method of designating the areas is by using Thiessen polygons (the middle watershed). Another method of weighting is isohyetal contours, but we won't worry about that for now!

In the weighted situation, we find the average by multiplying each precipitation values by the proportion of the watershed it represents, shown by the Thiessen polygons, and then add them all together. Let's do an example.



©The COMET Program

source:

https://edx.hydrolearn.org/assets/courseware/v1/e5dc65098f1e8c5faacae0e171e28ccf/asset-v1:HydroLearn+HydroLearn401+2019_S2+type@asset+block/l2_image004.png

The precip values for the watershed above are 4.5, 5.5, 5.8, 4.7, and 3.0

We will assume the proportions of the watershed that each gauge represents are 0.20, 0.15, 0.40, 0.15, 0.10, respectively (or 20%, 15%, 40%, 15%, 10%)

Write some code to compute the regular mean precip from the values, and then the weighted mean.

```
precip <- c(4.5, 5.5, 5.8, 4.7, 3.0)
weights <- c(0.2, 0.15, 0.4, 0.15, 0.1)

mean(precip)

## [1] 4.7

sum(precip * weights)

## [1] 5.05
```

5.5 Measures of variability

Measures of variability allow us to measure the width of our sample data histogram or pdf. If all the values in our sample are close together, we would have small measures of variability, and a pointy pdf/histogram. If they vary more, we would have larger measures of variability and a broad pdf/histogram.

We will explore four measures of variability:

5.5.0.1 Variance:

Sum of the squared difference of each value from the mean divided by the number of samples minus 1. var()

$$s^2 = \sum_{i=1}^n \frac{(X_i - \bar{X})^2}{(n-1)}$$

(<https://pubs.usgs.gov/tm/04/a03/tm4a3.pdf>) source: <https://pubs.usgs.gov/tm/04/a03/tm4a3.pdf>

5.5.0.2 Standard deviation:

The square root of the variance sd()

**Both variance and standard deviation are sensitive to outliers.

5.5.0.3 CV: Coefficient of Variation

CV is simply the standard deviation divided by the mean of the data. Because you divide by the mean, CV is dimensionless. This allows you to use it to compare the variation in samples with very different magnitudes.

5.5.0.4 IQR: Interquartile Range

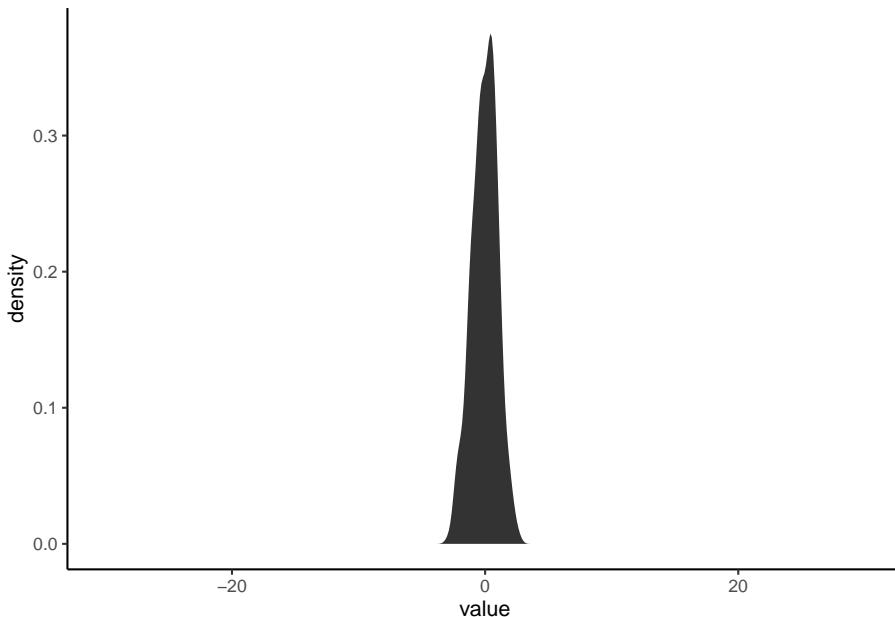
IQR is resistant to outliers because it works like a median. It measures the range of the middle 50% of the data in your distribution. So the IQR is the difference between the value between the 75th and 25th percentiles of your data, where the 75th percentile means 75% of the data is BELOW that value and the 25th percentile means 25% is below that value. Using the same vocabulary, the median is the same as the 50th percentile of the data.

If you ask R for the QUANTILES of your sample data, it will give you the values at which 0%, 25%, 50%, 75%, and 100% of the data are below. These are the 1,2,3,4, and 5th quantiles. Therefore, the IQR is the difference between the 4th and 2nd quantile.

Okay, code time.

First, let's explore how changing the variability of a distribution changes the shape of it's distribution. Create a plot a random normal distribution using `rnorm()` and set `sd` to different numbers. Make the mean of the distribution 0, the sample size 300, and the standard deviation 1 to start. Then increase the standard deviation incrementally to 10 and see what happens. Make the limits of the x axis on the plot -30 to 30.

```
rnorm(300, mean = 0, sd = 1) %>% as_tibble %>%
  ggplot(aes(value)) +
  stat_density() +
  xlim(c(-30, 30))
```



Now let's calculate the standard deviation, variance, coefficient of variation, and IQR of the Pine discharge data.

```
#standard deviation
sd(pineQ$cfs)

## [1] 84.47625

#variance
var(pineQ$cfs)

## [1] 7136.237

#coefficient of variation
sd(pineQ$cfs)/mean(pineQ$cfs)

## [1] 2.800221

#IQR using the IQR function
IQR(pineQ$cfs)

## [1] 8.1325
```

```
#IQR using the quantile function
quants <- quantile(pineQ$cfs)
```

```
quants[4] - quants[2]
```

```
##      75%
## 8.1325
```

5.5.0.5 What about how lopsided the distribution is?

There are several ways to measure this as well, but we are just going to look at one: The Quartile skew. The quartile skew is the difference between the upper quartiles (50th-75th) and the lower quartiles (25th-50th) divided by the IQR (75th-25th).

$$qs = \frac{(P_{0.75} - P_{0.50}) - (P_{0.50} - P_{0.25})}{P_{0.75} - P_{0.25}}$$

source: <https://pubs.usgs.gov/tm/04/a03/tm4a3.pdf>

Let's look at the quartile skew of the two distributions we've been measuring. Calculate it for the pineQ discharge data and the random normal distribution we generated.

Which one is more skewed?

```
quantsP <- quantile(pineQ$cfs)
```

```
((quantsP[3]-quantsP[2]) - (quantsP[2] - quantsP[1])) / quantsP[3] - quantsP[1]
```

```
##      50%
## -4.837233
```

```
quantsX <- quantile(ExNorm$value)
```

```
((quantsX[3]-quantsX[2]) - (quantsX[2] - quantsX[1])) / quantsX[3] - quantsX[1]
```

```
##      50%
## -1.933978
```

5.6 What is a normal distribution and how can we determine if we have one?

The distribution we generated with `rnorm()` is a normal distribution. The distribution of `pineQ` discharge is not normal. Now that we've looked at different ways to characterize distributions, we have the vocabulary to describe why.

Normal distributions:

- mean = median, half values to the right, half to the left
- symmetric (not skewed)
- single peak

Many statistical tests require that the distribution of the data you put into them is normally distributed. BE CAREFUL! There are also tests that use ranked data. Similar to how the median is resistant to outliers, these rank-based tests are resistant to non-normal data. Two popular ones are Kruskal-Wallis and Wilcoxon rank-sum.

But how far off can you be before you don't consider a distribution normal? Seems like a judgement call!

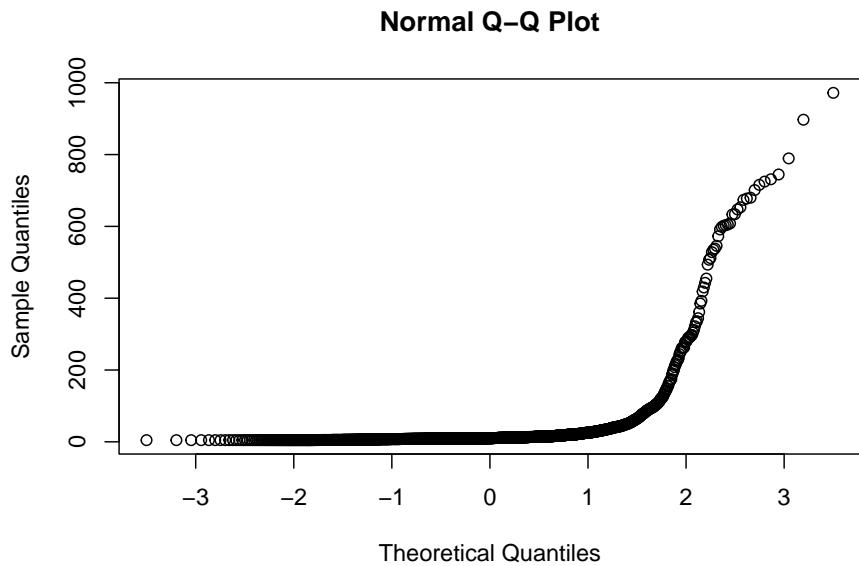
R to the rescue! There is a built in test for normality called `shapiro.test()`, which performs the Shapiro-Wilk test of normality. The hypothesis this test tests is “The distribution is normal.” So if this function returns a p-value less than 0.05, you reject that hypothesis and your function is NOT normal.

You can also make a quantile-quantile plot. A straight line on this plot indicates a normal distribution, a non-straight line indicates it is not normal.

```
shapiro.test(pineQ$cfs)
```

```
##  
## Shapiro-Wilk normality test  
##  
## data: pineQ$cfs  
## W = 0.27155, p-value < 2.2e-16
```

```
qqnorm(pineQ$cfs)
```



Chapter 6

ACTIVITY Intro Stats

Get this document at the template repository on github: <https://github.com/VT-Hydroinformatics/5-Intro-Stats-Activity>

Address each of the questions in the code chunk below and/or by typing outside the chunk (for written answers).

6.1 Problem 1

Load the tidyverse and patchwork libraries and read in the Flashy and Pine datasets.

6.2 Problem 2

Using the flashy dataset, make a pdf of the average basin rainfall (PP-TAVG_BASIN) for the NorthEast AGGECOREGION. On that pdf, add vertical lines showing the mean, median, standard deviation, and IQR. Make each a different color and note which is which in a typed answer below this question. (or if you want an extra challenged, make a custom legend that shows this)

6.3 Problem 3

Perform a Shapiro-Wilk test for normality on the data from question 2. Using the results from that test and the plot and stats from question 2, discuss whether or not the distribution is normal.

6.4 Problem 4

Make a plot that shows the distribution of the data from the PINE watershed and the NFDR watershed (two pdfs on the same plot). Log the x axis.

6.5 Problem 5

You want to compare how variable the discharge is in each of the watersheds in question 4. Which measure of spread would you use and why? If you wanted to measure the central tendency which measure would you use and why?

6.6 Problem 6

Compute 3 measures of spread and 2 measures of central tendency for the PINE and NFDR watershed. (hint: use group_by() and summarize()) Be sure your code outputs the result. Which watershed has higher flow? Which one has more variable flow? How do you know?

Chapter 7

Joins, Pivots, and USGS dataRetrieval

Use Template Repository from github:

Readings: Introduction to the dataRetrieval package <https://cran.r-project.org/web/packages/dataRetrieval/vignettes/dataRetrieval.html>

Chapter 12 & 13 of R for Data Science <https://r4ds.had.co.nz/tidy-data.html>

7.1 Goals for today

- Get familiar with the dataRetrieval package
- Intro to joins
- Learn about long vs. wide data and how to change between them

Prep question: How would you get data from the USGS NWIS (non-R)?

Install the dataRetrieval package. Load it and the tidyverse.

```
#install.packages("dataRetrieval")
library(dataRetrieval)

## Warning: package 'dataRetrieval' was built under R version 4.1.2

library(tidyverse)
library(lubridate)
```

7.2 Exploring what dataRetrieval can do.

Think about the dataRetrieval as a way to interact with same public data you can access through waterdata.usgs.gov but without having to click on buttons and search around. It makes getting data or doing analyses with USGS data much more reproducible and fast!

To explore a few of the capabilities (NOT ALL!!) we will start with the USGS gage on the New River at Radford. The gage number is 03171000.

The documentation for the package is extremely helpful: <https://cran.r-project.org/web/packages/dataRetrieval/vignettes/dataRetrieval.html>

I always have to look up how to do things because the package is very specialized! This is the case with most website APIs, in my experience. It's a good argument for getting good at navigating package documentation! Basically you just look through and try to piece together the recipe for what you want to do using the examples they give in the document.

First, let's get information about the site using the `readNWISsite()` and `whatNWISdata()` functions. Try each out and see what they tell you.

Remember, all the parameter codes and site names get passed to `dataRetrieval` functions as characters, so they must be in quotes.

```
#important: note the site number gets input as a character
site <- "03171000"

#Information about the site
siteinfo <- readNWISsite(site)

#What data is available for the site?
#Daily values, mean values
dataAvailable <- whatNWISdata(siteNumber = site, service = "dv", statCd = "00003")

dataAvailable

##   agency_cd site_no          station_nm site_tp_cd dec_lat_va dec_long_va
## 2      USGS 03171000 NEW RIVER AT RADFORD, VA           ST 37.14179 -80.56922
## 3      USGS 03171000 NEW RIVER AT RADFORD, VA           ST 37.14179 -80.56922
## 4      USGS 03171000 NEW RIVER AT RADFORD, VA           ST 37.14179 -80.56922
##   coord_acy_cd dec_coord_datum_cd alt_va alt_acy_va alt_datum_cd huc_cd
## 2            U             NAD83 1711.99     0.13      NAVD88 05050001
## 3            U             NAD83 1711.99     0.13      NAVD88 05050001
## 4            U             NAD83 1711.99     0.13      NAVD88 05050001
##   data_type_cd parm_cd stat_cd ts_id loc_web_ds medium_grp_cd parm_grp_cd
## 2           dv 00010 00003 241564       NA      wat      <NA>
## 3           dv 00060 00003 145684       NA      wat      <NA>
```

```

## 4      dv 00095 00003 145685      NA      wat <NA>
##   srs_id access_cd begin_date   end_date count_nu
## 2 1645597          0 2006-12-20 2009-03-18     704
## 3 1645423          0 1907-10-01 2023-02-05   33365
## 4 1646694          0 2006-12-20 2008-09-29     534

```

7.3 Joins

When we look at what whatNWISdata returns, we see it gives us parameter codes, but doesn't tell us what they mean. This is a common attribute of databases: you use a common identifier but then have the full information in a lookup file. In this case, the look-up information telling us what the parameter codes mean is in "parameterCdFile" which loads with the dataRetrieval package.

So, you could look at that and see what the parameters mean.

OR We could have R do it and add a column that tells us what the parameters mean. Enter JOINS!

Joins allow us to combine the data from two different data sets that have a column in common. At its most basic, a join looks for a matching row with the same key in both datasets (for example, a USGS gage number) and then combines the rows. So now you have all the data from both sets, matched on the key.

But you have to make some decisions: what if a key value exists in one set but not the other? Do you just drop that observation? Do you add an NA? Let's look at the different options.

Take for example the two data sets, FlowTable and SizeTable. The SiteName values are the key values and the MeanFlow and WSsize values are the data.

FlowTable		SizeTable	
SiteName	MeanFlow	SiteName	WSsize
River1	100	River1	800
River2	125	River2	950
River3	80	River5	700

Figure 7.1: Join Setup

Note River1 and River2 match up, but River3 and River5 only exist in one data set or the other.

The first way to deal with this is an **INNER JOIN: inner_join()** In an inner join, you only keep records that match. So the rows for River3 and River5 will be dropped because there is no corresponding data in the other set. See below:

INNER JOIN

FlowTable		SizeTable	
SiteName	MeanFlow	SiteName	WSsize
River1	100	River1	800
River2	125	River2	950
River3	80	River5	700

FlowSizeTable <- inner_join(FlowTable, SizeTable, by = "SiteName")
--

FlowSizeTable		
SiteName	MeanFlow	WSsize
River1	100	800
River2	125	950

Figure 7.2: Inner Join

But what if you don't want to lose the values in one or the other or both?!

For instance, let's say you have a bunch of discharge data for a stream, and then chemistry grab samples. You want to join the chemistry to the discharge based on the dates and times they were taken. But when you do this, you don't want to delete all the discharge data where there is no chemistry! We need another option. Enter OUTER JOINS

LEFT JOIN, left_join(): Preserves all values from the LEFT data set, and pastes on the matching ones from the right. This creates NAs where there is a value on the left but not the right. (this is what you'd want to do in the discharge - chemistry example above)

RIGHT JOIN, right_join(): Preserves all values from the RIGHT data set, and pastes on the matching ones from the left. This creates NAs where there is a value on the right but not the left.

FULL JOIN, full_join(): KEEP EVERYTHING! The hoarder of the joins. No matching record on the left? create an NA on the right! No matching value on the right? Create an NA on the left! NAs for everyone!

When you do this in R, you use the functions identified in the descriptions with the following syntax (see example below):

LEFT JOIN

FlowTable		SizeTable	
SiteName	MeanFlow	SiteName	WSsize
River1	100	River1	800
River2	125	River2	950
River3	80	River5	700

FlowSizeTable <- left_join(FlowTable, SizeTable, by = "SiteName")		
FlowSizeTable		
SiteName	MeanFlow	WSsize
River1	100	800
River2	125	950
River3	80	NA

Figure 7.3: Left Join

RIGHT JOIN

FlowTable		SizeTable	
SiteName	MeanFlow	SiteName	WSsize
River1	100	River1	800
River2	125	River2	950
River3	80	River5	700

FlowSizeTable <- right_join(FlowTable, SizeTable, by = "SiteName")		
FlowSizeTable		
SiteName	MeanFlow	WSsize
River1	100	800
River2	125	950
River5	NA	700

Figure 7.4: Right Join

FULL JOIN

FlowTable		SizeTable	
SiteName	MeanFlow	SiteName	WSsize
River1	100	River1	800
River2	125	River2	950
River3	80	River5	700

```
FlowSizeTable <- full_join(FlowTable, SizeTable, by = "SiteName")
```

FlowSizeTable		
SiteName	MeanFlow	WSsize
River1	100	800
River2	125	950
River3	80	NA
River5	NA	700

Figure 7.5: Full Join

if the column is named the same in both data sets > `xxx_left_join(left_tibble, right_tibble, by = "key_column")**`

if the column is named differently in both data sets > `xxx_left_join(left_tibble, right_tibble, by = c("left_key" = "right_key"))`

**LEFT JOIN
(unmatched names)**

FlowTable		SizeTable	
Site	MeanFlow	Name	WSsize
River1	100	River1	800
River2	125	River2	950
River3	80	River5	700

<code>FlowSizeTable <- left_join(FlowTable, SizeTable, by = c("Site" = "SiteName"))</code>

Site	MeanFlow	WSsize
River1	100	800
River2	125	950
River3	80	NA

Figure 7.6: Left Join Differing Col Names

Note in both of the above, when you specify which column to use as “by” you have to put it in quotes.

7.4 Join example

So in the chunk below let’s get add information about the parameters in dataAvailable by joining it with the key file: parameterCdFile. The column with the parameter codes is called parm_cd in dataAvailable and parameter_cd in parameterCdFile

```
dataAvailable <- left_join(dataAvailable, parameterCdFile,
                           by = c("parm_cd" = "parameter_cd"))

dataAvailable

##   agency_cd site_no          station_nm site_tp_cd dec_lat_va dec_long_va
## 1      USGS 03171000 NEW RIVER AT RADFORD, VA           ST 37.14179 -80.56922
## 2      USGS 03171000 NEW RIVER AT RADFORD, VA           ST 37.14179 -80.56922
## 3      USGS 03171000 NEW RIVER AT RADFORD, VA           ST 37.14179 -80.56922
```

```

##   coord_acy_cd dec_coord_datum_cd  alt_va alt_acy_va alt_datum_cd    huc_cd
## 1           U             NAD83 1711.99      0.13      NAVD88 05050001
## 2           U             NAD83 1711.99      0.13      NAVD88 05050001
## 3           U             NAD83 1711.99      0.13      NAVD88 05050001
##   data_type_cd parm_cd stat_cd  ts_id loc_web_ds medium_grp_cd parm_grp_cd
## 1           dv 00010 00003 241564       NA      wat     <NA>
## 2           dv 00060 00003 145684       NA      wat     <NA>
## 3           dv 00095 00003 145685       NA      wat     <NA>
##   srs_id access_cd begin_date   end_date count_nu parameter_group_nm
## 1 1645597        0 2006-12-20 2009-03-18      704      Physical
## 2 1645423        0 1907-10-01 2023-02-05    33365      Physical
## 3 1646694        0 2006-12-20 2008-09-29      534      Physical
##
## 1                                     Temperature, water, degrees
## 2                                     Discharge, cubic feet per
## 3 Specific conductance, water, unfiltered, microsiemens per centimeter at 25 degrees
##   casrn          srsname parameter_units
## 1           Temperature, water      deg C
## 2       Stream flow, mean. daily      ft3/s
## 3      Specific conductance      uS/cm @25C

```

```

#that made a lot of columns, let's clean it up
dataAvailClean <- dataAvailable %>% select(site_no,
                                              station_nm,
                                              parm_cd,
                                              srsname,
                                              parameter_units,
                                              begin_date,
                                              end_date)

```

```
dataAvailClean
```

```

##   site_no          station_nm  parm_cd          srsname
## 1 03171000 NEW RIVER AT RADFORD, VA 00010      Temperature, water
## 2 03171000 NEW RIVER AT RADFORD, VA 00060 Stream flow, mean. daily
## 3 03171000 NEW RIVER AT RADFORD, VA 00095      Specific conductance
##   parameter_units begin_date   end_date
## 1           deg C 2006-12-20 2009-03-18
## 2           ft3/s 1907-10-01 2023-02-05
## 3          uS/cm @25C 2006-12-20 2008-09-29

```

7.5 Finding IDs to download USGS data

You can find sites via map and just enter the id like we did in the chunks above:
<https://maps.waterdata.usgs.gov/mapper/index.html>

Below we will look at two other ways to get sites: using a bounding box of a geographic region, or search terms like State and drainage area

```
#find sites in a bounding box
#coords of bottom left, top right
swva <- c(-81.36, 36.72, -80.27, 37.32)

#get sites in this bounding box that have daily water temperature and discharge
swva_sites <- whatNWISsites(bBox = swva,
                             parameterCd = c("00060", "00010"),
                             hasDataTypeCd = "dv")

swva_sites
```

## agency_cd site_no	station_nm
## 1 USGS 03473500	M F HOLSTON RIVER AT GROSECLOSE, VA
## 2 USGS 03175140	WEST FORK COVE CREEK NEAR BLUEFIELD, VA
## 3 USGS 03177710	BLUESTONE RIVER AT FALLS MILLS, VA
## 4 USGS 03177700	BLUESTONE RIVER AT BLUEFIELD, VA
## 5 USGS 03166000	CRIPPLE CREEK NEAR IVANHOE, VA
## 6 USGS 03164500	NEW RIVER NEAR GRAYSON, VA
## 7 USGS 03165500	NEW RIVER AT IVANHOE, VA
## 8 USGS 03166880	WEST SP AT NAT FISH HAT NEAR GRAHAMS FORGE, VA
## 9 USGS 03166800	GLADE CREEK AT GRAHAMS FORGE, VA
## 10 USGS 03166900	BOILING SP AT NAT FISH HAT NR GRAHAMS FORGE, VA
## 11 USGS 03167000	REED CREEK AT GRAHAMS FORGE, VA
## 12 USGS 03175500	WOLF CREEK NEAR NARROWS, VA
## 13 USGS 03168500	PEAK CREEK AT PULASKI, VA
## 14 USGS 03168000	NEW RIVER AT ALLISONIA, VA
## 15 USGS 03167500	BIG REED ISLAND CREEK NEAR ALLISONIA, VA
## 16 USGS 03172500	WALKER CREEK AT STAFFORDSVILLE, VA
## 17 USGS 03173000	WALKER CREEK AT BANE, VA
## 18 USGS 03171500	NEW RIVER AT EGGLESTON, VA
## 19 USGS 03171000	NEW RIVER AT RADFORD, VA
## 20 USGS 03170000	LITTLE RIVER AT GRAYSONTOWN, VA
## 21 USGS 03169500	LITTLE RIVER NEAR COPPER VALLEY, VA
## site_tp_cd dec_lat_va dec_long_va colocated	queryTime
## 1 ST 36.88873 -81.34733 FALSE	2023-02-06 14:55:03
## 2 ST 37.18428 -81.32982 FALSE	2023-02-06 14:55:03
## 3 ST 37.27151 -81.30482 FALSE	2023-02-06 14:55:03
## 4 ST 37.25595 -81.28177 FALSE	2023-02-06 14:55:03

```

## 5      ST  36.85984 -80.98036 FALSE 2023-02-06 14:55:03
## 6      ST  36.75985 -80.95619 FALSE 2023-02-06 14:55:03
## 7      ST  36.83485 -80.95258 FALSE 2023-02-06 14:55:03
## 8      SP  36.93429 -80.90313 FALSE 2023-02-06 14:55:03
## 9      ST  36.93095 -80.90036 FALSE 2023-02-06 14:55:03
## 10     SP  36.93068 -80.89619 FALSE 2023-02-06 14:55:03
## 11     ST  36.93901 -80.88730 FALSE 2023-02-06 14:55:03
## 12     ST  37.30568 -80.84980 FALSE 2023-02-06 14:55:03
## 13     ST  37.04721 -80.78472 FALSE 2023-02-06 14:55:03
## 14     ST  36.93762 -80.74563 FALSE 2023-02-06 14:55:03
## 15     ST  36.88901 -80.72757 FALSE 2023-02-06 14:55:03
## 16     ST  37.24179 -80.71090 FALSE 2023-02-06 14:55:03
## 17     ST  37.26818 -80.70951 FALSE 2023-02-06 14:55:03
## 18     ST  37.28957 -80.61673 FALSE 2023-02-06 14:55:03
## 19     ST  37.14179 -80.56922 FALSE 2023-02-06 14:55:03
## 20     ST  37.03763 -80.55672 FALSE 2023-02-06 14:55:03
## 21     ST  36.99652 -80.52144 FALSE 2023-02-06 14:55:03

#find sites with other criteria, VA, less than 20 sqmi, other criteria can be used..
#check out the CRAN documentation
smallVA <- readNWISdata(service = "dv",
                        stateCd = "VA",
                        parameterCd = "00060",
                        drainAreaMax = "20",
                        statCd = "00003")

```

7.6 OK let's download some data!

We are going to use `readNWISdv()`, which downloads daily values.

We will tell it which sites to download, which parameters to download, and then what time period to download.

`siteNumber` gets the sites we want to download, USGS site numbers, as a character. We will use the `swva_sites` data we generated (yep, you can download multiple sites at once!)

`startDate` and `endDate` get the.... start and end dates. IMPORTANT: These must be in YYYY-MM-DD format, but you don't have to tell R they are dates before you give them to the function, it'll do that for you.

`parameterCd` is the parameters you want to download. We want water temperature and discharge, which are "00060" and "00010", respectively.

Once we have the data, the column names correspond to the keys that identify them, for example, discharge will be 00060 something something. Fortunately

the dataRetrieval package also provides “`renameNWISColumns()`” which translates these into words, making them more easily understood by humans. We can pipe the results of our download to that function after we get the data to make the column names easier to understand.

```
start <- "2006-10-01"
end <- "2008-09-30"
params <- c("00010", "00060")

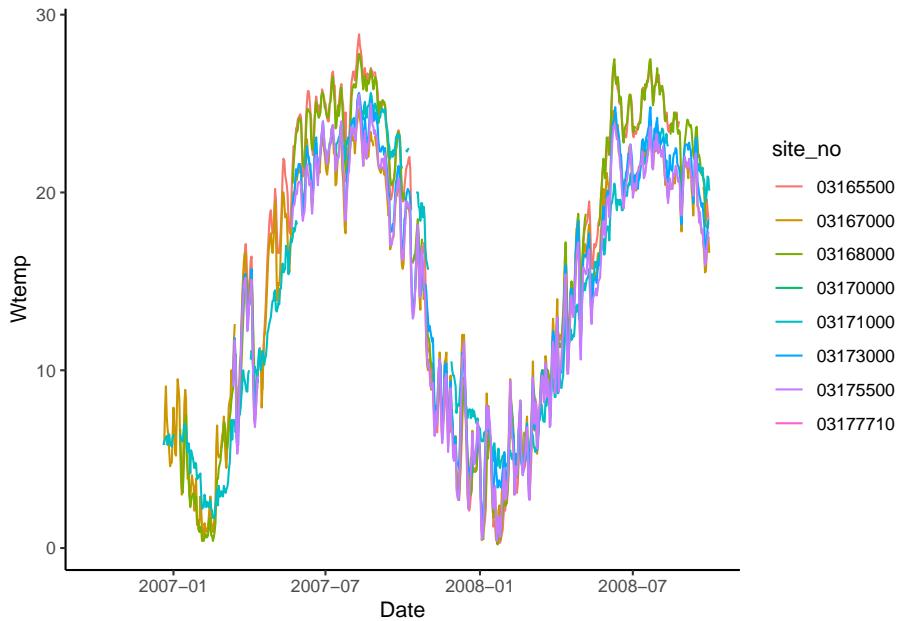
swva_dat <- readNWISdv(siteNumber = swva_sites$site_no,
                        parameterCd = params,
                        startDate = start,
                        endDate = end) %>%
  renameNWISColumns()
```

Let's plot the water temperature data as a line and control the color of the lines with the different sites.

What could be better about this plot?

```
swva_dat %>% ggplot(aes(x = Date, y = Wtemp, color = site_no)) +
  geom_line()

## Warning: Removed 2218 row(s) containing missing values (geom_path).
```



We can add site names with....More joins! Our swva_sites data has the names of the sites in human-friendly language. The column in the downloaded data and in the swva_sites data is called “site_no” so we just give that to the “by” argument. Perform a left join to add the names of the sites to the data.

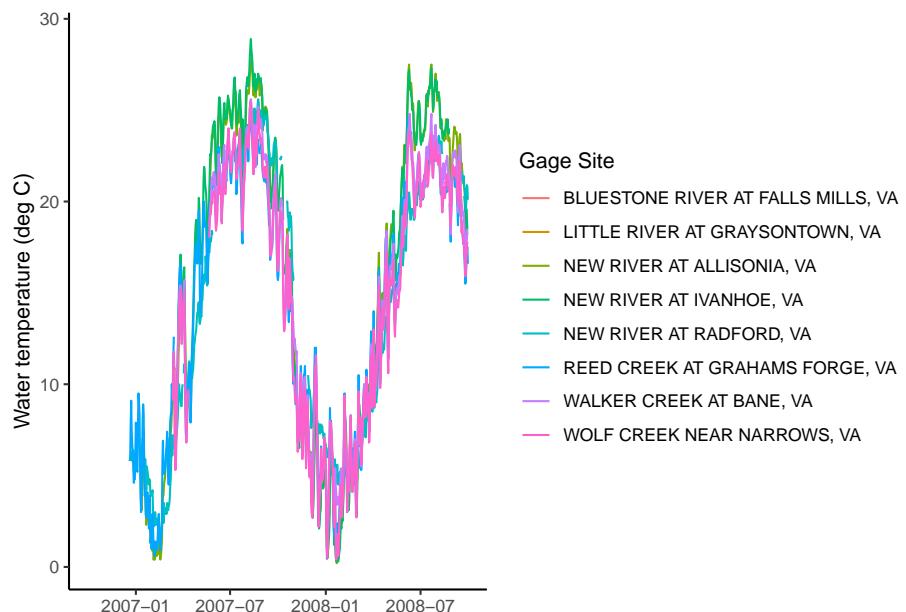
Then use select to remove some of the unnecessary columns.

Then make the plot and then snazz it up with labels and a non-junky theme.

```
swva_dat_clean <- left_join(swva_dat, swva_sites, by = "site_no") %>%
  select(station_nm, site_no, Date, Flow, Wtemp, dec_lat_va, dec_long_va)

swva_dat_clean %>% ggplot(aes(x = Date, y = Wtemp, color = station_nm)) +
  geom_line()+
  ylab("Water temperature (deg C)")+
  xlab(element_blank())+
  labs(color = "Gage Site")+
  theme_classic()

## Warning: Removed 2218 row(s) containing missing values (geom_path).
```



7.7 Pivoting: wide and long data

Okay, so with the data above: what would you do if you wanted to subtract the discharge or temperature of one gage from another on the same river: to compute rate of change between the two sites, for instance.

You could split them into two objects, then join based on date?

Or...now hear me out... you could PIVOT them.

A two-dimensional object can be either long or wide. Each has it's advantages.

LONG

Each observation has it's own row. In the first image below, the table on the right is long because each measurement has it's own row. It's value and name are identified by a other columns, and the values in that column repeat a lot.

WIDE

Observations of different things have their own columns. In the first image below, notice in the left hand table there are “Flow”, “Temp”, and “NO3” columns rather than an identifier in a separate column like in the table on the right.

Why?

Long and wide data are more efficient for different things. Think about plotting a data set with 10 stream gages. If they are in a long format, you can just add color = Gage to your ggplot aes(). If they are in a wide format, meaning each gage has it's own column, you'd have to write a new geom for EACH gage, because they're all in separate columns.

Now imagine you want to do some math to create new data: let's say cases NO3 multiplied by Flow.... How would you even do that using the data on the right? With the wide data on the left it is simply mutate(NO3 * Flow).

Finally, which table is easier to read in TABLE format (not a plot) in each of the two images below? Wide data is much more fitting for tables.

dplyr, part of the tidyverse, has functions to convert data between wide and long data. I have to look up the syntax every single time I use them. But they are VERY useful.

7.8 Pivot Examples

Back to our original question: I want to subtract the flow at Ivanhoe from the flow at Radford on the new river to see how much flow increases between the two sites through time.

PIVOT LONGER

```
MyDataLong <- pivot_longer(MyData, cols = c(Flow, Temp, NO3))
```

MyData

Site	Flow	Temp	NO3
Stream1	12	20	15
Stream2	30	18	10
Stream3	50	22	20



MyDataLong

Site	Value	Name
Stream1	12	Flow
Stream1	20	Temp
Stream1	15	NO3
Stream2	30	Flow
Stream2	18	Temp
Stream2	10	NO3
Stream3	50	Flow
Stream3	22	Temp
Stream3	20	NO3

Figure 7.7: Pivoting to a longer format

PIVOT WIDER

```
MyDataWide <- pivot_wider(MyData, names_from = Name, values_from = Value)
```

MyData

Site	Value	Name
Stream1	12	Flow
Stream1	20	Temp
Stream1	15	NO3
Stream2	30	Flow
Stream2	18	Temp
Stream2	10	NO3
Stream3	50	Flow
Stream3	22	Temp
Stream3	20	NO3



MyDataWide

Site	Flow	Temp	NO3
Stream1	12	20	15
Stream2	30	18	10
Stream3	50	22	20

Figure 7.8: Pivoting to a wider format

To do this I am going to use pivot_wider() to give Ivanhoe and Radford discharges their own column.

First, we will use select to trim the data to just what we need, then call pivot_wider telling it which data to use for the new column names (names_from = station_nm) and what values we want to pivot into the data under those columns (values_from = Flow).

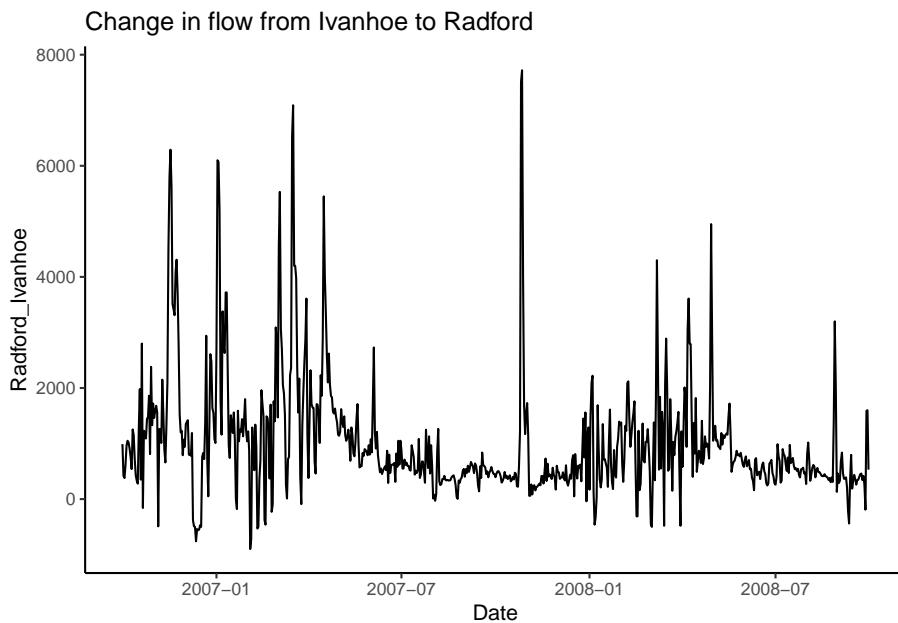
Then, subtract the two and make a plot!

```
#Pivot so we can compute diffs between one river and others

swva_wide <- swva_dat_clean %>% select(station_nm, Flow, Date) %>%
  pivot_wider(names_from = station_nm, values_from = Flow)

swva_wide <- swva_wide %>%
  mutate(Radford_Ivanhoe =
    `NEW RIVER AT RADFORD, VA` - `NEW RIVER AT IVANHOE, VA`)

ggplot(swva_wide, aes(x = Date, y = Radford_Ivanhoe))+
  geom_line()+
  ggtitle("Change in flow from Ivanhoe to Radford")+
  theme_classic()
```



To further illustrate how to move between long and wide data and when to use them, let's grab some water quality data. This process will also review some of the other concepts from this topic.

In the chunk below we will look to see what sites have data for nitrate and chloride in our swva bounding box from above. We will then filter them to just stream sites (leave out groundwater and springs). And finally we will download the nitrate and chloride data for those sites.

Note that the `readWQPqw()` function requires the site number to have a leading “USGS-” so we will add that using the `paste0()` function in a new column using a `mutate`. You could also use `paste()` with `sep = ''`. We will also use the resulting column when we join site information with the downloaded data in the next step.

```
#Nitrate as nitrate and chloride
params <- c("00940", "71851")

#what sites in our bounding box have chloride and nitrate
swva_chem_sites <- whatNWISsites(bBox = swva,
                                    parameterCd = params)

#filter to just stream water and
#prep to send to readWQPqw by adding USGS- to site_no
swva_chem_sites <- filter(swva_chem_sites, site_tp_cd == "ST") %>%
  mutate(site_no_USGS = paste0("USGS-", site_no))

wqdat <- readWQPqw(siteNumber = swva_chem_sites$site_no_USGS,
                     parameterCd = params)
```

Now, let’s clean things up a bit.

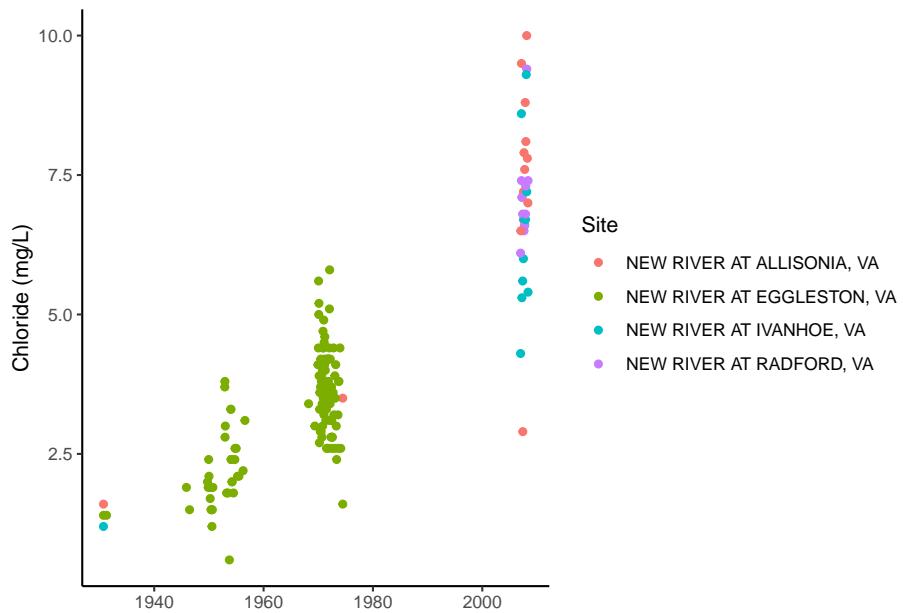
Join the parameter names from `parameterCdFile` and then join the site names from `swva_chem_site`. Then select just the columns we want, and finally filter the remaining data to just look at sites from the New River.

To illustrate the functionality of the data in this format, plot Chloride for each site, and then plot Chloride AND Nitrate, using the parameter name in `facet_wrap`.

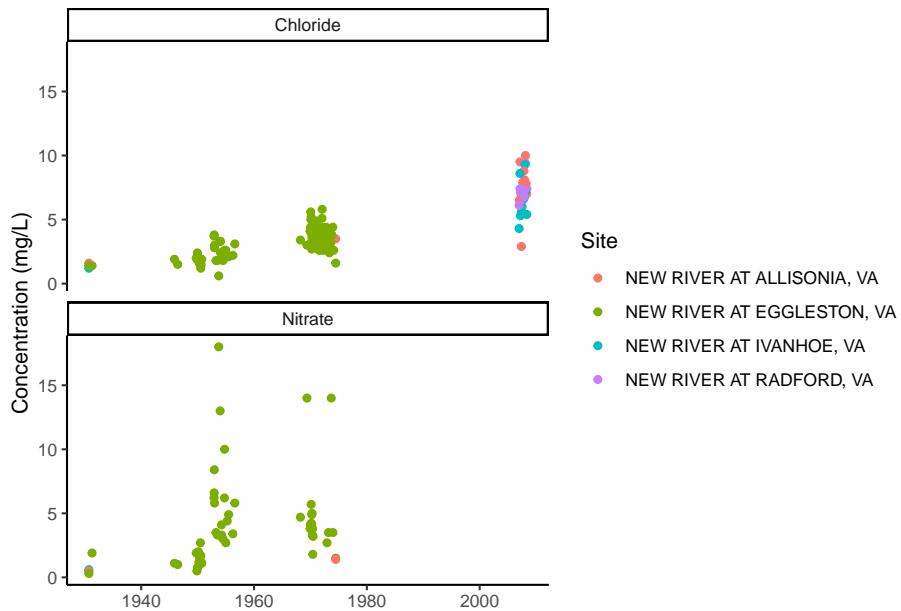
```
wqdat_clean <- wqdat %>%
  left_join(swva_chem_sites,
            by = c("MonitoringLocationIdentifier" = "site_no_USGS")) %>%
  select(MonitoringLocationIdentifier, ActivityStartDate,
         CharacteristicName, ResultMeasure.MeasureUnitCode,
         station_nm, ResultMeasureValue) %>%
  filter(str_detect(station_nm, "NEW RIVER"))

wqdat_clean %>% filter(CharacteristicName == "Chloride") %>%
  ggplot(aes(x = ActivityStartDate, y = ResultMeasureValue, color = station_nm)) +
  geom_point()
```

```
ylab("Chloride (mg/L)")+
  xlab(element_blank())+
  labs(color = "Site")+
  theme_classic()
```



```
wqdat_clean %>%
  ggplot(aes(x = ActivityStartDate, y = ResultMeasureValue, color = station_nm)) +
  geom_point() +
  facet_wrap(facets = "CharacteristicName", nrow = 2) +
  ylab("Concentration (mg/L)") +
  xlab(element_blank()) +
  labs(color = "Site") +
  theme_classic()
```



Now let's say we want to calculate something with chloride and nitrate. We need to make the data wide so we have a nitrate column and a chloride column. Do that below. What goes into `values_from`? what goes into `names_from`?

Next, plot Chloride and Nitrate added together (Chloride + Nitrate). Could you do this with the data in the previous format?

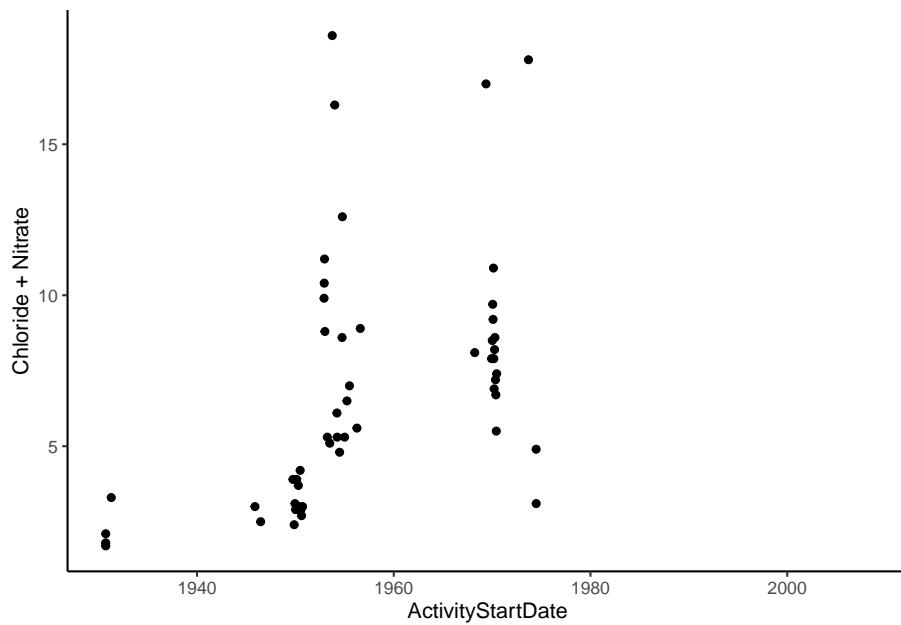
Finally, use `pivot_longer` to transform the data back into a long format. Often you'll get data in a wide format and need to convert it to long, and we haven't tried that yet. The only argument you'll need to pass to `pivot_longer()` in this case is to tell it what columns to turn into the new DATA column (using the `cols =`) parameter.

```
#make wqdat_clean wide

wqdat_wide <- wqdat_clean %>% select(-ResultMeasure.MeasureUnitCode) %>%
  pivot_wider(values_from = ResultMeasureValue, names_from = CharacteristicName)

ggplot(wqdat_wide, aes(x = ActivityStartDate, y = Chloride + Nitrate)) +
  geom_point()+
  theme_classic()

## Warning: Removed 104 rows containing missing values (geom_point).
```



```
wqlonger <- wqdat_wide %>%
  pivot_longer(cols = c("Chloride", "Nitrate"))
```


Chapter 8

ACTIVITY Joins Pivots dataRetrieval

Get this document at the template repository on github: https://github.com/VT-Hydroinformatics/7-Activity-Joins-Pivots_dataRetrieval

8.1 Load the tidyverse, dataRetrieval, and patchwork packages.

```
library(tidyverse)
library(dataRetrieval)
library(patchwork)
```

8.2 Problem 1

Using `readNWISqw()`, read all the chloride (00940) data for the New River at Radford (03171000). Use the `head()` function to print the beginning of the output from `readNWISqw`.

8.3 Problem 2

Using the `readNWISdv` (daily values) function, download discharge (00060), temperature (00003), and specific conductivity (00095) for the New River at

Radford from 2007 to 2009 (regular year). Use `renameNWIScolumns()` to rename the output of the download. Use `head()` to show the beginning of the results of your download.

8.4 Problem 3

Do a left join on `newphys` and `newriver` to add the chloride data to the daily discharge, temp, and conductivity data. hint: you will join on the date. Preview your data below the chunk using `head()`.

8.5 Problem 4

Create a line plot of Date (x) and Flow (y). Create a scatter plot of Date (x) and chloride concentration (y). Put the graphs on top of each other using the `patchwork` library.

8.6 Problem 5

Create a scatter plot of Specific Conductivity (y) and Chloride (x). Challenge: what could you do to get rid of the warning this plot generates about NAs.

8.7 Problem 6

Read in the GG chem subset data and plot `Mg_E1` (x) vs `Ca_E1` (y) as points.

8.8 Problem 7

We want to look at concentrations of each element in the #6 dataset along the stream (Distance), which is difficult in the current format. Pivot the data into a long format, the data from Ca, Mg, and Na _E1 columns should be pivoted. Make line plots of each element where y is the concentration and x is distance. Use `facet_wrap()` to create a separate plot for each element and use the “scales” argument of `facet_wrap` to allow each plot to have different y limits.

Chapter 9

ACTIVITY Summative 1

Get this document at the template repository on github: https://github.com/VT-Hydroinformatics/8-Test_1

9.0.1 Instructions

Please read carefully!

Write your code in the provided code chunks and answer any questions by typing outside the chunk.

Comment your code to let me know what you are trying to do, in case something doesn't work.

Turn in a knitted rmd (html or pdf). If you can't get your document to knit when you go to turn it in, just comment out the lines of code that are causing the knit to fail, knit the document, and submit.

9.1 Problem 1

Load the tidyverse, lubridate, and dataRetrieval packages.

9.2 Problem 2

Read in the McDonald Hollow dataset in the project folder.

What are the data types of the first three columns?

How long is the data (number of rows)?

What is the name of the last column?

9.3 Problem 3

Plot the stage of the stream (Stage_m_pt) on the y axis as a line and the date on the x. These stage data are in meters, convert them to centimeters for the plot.

For all plots in this test, label axes properly and use a theme other than the default.

9.4 Problem 4

We want to look at the big event that happens from November 11, 2020 to November 27, 2020. Filter the dataset down to this time frame and save it separately. Make a plot with the same setup as in #3 with these newly saved data.

9.5 Problem 5

For this storm, we are curious about how conductivity changes with the stream level. To do this, make a scatter plot that shows Stage on the x axis and specific conductivity (SpC_mScm) on the y. (units: mScm) Color the points on the plot using the datetime column. Use the plot to describe how specific conductivity changes with stream stage throughout the storm. (not functionally, just how the values change)

9.6 Problem 6

Continuing to look at the storm, as an exploratory data analysis, we want to create a plot that shows all the parameters measured. To do this, pivot the STORM EVENT data so there is a column that has the values for all the parameters measured as individual rows, along with another column that identifies the type of measurement. Then use facet_wrap with the “name” column (or whatever you call it) as the facet. Be sure to set the parameters of facet_wrap such that the y axes are all allowed to be different ranges.

EX:

Date Value Name

10/1/20 12 Stage

10/1/20 6 Temp

....

9.7 Problem 7

We want to create a table that clearly shows the differences in water temperature for the three months at the two locations (flow and pool) in the FULL data set (not the storm subset). To do this: Create a new column in the full dataset called “month” and set it equal to the month of the datetime column using the month() function. Then group your dataset by month and summarize temperature at each location by mean. Save these results to a new object and output it so it appears below your chunk when you knit. Be sure the object has descriptive column names.

You can do this all in one statement using pipes.

9.8 Problem 8

Plot the distribution of the flow temperature and show as vertical lines on the plot the mean, median, and IQR. Be careful about how you show IQR. Look at the definition and then think about how you would put it on the plot. Describe in the text above the chunk what color is what statistic in the plot. Using the shape of the distribution and the measures you plotted, explain why you think the distribution is normal or not. What statistical test could you perform to see if it is normal?

9.9 Problem 9

In this question we will get and format data for three USGS gages.

Gages: 03177710, 03173000, 03177480

Discharge in cubic feet per second (cfs) code: 00060

- a. Read and save the gage information for the three gages using readNWISsite().
- b. Use the readNWISdv() function to read and save the daily discharge values for the following three gages for the 2020 water year (10-01-2019 to 9-30-2020). And then use the renameNWIScolumns() function to make the names human-friendly.
- c. Join the gage site information from (a) to the data from (b) so you can reference the gages by their names.

9.10 Problem 10

Using the data from #9, Plot flow on the y axis and date on the x axis, showing the data as a line, and coloring by gage name.

Chapter 10

Flow Duration Curves

Get this document and a version with empty code chunks at the template repository on github: <https://github.com/VT-Hydroinformatics/9-Flow-Duration-Curves>

Alright team. So far we have learned to wrangle data, make plots, and look at data distributions. Now it is time to put all that knowledge to use.

We are on our way to doing analyses of extreme discharge events: low flow statistics and floods. But in order to do that, we need to understand a common way to look at data distributions in hydrology: the flow duration curve. As you'll see below, this is basically just a different way of looking at a pdf, and it can take some getting used to. But it is also a very useful tool!

As always let's load the packages we will use: tidyverse, dataRetrieval, lubridate, and patchwork. Patchwork will help us make a multi-panel graph in the last part of the exercise.

We will also use theme_set() in this chunk so we don't have to change the ggplot theme every time we make a plot.

```
library(tidyverse)
library(dataRetrieval)
library(lubridate)
library(patchwork)

#set plot theme for the document so we
#don't have to do it in every plot
theme_set(theme_classic())
```

10.1 Get data

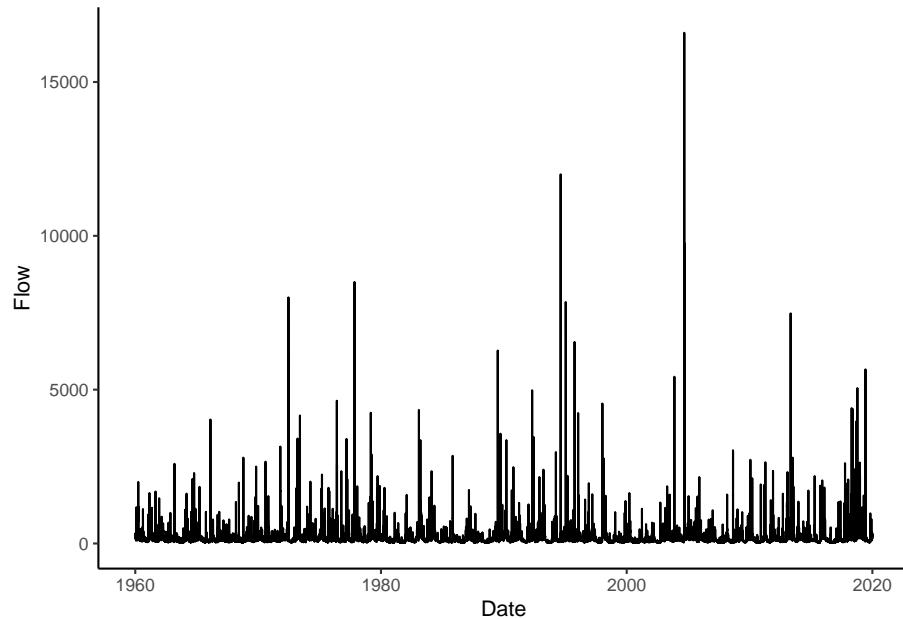
To start, let's grab the USGS discharge data for the gage in Linville NC from 1960 to 2020.

We will download the data using USGS dataRetrieval and look at a line plot.

```
siteno <- "02138500" #Linville NC
startDate <- "1960-01-01"
endDate <- "2020-01-01"
parameter <- "00060"

Qdat <- readNWISdv(siteno, parameter, startDate, endDate) %>%
  renameNWISColumns()

#Look at the data
Qdat %>% ggplot(aes(x = Date, y = Flow))+
  geom_line()
```

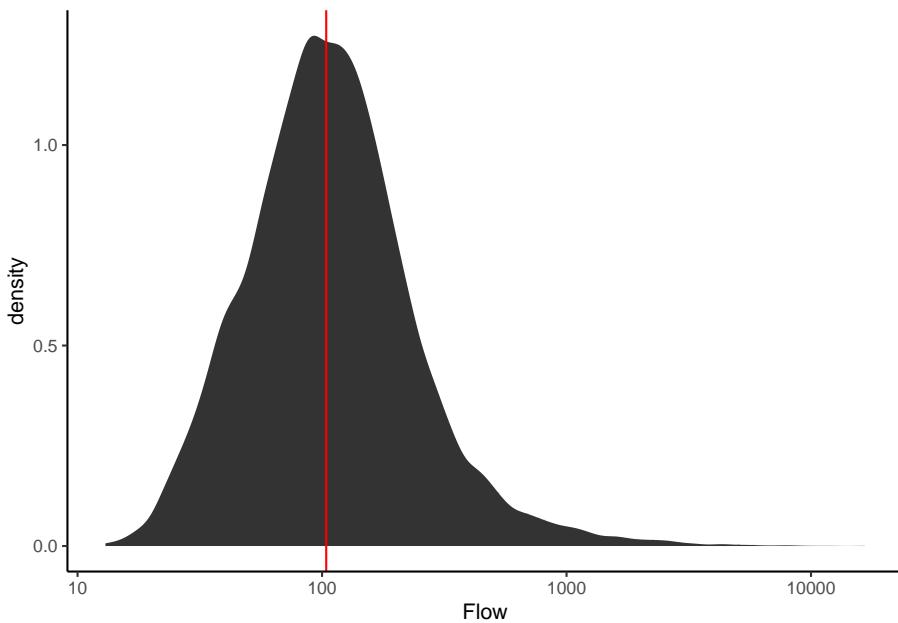


10.2 Review: describe the distribution

Make a plot to view the distribution of the discharge data.

- What is the median flow value?
- What does this tell us about flow at that river?
- How often is the river at or below that value?
- Could you pick that number off the plot?
- What about the flow the river is at or above only 5% of the time?

```
Qdat %>% ggplot(aes(Flow))+
  stat_density()+
  scale_x_log10()+
  geom_vline(xintercept = median(Qdat$Flow), color = "red")
```



10.3 ECDFs

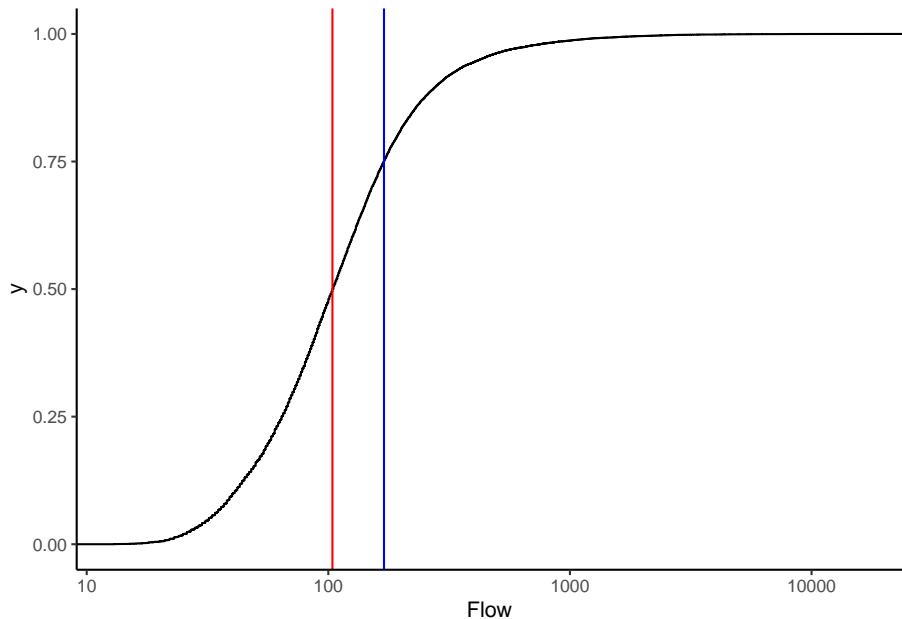
Let's look at an Empirical Cumulative Density Function (ECDF) of the data.

Look at this carefully, what does it show? How is it different from the pdf of the data?

Plot the median again. Without the line on the plot, how would you tell where the median is?

Given your answer to the question above, can you determine the flow the river is at or above only 25% of the time? Think carefully about what the y axis of the ECDF means.

```
Qdat %>% ggplot(aes(Flow))+
  stat_ecdf()+
  scale_x_log10()+
  geom_vline(xintercept = median(Qdat$Flow), color = "red")+
  geom_vline(xintercept = quantile(Qdat$Flow) [4], color = "blue")
```



10.4 Calculate flow exceedence probabilities

In hydrology, it is common to look at a similar representation of flow distributions, but with flow on the Y axis and “% time flow is equaled or exceeded” on the X axis. There are a number of ways we could make this plot: for example we could transform the axes of the plot above or we could use the function that results from the ECDF function in R to calculate exceedence probabilities at flow throughout our range of flows. But for our purposes, we are just going to calculate it manually.

We are going to calculate our own exceedence probabilities because knowing how to do this will hopefully help us understand what a flow duration curve is AND we will need to do similar things in our high and low flow analyses.

The formula for exceedence probability (P) is below. What do we need to calculate this?

Exceedence probability (P), Probability a flow is equaled or exceeded

10.5. PLOT A FLOW DURATION CURVE USING THE PROBABILITIES93

$$P = 100 * [M/(n + 1)]$$

M = Ranked position of the flow n = total number of observations in data record

Here's a description of what we will do:

Pass our Qdat data to mutate and create a new column that is equal to the ranks of the discharge column.

Then pass that result to mutate again and create another column equal exceedence probability (P) * 100, which will give us %.

```
#Flow is negative in rank() to make
#high flows ranked low (#1)
Qdat <- Qdat %>%
  mutate(rank = rank(-Flow)) %>%
  mutate(P = 100 * (rank / (length(Flow) + 1)))
```

10.5 Plot a Flow Duration Curve using the probabilities

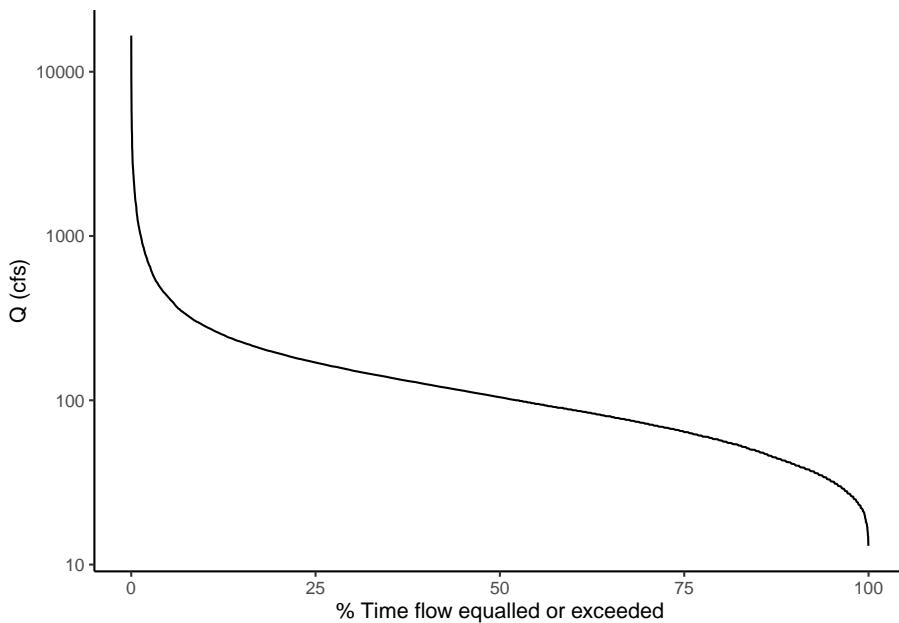
Now construct the following plot: A line with P on the x axis and flow on the y axis. Name the x axis "% Time flow equaled or exceeded" and log the y axis.

That's a flow duration curve!

Questions about the flow duration curve:

- How often is a flow of 100 cfs exceeded at this gage?
- Is flow more variable for flows exceeded 0-25% or of the time or 75-100% * of the time?
- How can you tell?
- These data are daily observations. Given that, what is a more accurate name for the x axis?
- What would the X axis be called if we were using maximum yearly data?

```
Qdat %>% ggplot(aes(x = P, y = Flow)) +
  geom_line() +
  scale_y_log10() +
  xlab("% Time flow equalled or exceeded") +
  ylab("Q (cfs)")
```

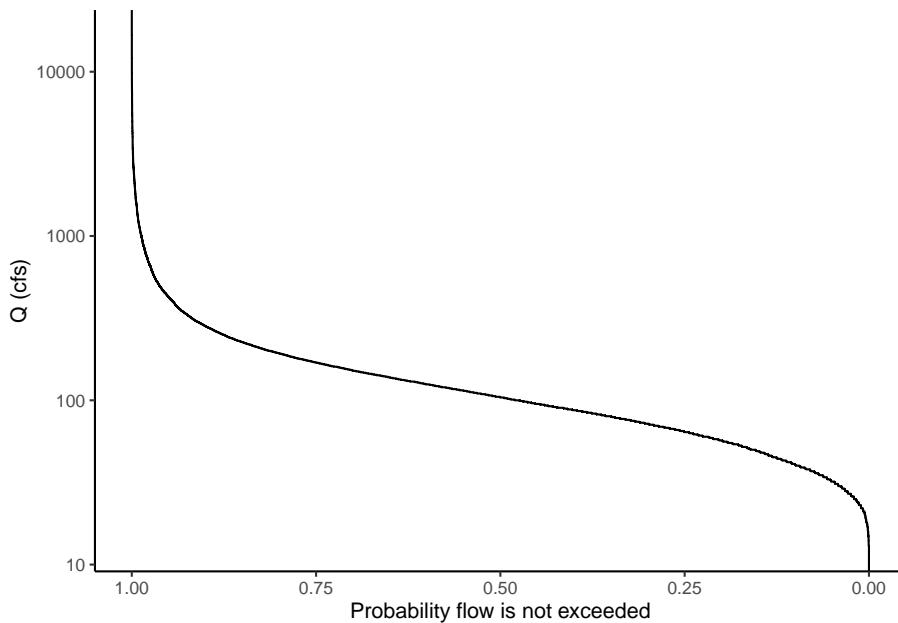


10.6 Make an almost FDC with stat_ecdf

Below is an example of making a very similar plot with the `stat_ecdf()` geometry in ggplot. Notice how similar the result is to the one we calculated manually.

To make the plot similar, we will reverse the y axis of the ecdf plot with `scale_y_reverse` and flip the axes (change the x to y and the y to x) with `coord_flip()`

```
Qdat %>% ggplot(aes(Flow))+
  stat_ecdf()+
  scale_x_log10()+
  scale_y_reverse()+
  coord_flip()+
  xlab("Q (cfs)")+
  ylab("Probability flow is not exceeded")
```



10.7 Example use of an FDC

Let's explore one potential use of flow duration curves: examining the differences between two sets of flow data.

From the line plot of the discharge, it looked like the flow regime may have shifted a bit in the data between the early years and newer data. Let's use flow duration curves to examine potential differences. We can come up with groups and then use `group_by` to run the analysis by groups instead of the whole dataset.

We are introducing a new function here called `case_when()`. This allows you to assign values to a new column based on values in another column. In our case, we are going to name different time periods in our data.

We will then group the data by these periods and calculate exceedence probabilities for each. The procedure works the same, except we add a `group_by` statement to group by our time period column before we create the rank and P columns. Then, when we plot, we can just tell ggplot to create different colored lines based on the time period names and it will plot a separate flow duration curve for each. Tidyverse FOR THE WIN!

Describe the differences in flow regime you see between the three periods of 1960-1980, 1980-2000, and 2000-2020.

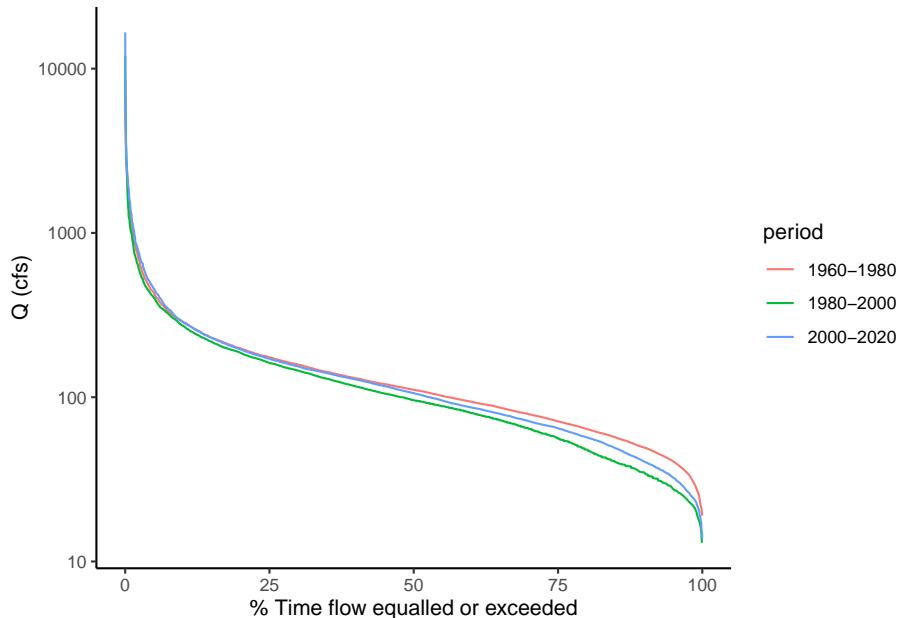
```

Qdat <- Qdat %>%
  mutate(year = year(Date)) %>%
  mutate(period = case_when( year <= 1980 ~ "1960-1980",
                             year > 1980 & year <= 2000 ~ "1980-2000",
                             year > 2000 ~ "2000-2020"))

Qdat <- Qdat %>%
  group_by(period) %>%
  mutate(rank = rank(-Flow)) %>%
  mutate(P = 100 * (rank / (length(Flow) + 1)))

Qdat %>% ggplot(aes(x = P, y = Flow, color = period))+
  geom_line()+
  scale_y_log10()+
  xlab("% Time flow equalled or exceeded")+
  ylab("Q (cfs)")

```



10.8 Compare to a boxplot of the same data

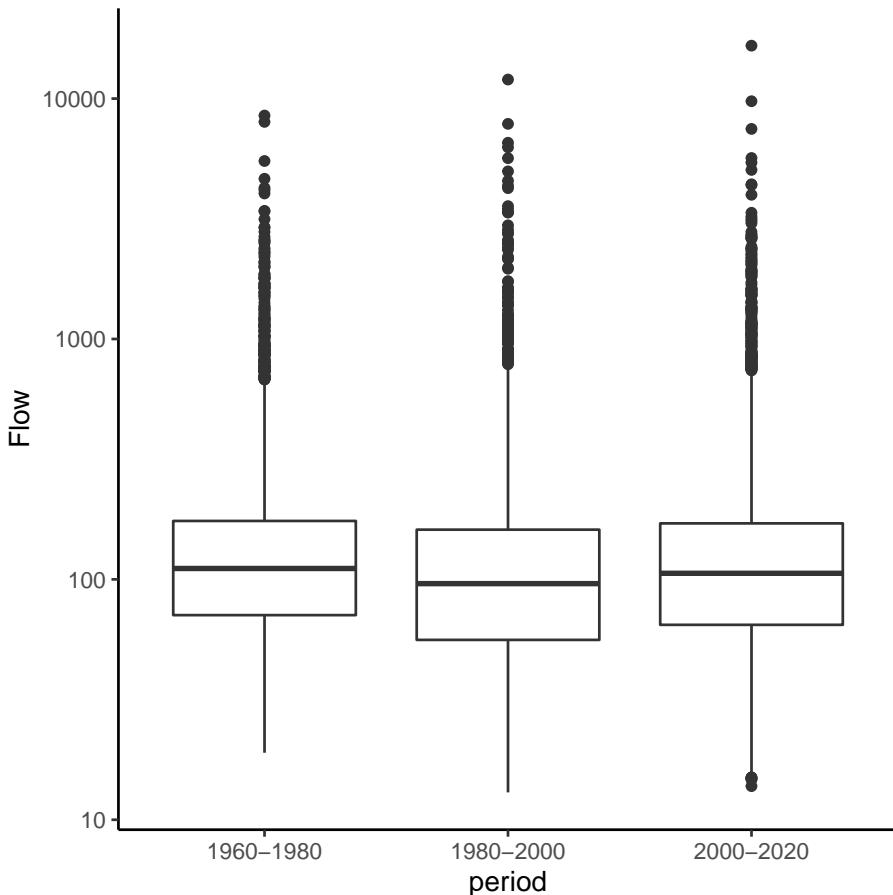
We are really just looking at the data distribution here. Remember another good way to compare distributions is a boxplot. Let's create a boxplot showing flows from these time periods. (we will also mess with the dimensions of the

10.9. CHALLENGE: EXAMINING FLOW REGIME CHANGE AT THE GRAND CANYON 97

plot so the boxes aren't so wide using `fig.width` and `fig.height` in the “` header above the code chunk)

What are the advantages/disadvantages of the flow duration curves vs. boxplots?

```
Qdat %>% ggplot(aes(x = period, y = Flow)) +  
  geom_boxplot() +  
  scale_y_log10()
```



10.9 Challenge: Examining flow regime change at the Grand Canyon

The USGS Gage “Colorado River at Yuma, AZ” is below the Hoover dam. The Hoover Dam closed in 1936, changing the flow of the Colorado River below. Load

average daily discharge data from 10-01-1905 to 10-01-1965 from the Yuma gage. Use a line plot of discharge and flow duration curves to examine the differences in discharge for the periods: 1905 - 1936, 1937 - 1965.

How does the FDC show the differences you observed in the line plot?

```

siteid <- "09521000"
startDate <- "1905-10-01"
endDate <- "1965-10-01"
parameter <- "00060"

WS <- readNWISdv(siteid, parameter, startDate, endDate) %>%
  renameNWISColumns() %>%
  mutate(year = year(Date)) %>%
  mutate(period = case_when( year <= 1936 ~ "Pre Dam",
                             year > 1936 ~ "Post Dam")) %>%
  group_by(period) %>%
  mutate(rank = rank(-Flow)) %>%
  mutate(P = 100 * (rank / (length(Flow) + 1)))

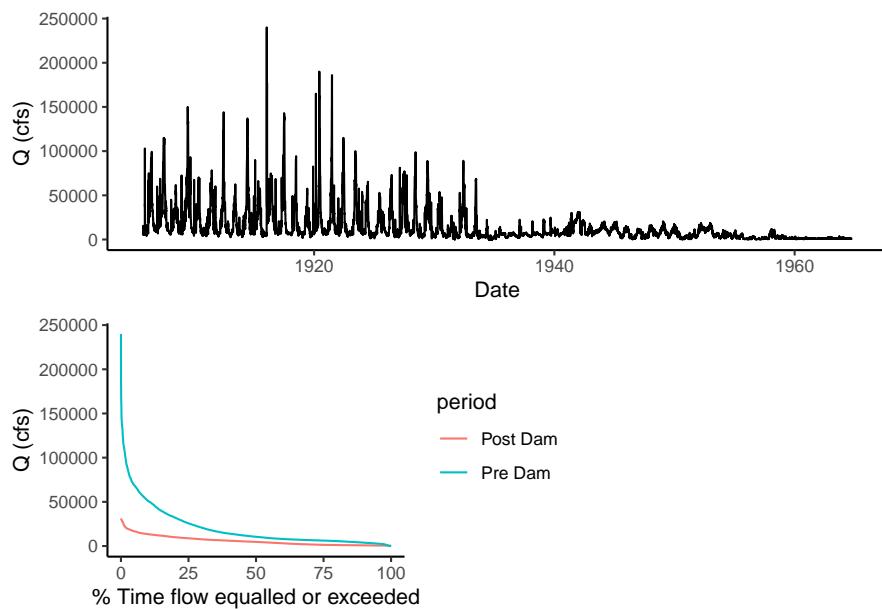
flow <- ggplot(WS, aes(Date, Flow)) +#, color = period) +
  geom_line() +
  ylab("Q (cfs)")

fdc <- WS %>% ggplot(aes(x = P, y = Flow, color = period)) +
  geom_line() +
  #scale_y_log10() +
  xlab("% Time flow equalled or exceeded") +
  ylab("Q (cfs)")

flow / (fdc + plot_spacer())

```

10.9. CHALLENGE: EXAMINING FLOW REGIME CHANGE AT THE GRAND CANYON99



That's it! Next we will apply some of these principles to look at low-flow statistics.

Chapter 11

Low Flow Analysis

Get this document and a version with empty code chunks at the template repository on github: <https://github.com/VT-Hydroinformatics/10-Low-Flow-Analysis>

Pre-activity reading:

<https://www.epa.gov/ceam/definition-and-characteristics-low-flows#1Q10>

Analysis based on:

<https://github.com/DEQdsobota/Oregon7Q10/blob/master/R/Oregon7Q10.R>

<https://nepis.epa.gov/Exe/ZyPDF.cgi?Dockey=P100BK6P.txt>

Load packages for analysis. zoo will allow us to easily perform rolling means, and moments will allow easy calculation of skewness.

```
library(zoo)
library(tidyverse)
library(dataRetrieval)
library(lubridate)
library(moments)

theme_set(theme_classic())
```

11.1 What are low flow statistics?

Low flow design flows can be specified based on hydrological or biological data. Biological methods look more at water quality standards relevant to biota. The

hydrologic method just looks at the statistical distribution of low flows over a period of time.

- Just from this simple definition, can you think of a management situation where it would make sense to use the biological method? the hydrologic method? What are the advantages to each?

We will focus on hydrologic methods. What a surprise! You will most frequently see low flow stats in the format of xQy . So for example 7Q10 or 1Q10 are common design flows. Let's look at the EPA definition of these and then break them down.

"The 1Q10 and 7Q10 are both hydrologically based design flows. The 1Q10 is the lowest 1-day average flow that occurs (on average) once every 10 years. The 7Q10 is the lowest 7-day average flow that occurs (on average) once every 10 years." -EPA <https://www.epa.gov/ceam/definition-and-characteristics-low-flows#1Q10>

So the first number, **the 7 in 7Q10** is how many days we will average flow over to calculate the statistic. Why does does this matte? Why not always use a 1 day flow record?

Then the second number is the return-interval of the flow, or the probability that a flow of that magnitude or lower will occur any given year. **The 10 in 7Q10** means there is a 10 percent chance that the associated 7-day average flow or below will occur in any given year. Another way of saying this is that a flow of that magnitude or below occurs on average once every 10 years. **However** expressing it this way can be dangerous, especially with the opposite type of extreme flows: Floods. Why do you think it could be dangerous to say a flow of this magnitude or below will happen on average once every 10 years?

So, to calculate a 7Q10 we need: * 7-day mean-daily flows * The minimum value per year of those 7-day mean-daily flows * The return intervals of those flows minimum yearly flows

Because a 7Q10 flow means * There is a 10% chance (return interval = 10) that a river will have a average weekly flow of that level or below in a given year.

11.2 Get data

Let's get started on an example. We will calculate the 7Q10 low flow statistic for the Linville NC usgs gage (02138500) using daily discharge data from 1922-1984. (parameter = 00060)

```
siteno <- "02138500"
startDate <- "1922-01-01"
endDate <- "1984-01-01"
parameter <- "00060"

Qdat <- readNWISdv(siteno, parameter, startDate, endDate) %>%
  renameNWISColumns()
```

11.3 Create the X days average flow record

Remember the 7 in 7Q10 means we are looking at the 7-day average flow. We just have daily values from the USGS gage, so we need to create this data record.

To do this we will calculate a rolling average, also called a moving-window average. This just means you grab the first 7 days, average them, then move the window of the days you are averaging forward a day, and average again... all the way through the record.

For your rolling mean you can have the window look forward, backward, or forward and backward. For example, a forward window takes the average of X number of records and places the value at the beginning. Backward places that value at the end, and both would put it in the middle. In the function we will use to do this, forward is a left align, backward is right align, and both is centered.

For example

data window = 1, 2, 3, 4, 5 (lots of values before and after this)

mean = 3

forward window/left align: 3, NA, NA, NA, NA

backward window/right align: NA, NA, NA, NA, 3

both/center align: NA, NA, 3, NA, NA

We could certainly set up some code to calculate this, but there is a nice and fast function in the zoo package for calculating rolling means.

As we write the code to do this analysis, we are going to keep in mind that we may want to calculate a different type of low flow, like a 1Q10, so we are going to store the x and y of the xQy low flow statistic as objects rather than including them several places in the code. That way we can just change them in one place and run the analysis to compute a different statistic.

```
#set x and y for xQy design flow
Xday <- 7
YrecInt <- 10

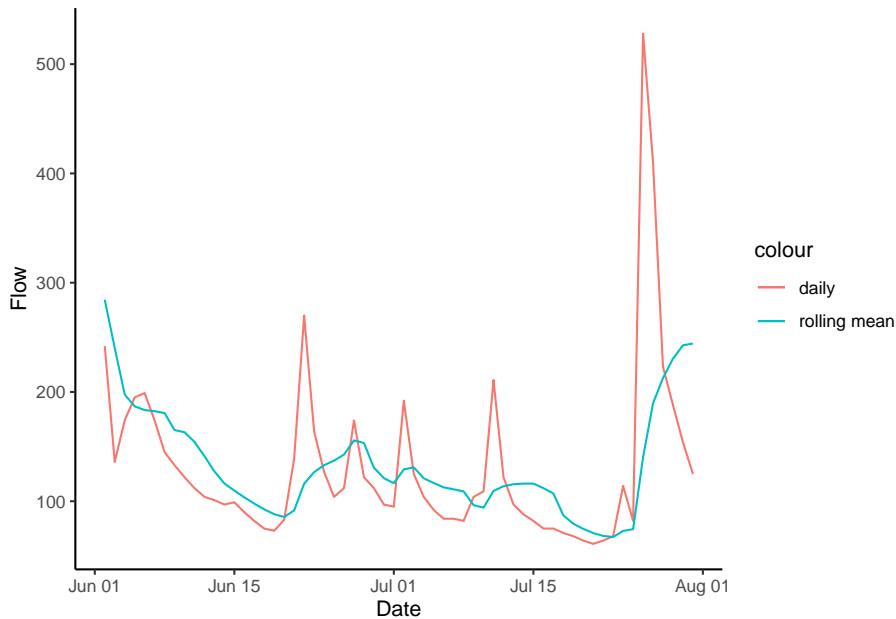
#X day rolling mean, don't fill the ends of the timeseries,
#don't ignore NAs, use a backward-looking window (right align)
Qdat <- Qdat %>% mutate(xdaymean = rollmean(Flow,
                                                 Xday,
                                                 fill = NA,
                                                 na.rm = F,
                                                 align = "right"))
```

11.4 Look at what a rolling mean does.

We just added a new column with the rolling mean, so let's plot it and see what it did to the discharge record.

Let's look at June-August 1960. You can't see too well what is going on in the full record.

```
Qdat %>%
  filter(Date > mdy("06-01-1960") & Date < mdy("08-01-1960")) %>%
  ggplot(aes(Date, Flow, color = "daily"))+
  geom_line()+
  geom_line(aes(x = Date, y = xdaymean, color = "rolling mean"))
```



11.5 Calculate yearly minimums

Okay, we have our X-day rolling mean. Now we need to calculate the probability that a given magnitude flow or below will happen in a given year. Because we are concerned with *a given year* we need the lowest flow per year.

We will calculate minimum flow per year by creating a *Year* column, grouping by that column, and using the summarize function to calculate the minimum flow per year. The code we are going to write will also drop any years that are missing too much data by dropping years missing 10% or more days.

```
#missing less than 10% of each year and 10% or fewer NAs
QyearlyMins <- Qdat %>% mutate(year = year(Date)) %>%
  group_by(year) %>%
  summarize(minQ = min(xdaymean, na.rm = T),
            lenDat = length(Flow),
            lenNAs = sum(is.na(xdaymean))) %>%
  filter(lenDat > 328 & lenNAs / lenDat < 0.1)
```

11.6 Calculate return interval

Now that we have an object that contains our yearly minimum flows, we can calculate the return interval as

$$\text{ReturnInterval} = (n + 1)/\text{rank}$$

Where n is the number of records in the data (number of years) and rank is the rank of each year's low flow (lowest flow = rank 1 and so on). We can calculate the rank with the rank() function in base R. In the rank function we will specify that in the case of a tie, the first value gets the lower rank using ties.method = "first".

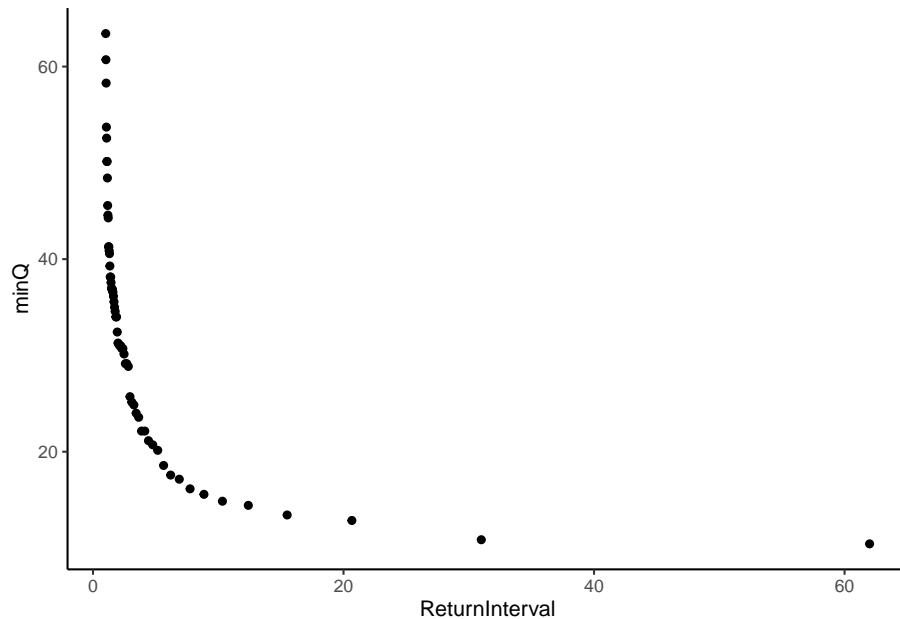
We can then transform that to an exceedence probability as

$$\text{ExceedenceProbability} = 1/\text{ReturnInterval}$$

Once we calculate the return interval and exceedence probability we will plot the return interval against the minimum discharge.

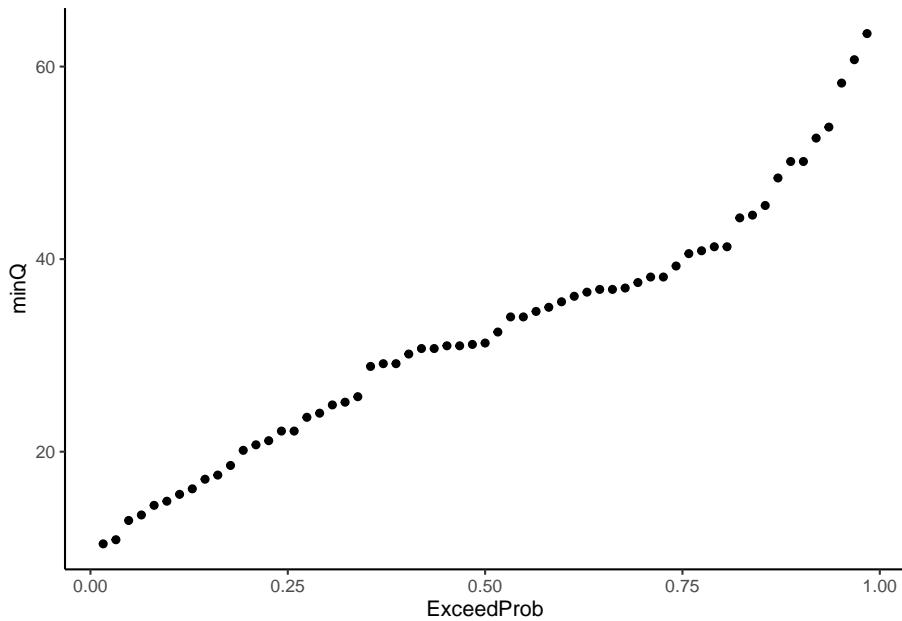
```
# add rank column and return interval column
QyearlyMins <- QyearlyMins %>%
  mutate(rank = rank(minQ, ties.method = "first")) %>%
  mutate(ReturnInterval = (length(rank) + 1)/rank) %>%
  mutate(ExceedProb = 1 / ReturnInterval)

ggplot(QyearlyMins, aes(x = ReturnInterval, y = minQ))+
  geom_point()
```



Challenge question How is this similar to a flow duration curve? Could you make a “flow duration curve” from these data? What would it tell you?

```
ggplot(QyearlyMins, aes(x = ExceedProb, y = minQ))+
  geom_point()
```



11.7 Fit to Pearson Type III distribution

Source for these calculations: https://water.usgs.gov/osw/bulletin17b/dl_flow.pdf

We now have everything we need to calculate what the 10-year return interval flow is (the 0.1 probability flow). To do this, we have to fit a distribution to our data and then use that fitted distribution to predict the value of the 10-year return interval flow.

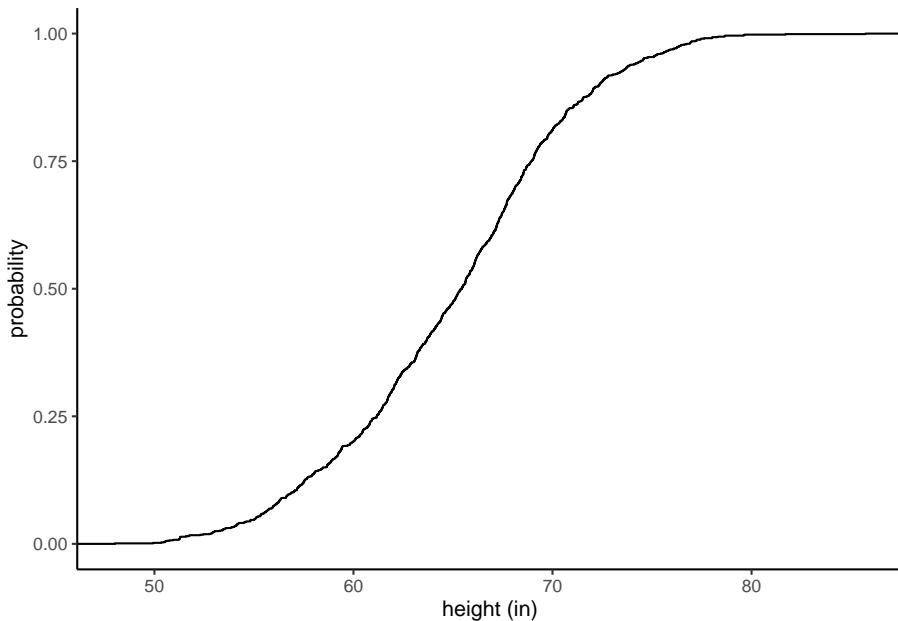
This may sound a little complex, but let's think about it this way:

- You have some data, let's say: heights of students at Virginia Tech
- You did some tests on it and know it is a normal distribution
- If you measure the mean and standard deviation of that distribution, you could create a “fitted” representation of your distribution by generating a normal distribution with the same mean and standard deviation with the `rnorm()` function.
- Now you could plot that fitted, synthetic distribution as an ECDF and read the plot to determine, say, 10% of students (0.1 probability) are at or above what height?

Assume the average height from your data was 65 inches and the standard deviation was 6 inches (this is 100% made up), let's look at it.

```
fitteddistribution <- rnorm(1000, mean = 65, sd = 6) %>%
  as_tibble()

ggplot(fitteddistribution, aes(x = value)) +
  stat_ecdf() +
  xlab("height (in)") +
  ylab("probability")
```



To get our 10 year return period (0.1 exceedence probability) we are going to do the same thing, except we know the distribution of the data isn't normal, so we have to use a different distribution.

There are a bunch of “extreme value” distributions used in these type of analyses. When we talk about floods we will use the Gumbel distribution, for example. For this type of analysis, it is common to use the Pearson Type III distribution.

When we used the normal distribution example, we let R produce the distribution that fit our data. In this case we will use an equation that describes the Person Type III distribution. To predict flow at a given recurrence interval we will need the mean of the logged discharges ($X_{\bar{}}^{}$), the frequency factor (K), the standard deviation of the log discharges (S), skewness (g), and the standard normal variate (z). We will first compute this for all of the values in our dataset to see how the fitted values fit our calculated values.

Pearson Type III: Frequency Factor method

$$Flow = 10^{(Xbar + KS)}$$

where:

Xbar = mean of the log discharge you are investigating

K = frequency factor

S = standard deviation of log discharges

Frequency Factor

$$K = (2/g) * ((1 + (g * z)) / 6 - ((g^2) / 36))^3 - 1$$

Skewness

g = skewness() from moments package

Standard normal variate

$$z = 4.91 * ((1/y)^{0.14} - (1 - (1/y))^{0.14})$$

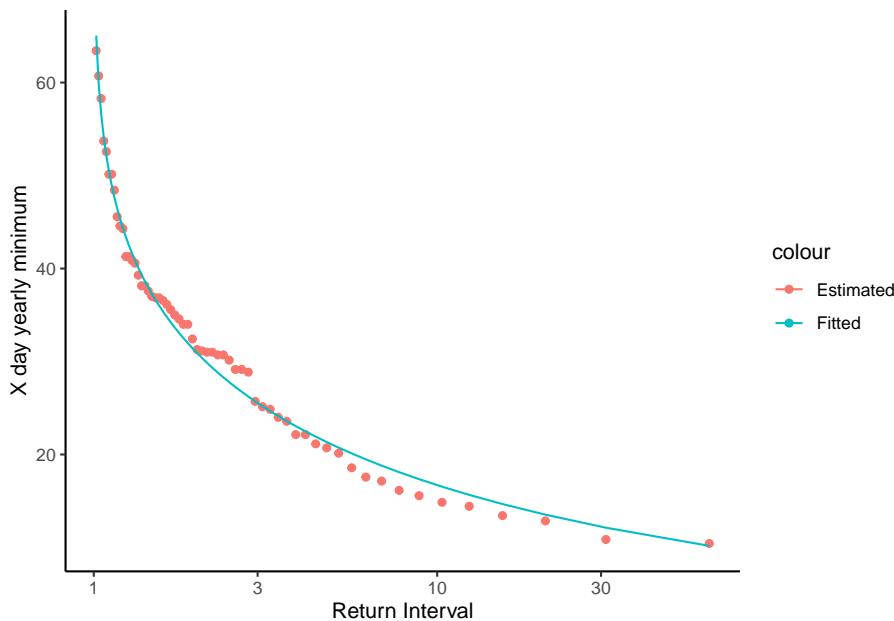
y = recurrence interval

```
#Measures of the distribution
Xbar <- mean(log10(QyearlyMins$minQ))
S    <- sd(log10(QyearlyMins$minQ))
g    <- skewness(log10(QyearlyMins$minQ))

#calculate z, K, to plot the fitted Pearson Type III
QyearlyMins <- QyearlyMins %>%
  mutate(z = 4.91 * ((1 / ReturnInterval) ^ 0.14 - (1 - 1 / ReturnInterval) ^ 0.14)) %>%
  mutate(K = (2 / g) * (((1 + (g * z) / 6 - (g ^ 2) / 36) ^ 3) - 1)) %>%
  mutate(Qfit = 10^(Xbar + (K * S)))
```

Let's look our results and see how they fit. Plot the return interval on the x axis and flow on the y. Plot minQ, the minimum Q data, and Qfit, the data from the the model fit.

```
QyearlyMins %>%
  ggplot(aes(x = ReturnInterval, y = minQ, color = "Estimated"))+
  geom_point()+
  geom_line(aes(x = ReturnInterval, y = Qfit, color = "Fitted"))+
  theme_classic()+
  scale_x_log10()+
  ylab("X day yearly minimum")+
  xlab("Return Interval")
```



Above we calculated z , K and the flow for each return interval in our data record to see how the distribution fit our data. We can see it fits quite well.

We can use the same calculations as we used on the entire record to calculate a specific return period of interest. In our case, the 10 year return period for the 7Q10.

We will set y equal to Y_{recInt} , which we set above. This way we can just change it at the top of the code to run whatever xQy metric we want.

```
#xQy ei: 7Q10
y = YrecInt

#Find these values based on established relationships
z <- 4.91 * ((1 / y) ^ 0.14 - (1 - 1 / y) ^ 0.14)
K <- (2 / g) * (((1 + (g * z) / 6 - (g ^ 2) / 36) ^ 3) - 1)

PearsonxQy <- 10^(Xbar + K * S)
```

So, our 7Q10 flow in cfs for this gage is....

```
#Low flow stat (7Q10 in exercise)
PearsonxQy
```

```
## [1] 16.70488
```

11.8 Distribution-free method

We won't go over this in the same detail, but the xQy flow can also be calculated using a formula that does not assume a specific distribution. The expression, and code to perform it, is below.

The expression for xQy is:

$$xQy = (1 - e)X(m1) + eX(m2)$$

where: [] indicates the value is truncated

$X(m)$ = the m-th lowest annual low flow of record

$$m1 = [(n + 1)/y]$$

$$m2 = [(n + l)/y] + 1$$

$[z]$ = the largest integer less than or equal to z

$$e = (n + l)/y - [(n + l)/y]$$

This method is only appropriate when the desired return period is less than n/5 years

```

x <- Xday
y <- YrecInt
n <- length(QyearlyMins$minQ)

m1 <- trunc((n + 1)/y)
m2 <- trunc(((n + 1)/y) + 1)

e <- ((n + 1)/y) - m1

Xm1 <- QyearlyMins$minQ[QyearlyMins$rank == m1]
Xm2 <- QyearlyMins$minQ[QyearlyMins$rank == m2]

DFxQy <- (1-e) * Xm1 + e * Xm2

DFxQy

```

```
## [1] 15
```

Chapter 12

Flood Frequency Analysis and Creating Functions

12.1 Template Repository

The following activity is available as a template github repository at the following link:

12.2 Intro

This methods for this chapter are adapted from the following activity: <https://serc.carleton.edu/hydromodules/steps/166250.html>

After working with Flow Duration Curves (FDCs) and performing a low flow analysis, we now understand all the concepts necessary to perform a flood frequency analysis. In this chapter we will perform a flood frequency analysis using a Gumbel extreme value distribution and then write our own function that will return the magnitude of whatever probability flood we want!

First we will load the tidyverse and dataRetrieval packages and the set the theme for our plots.

```
library(tidyverse)
library(dataRetrieval)

theme_set(theme_classic())
```

Next, download the yearly peakflow data from USGS dataRetrieval using the readNWISpeak() function. We don't have to create our own yearly values like

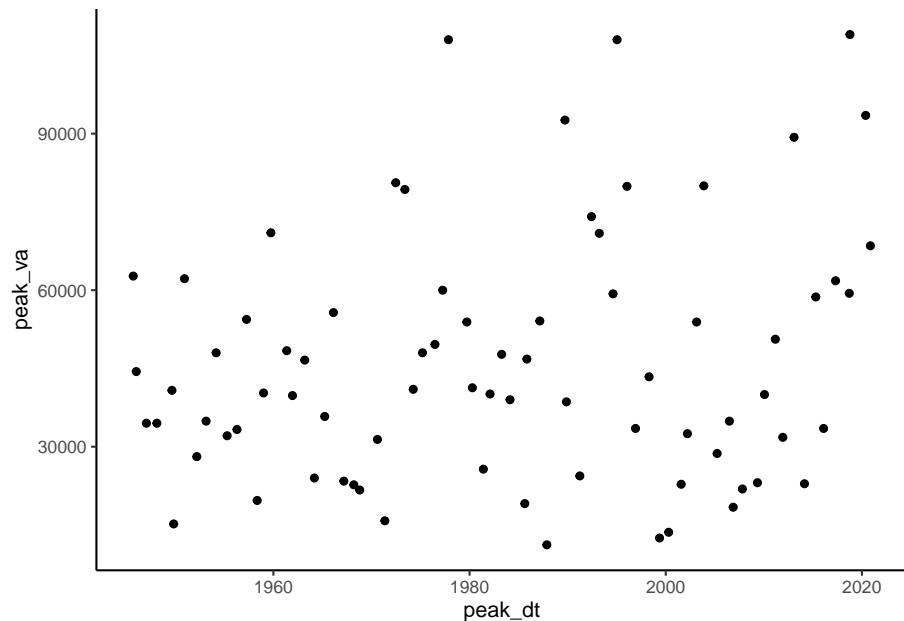
114 CHAPTER 12. FLOOD FREQUENCY ANALYSIS AND CREATING FUNCTIONS

we did in the low flow analysis. This function just returns the highest flow for each year.

Download the data for the New River at Radford starting in 1945.

Then make a plot of the peak flow for each year.

```
radford <- "03171000"  
  
peakflows <- readNWISpeak(radford, startDate = "1945-01-01")  
  
ggplot(peakflows, aes(peak_dt, peak_va)) +  
  geom_point()
```



As with the couple previous chapters, the next step is to create a column that contains the ranks of each flow in the record. Create a column that has the rank of each flow, with the highest flow ranked #1. Use `select()` to trim your dataset to just the peak data, peak values, and ranks columns.

Make the plot from the last code chunk again but color the points by rank to check that this worked. Also, look at the data through the environment tab in RStudio or using `head()` to double check.

```
#create rank column (minus flips the ranking)  
#then clean it up, pull out only peak value, date, rank  
peakflows <- peakflows %>%
```

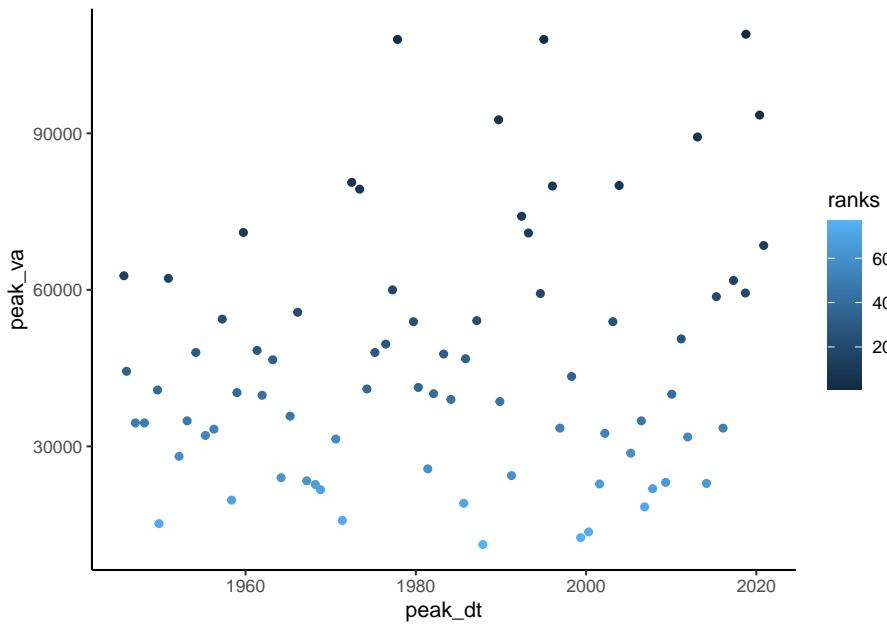
```

    mutate(ranks = rank(-peak_va)) %>%
    select(peak_dt, peak_va, ranks)

#look at it
ggplot(peakflows,
       aes(peak_dt, peak_va, color = ranks))+  

  geom_point()

```



```
head(peakflows)
```

```

##      peak_dt peak_va ranks
## 1 1945-09-18   62700 15.0
## 2 1946-01-08   44400 35.0
## 3 1947-01-20   34500 49.5
## 4 1948-02-14   34500 49.5
## 5 1949-08-29   40800 39.0
## 6 1949-11-02   15200 74.0

```

Now we need to calculate the exceedance probability and return interval for each value in our data. For flood frequency analysis, it is common to use the Gringorten plotting position formula:

qi = Exceedance probability

$$q_i = \frac{i - a}{N + 1 - 2a}$$

Figure 12.1: Plotting Position Formula

N = Number of observations in your record

i = Rank of specific observation, i = 1 is the largest, i = N is the smallest.

a = constant for estimation = 0.44

$$\text{Non-exceedence probability} = p_i = 1 - q_i$$

Return period

$$Tp = 1/(1 - p)$$

In the chunk below, create a column in your dataset and calculate each: exceedance probability, non-exceedance probability, and return period.

Then make a plot with peak flow on the Y axis and Return Period on the X.

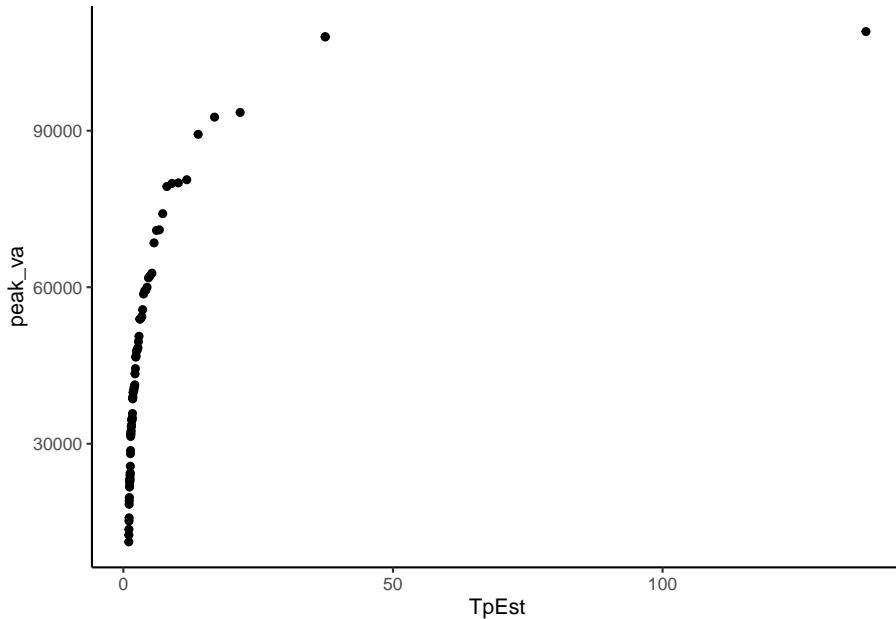
```

N <- length(peakflows$peak_dt)
a <- 0.44

#calculate exceedence/non-exceedence with gringorten and return period
peakflows <- peakflows %>%
  mutate(qi = (ranks - a) / (N + 1 - (2*a))) %>%
  mutate(pi = 1 - qi) %>%
  mutate(TpEst = 1 / (1-pi))

#Plot peak flows on y and est return period on the x
peakflows %>%
  ggplot(aes(x = TpEst, y = peak_va)) +
  geom_point()

```



Now we need to fit these data to a distribution in order to make a relationship we can use to predict the discharge of specific return intervals.

There are many distributions that can be used in this situation, but a common one for flood frequency analyses is the Gumbel extreme value distribution:

$$F_x(x) = \exp\left[-\exp\left(-\frac{x-u}{\alpha}\right)\right] = p$$

Figure 12.2: Gumbel Distribution

x is observed discharge data, u and x are parameters that shape the distribution.

We can calculate u and x in order to create a distribution that best fits our data with the following equations. Notice **x bar** is mean and **sx2** is variance. We will need to find **sx**, which is the square root of the variance, also known as the standard deviation.

In the chunk below, calculate u and alpha by first calculating xbar (mean) and sx (standard deviation) and then using them in the above equations for u and x.

```
xbar <- mean(peakflows$peak_va)
sx <- sd(peakflows$peak_va)
```

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

$$s_x^2 = \frac{1}{(n-1)} \sum_{i=1}^n (x_i - \bar{x})^2$$

$$u = \bar{x} - 0.5772\alpha$$

$$\alpha = \frac{\sqrt{6}s_x}{\pi}$$

Figure 12.3: Gumbel parameters

```
alpha <- (sqrt(6)*sx) / pi
u <- xbar - (0.5772 * alpha)
```

Now that we have the parameters that best represent our data as a Gumbel Distribution, we can use the formula to create the theoretical values for the return interval according to that distribution.

$$F_x(x) = \exp\left[-\exp\left(-\frac{x-u}{\alpha}\right)\right] = p$$

Figure 12.4: Gumbel Distribution

In the chunk below:

First calculate p theoretical with the equation above.

Then calculate Tp theoretical (the return period) as T was calculated above $T_p = 1 / (1-p)$

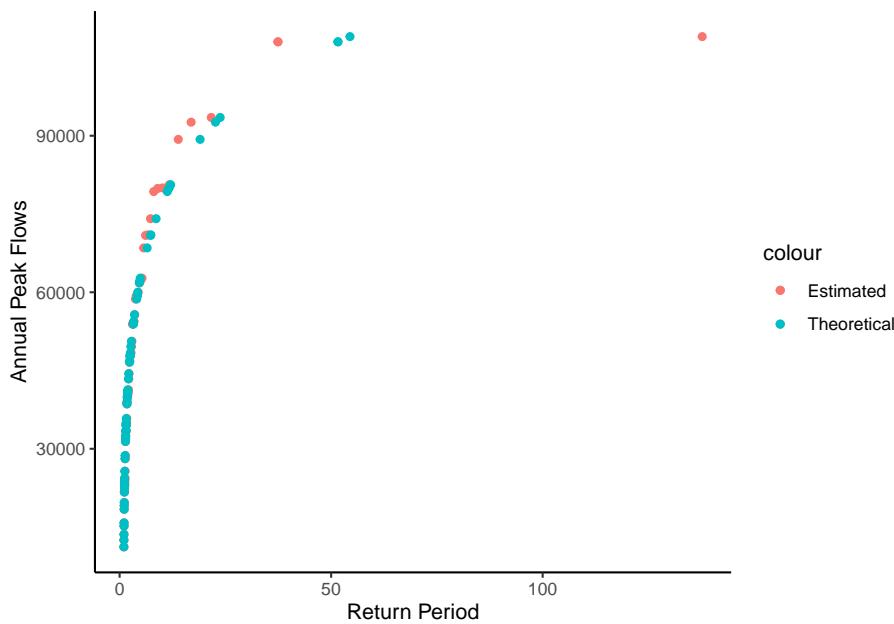
Finally create a plot of return period on the x axis and peak values on the y. Include return periods calculated from your data and those calculated from the Gumbel distribution on your plot as points of different colors.

```

peakflows <- peakflows %>%
  mutate(pTheoretical =
    exp(-exp(-((peak_va - u) / alpha)))) %>%
  mutate(TpTheoretical = (1 / (1-pTheoretical)))

peakflows %>% ggplot(aes(x = TpEst, y = peak_va, color = "Estimated")) +
  geom_point()+
  geom_point(aes(x = TpTheoretical, y = peak_va, color = "Theoretical"))+
  ylab("Annual Peak Flows")+
  xlab("Return Period")+
  theme_classic()

```



Let's look at these data a slightly different way to make it easier to see what is going on and how we can pull off flows for different return periods.

Make the same plot as above but show the theoretical values (from the distribution) as a line, with the estimated values as points, and log the x axis with limits set to 1 - 100.

With this plot you could look up the return period for any flood or the discharge level for any return period.

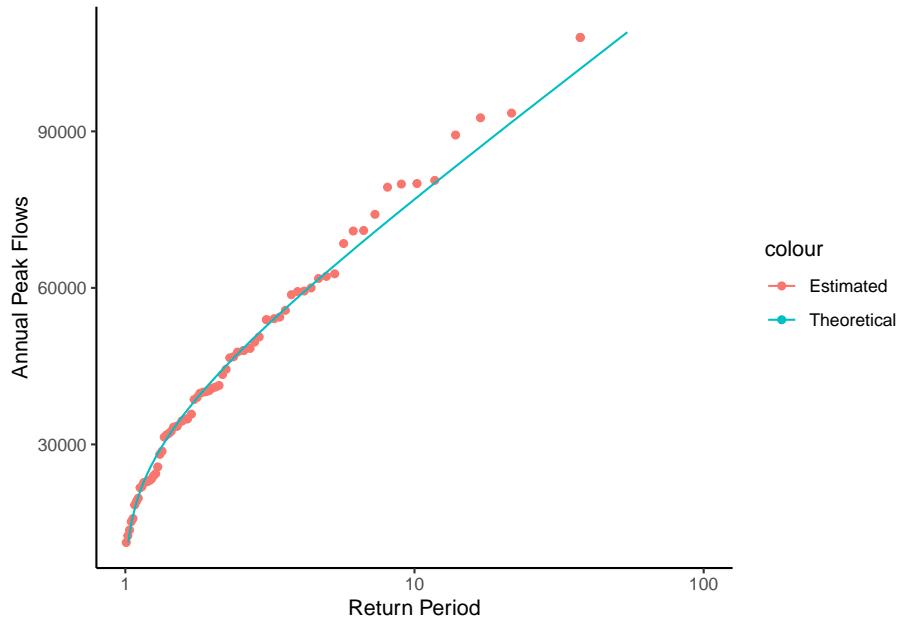
```

peakflows %>% ggplot(aes(x = TpEst, y = peak_va, color = "Estimated")) +
  geom_point()+

```

```
geom_line(aes(x = TpTheoretical, y = peak_va, color = "Theoretical"))+
  ylab("Annual Peak Flows")+
  xlab("Return Period")+
  scale_x_log10(limits = c(1,100))+
  theme_classic()
```

Warning: Removed 1 rows containing missing values (geom_point).



This plot is showing a representation of the fitted distribution by calculating the return period for each point in our dataset. But we can also use it to calculate the specific flow that corresponds to any return period by using the established relationship.

In the chunk below, calculate the magnitude of a 1 in 100 chance, or 100 year flood using the following two formulas where p = non-exceedance probability and Tp = return period. These are just the equations used to calculate return period rearranged to calculate peak flow.

$$p = 1 - (1 / Tp)$$

$$\text{peakflow} = u - (\alpha * \ln(-\ln(p)))$$

($\log()$ in r is the natural log, $\log10()$ is base 10 log)

According to this flow, what is the 1 in 100 chance flood at this location? Do you see any issues with reporting this as the 1 in 100 chance flood? What are they?

```

Tp = 100
p = 1 - (1/Tp)
peak_va = u - (alpha * log(-log(p)))
peak_va

## [1] 120275.3

```

This is a good opportunity to illustrate the usefulness of writing your own functions. When you install packages in R, you get a bunch of functions you can use. But you can also create these on your own to simplify your analyses!

You do this with the following syntax `>MyNewFunction <- function(param1, param2){ >code >}`

Whatever the last line of the “code” portion of the function spits out, get’s returned from the function. So if you said `X <- mynewfunction(param1, param2)` X would now have it in whatever your function returned. See a simple example below: a function that adds 1 to any number we pass to it.

```

add1 <- function(number){
  number + 1
}

add1(4)

## [1] 5

```

Let’s create a function that returns the return period for a flood of any magnitude for the gage we are investigating. Creating functions is a great way to streamline your workflow. You can write a function that performs an operation you need to perform a bunch of times, then just use the function rather than re-writing/copying the code.

Our function will be called “ReturnPeriod” and we will pass it the flow we want the return period for, and the u and alpha of the distribution.

We will test the function by having it calculate the return period for the 100 year flood we calculated earlier (120027). If it works, it should spit out 100.

```

ReturnPeriod <- function(flow, u, alpha){

  pTheoretical = exp(-exp(-((flow - u) / alpha)))
}

```

```
TpTheoretical = (1 / (1 - pTheoretical))

TpTheoretical
}

ReturnPeriod(120027, u, alpha)

## [1] 98.66813
```

12.3 Challenge: Create a function

Create a function that returns the 100 year flood when given a USGS gage id, startdate, and enddate for the period you want to investigate.

Chapter 13

Geospatial data in R - Vector

The following activity is available as a template github repository at the following link: https://github.com/VT-Hydroinformatics/12-Intro_Geospatial_Vector

For more: <https://datacarpentry.org/r-raster-vector-geospatial/>

Much of this is adapted from: <https://geocompr.robinlovelace.net/index.html>
Chapter 8

This demo uses data from the CAMELS dataset (full ref below) <https://ral.ucar.edu/solutions/products/camels>

A. Newman; K. Sampson; M. P. Clark; A. Bock; R. J. Viger; D. Blodgett, 2014.
A large-sample watershed-scale hydrometeorological dataset for the contiguous USA. Boulder, CO: UCAR/NCAR. <https://dx.doi.org/10.5065/D6MW2F4D>

```
library(tidyverse)
library(sf)
library(tmap)    # for static and interactive maps
library(leaflet) # for interactive maps

theme_set(theme_classic())
```

13.1 Goals

Our goals for this chapter are just to see some of the ways we can wrangle and plot vector spatial data using R. This is by no means the only way and is not an exhaustive demonstration of the packages loaded, but it'll get us started.

First, we need to define raster and vector spatial data.

Check out the images below for two examples of the same data represented as raster data or vector data.

Vector: Points, lines, polygons, boundaries are crisp regardless of scale
 Raster: Grid of same sized cells, values in cells, cell size = resolution (smaller cells, higher resolution)

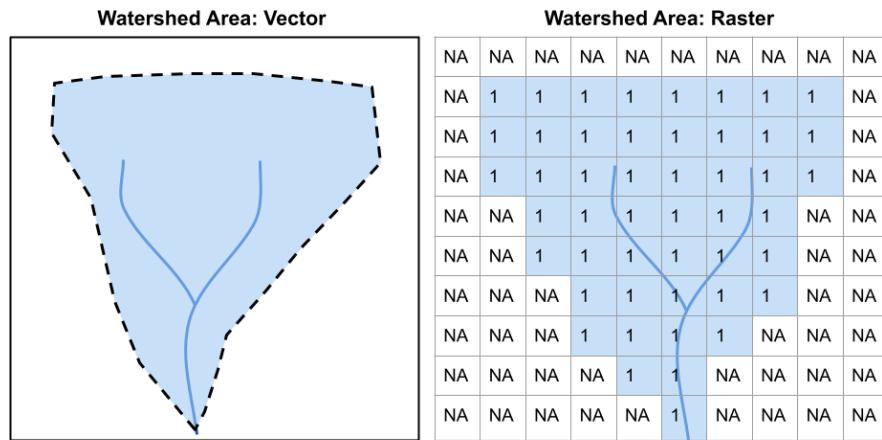


Figure 13.1: Raster vs. Vector 1

Questions from these two images:

What are the advantages/disadvantages of raster/vector for each? Which is best to show on a map for each? *For elevation, which would be better for calculating slope?

So, today we are sticking to vector data, but then we will deal primarily with raster elevation data.

13.2 Intro to tmap

We are going to make maps mostly with tmap. But there are several other options (ggplot, leaflet, etc).

Let's look at how tmap works. It uses the same syntax as ggplot: the grammar of graphics.

First we want to set tmap to static map mode. This is what we would want if we were making maps for a manuscript or slides. You can also make interactive

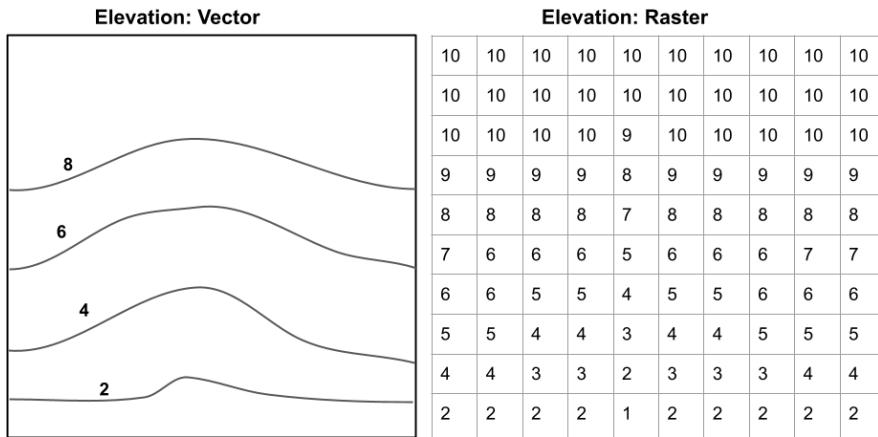


Figure 13.2: Raster vs. Vector 2

maps with tmap, which we will show later. We will also set the check.and.fix option in tmap_options to TRUE, we need to do this for the data we are using, but it isn't always necessary.

Second, we will read in our data. We'll read in the "smallerws" shapefile from the CAMELS dataset and another shapefile of the outline of US states. To read in the shapefiles we will use st_read() from the sf package.

Note that each of these shapefiles is in a separate folder and contains several files. You must have all of those files for the shapefile to work. *yes: "A shapefile" is actually several files. Silly? Maybe. The cause of much confusion when emailing someone "a shapefile"? Definitely.

Finally we will read in a csv called gauge information that has some extra info we will join to the watershed shapefile later.

Once that is all done, we will look at the watershed data to see what is available in the shapefile attribute table.

What extra information does the data have beyond a regular R object? Play around with it, can you reference columns in the table the same way you would with a regular object?

```
#make sure tmap is in static map mode
tmap_mode("plot")
```

```
## tmap mode set to plotting
```

```

#the CAMELS shapefile throws an error about having
#invalid polygons, this line allows it to plot
tmap_options(check.and.fix = TRUE)

#Read shapefiles
watersheds <- st_read("small_ws/smallerws.shp")

## Reading layer `smallerws` from data source
##   `/Users/jpgannon/My Drive (jpgannon@vt.edu)/CLASSES/SPRING Hydroinformatics/Hydro
##     using driver `ESRI Shapefile'
## Simple feature collection with 671 features and 9 fields
## Geometry type: MULTIPOLYGON
## Dimension:      XY
## Bounding box:  xmin: -124.4377 ymin: 26.96621 xmax: -67.69059 ymax: 48.99931
## Geodetic CRS:  GCS_unknown

states <- st_read("cb_2018_us_state_20m/cb_2018_us_state_20m.shp")

## Reading layer `cb_2018_us_state_20m` from data source
##   `/Users/jpgannon/My Drive (jpgannon@vt.edu)/CLASSES/SPRING Hydroinformatics/Hydro
##     using driver `ESRI Shapefile'
## Simple feature collection with 52 features and 9 fields
## Geometry type: MULTIPOLYGON
## Dimension:      XY
## Bounding box:  xmin: -179.1743 ymin: 17.91377 xmax: 179.7739 ymax: 71.35256
## Geodetic CRS:  NAD83

gageinfo <- read_csv("gauge information.csv")

## Rows: 672 Columns: 7

## -- Column specification -----
## Delimiter: ","
## chr (3): HUC_02, GAGE_ID, GAGE_NAME
## dbl (4): LAT, LONG, DRAINAGE_AREA_KM2, Elevation_m
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

#look at the watersheds shapefile data
head(watersheds)

```

```

## Simple feature collection with 6 features and 9 fields
## Geometry type: MULTIPOLYGON
## Dimension: XY
## Bounding box: xmin: -71.34071 ymin: 44.59213 xmax: -67.69059 ymax: 47.31557
## Geodetic CRS: GCS_unknown
##   hru_id ann_P lon_cen lat_cen      AREA elev_mean ave_T july_T Perimeter
## 1 1013500    0 -68.56551 47.01169 2303988087  277.4935     NA     NA  647993
## 2 1022500    0 -68.07313 44.79691  620387273  103.6042     NA     NA  312624
## 3 1030500    0 -68.14985 45.80527 3676154745  174.4339     NA     NA  662248
## 4 1031500    0 -69.58119 45.23568  766544703  304.7435     NA     NA  309614
## 5 1047000    0 -70.16213 44.98744  904956194  379.7800     NA     NA  310157
## 6 1052500    0 -71.17197 44.96168  396110305  646.0736     NA     NA  172588
##   geometry
## 1 MULTIPOLYGON (((-68.06259 4...
## 2 MULTIPOLYGON (((-67.97836 4...
## 3 MULTIPOLYGON (((-68.09162 4...
## 4 MULTIPOLYGON (((-69.31629 4...
## 5 MULTIPOLYGON (((-70.10847 4...
## 6 MULTIPOLYGON (((-71.10862 4...

```

Let's make a map showing the watersheds data. Each watershed has coordinates to draw it in the dataset, and tmap knows how to deal with that. It uses the same format as ggplot, but instead of `ggplot()` you will use `tm_shape()`. Then the geoms are prefixed `tm_`, so we will use `tm_fill` to show a map of the watersheds filled in with a color.

```

# Pass the watershed data to tmap and fill the polygons
tm_shape(watersheds) +
  tm_fill()

```

```

## Warning: The shape watersheds is invalid. See sf::st_is_valid

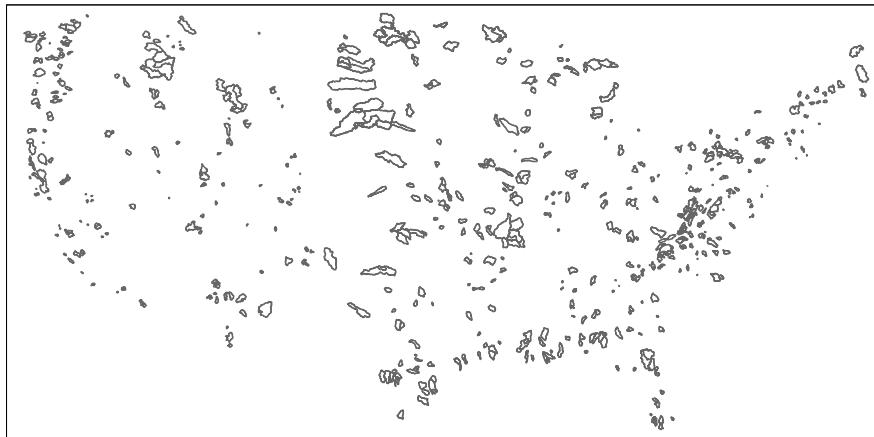
```



If we use `tm_borders` instead, it will just outline the watersheds.

```
# Add border layer to shape  
tm_shape(watersheds) +  
  tm_borders()
```

```
## Warning: The shape watersheds is invalid. See sf::st_is_valid
```

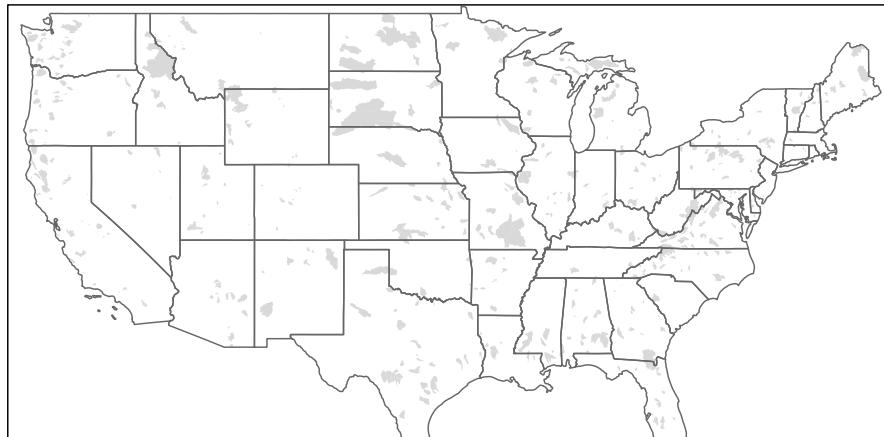


That's fun, and maybe you can guess what country we are in based on the distribution of watersheds, but it would be better to show some geopolitical boundaries to get a sense of where we are.

To do this, we will use a second tm_shape and show the states data we read in on the plot as well. Just like in ggplot, you can use multiple tm_ functions to show multiple datasets on the same map.

```
# Add border layer to shape
tm_shape(watersheds) +
  tm_fill() +
  tm_shape(states) +
  tm_borders()

## Warning: The shape watersheds is invalid. See sf::st_is_valid
```



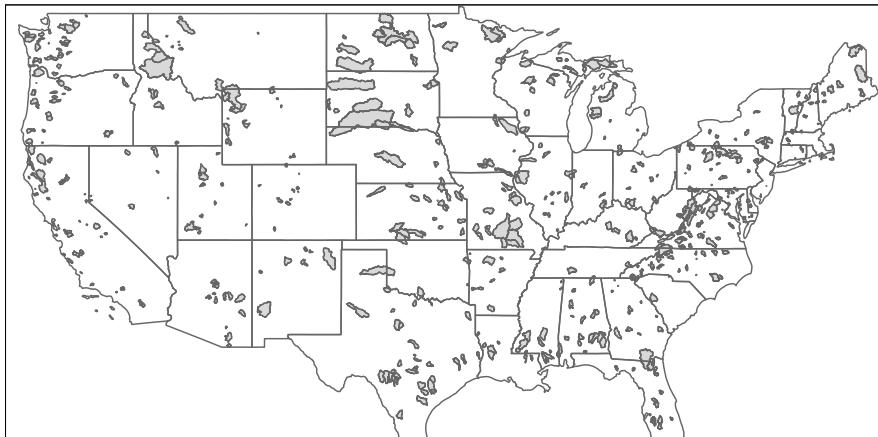
You can also show the SAME data with multiple geom's. Let's add tm_borders under the watershed portion of the map and before the states portion so we get fill and borders on our watersheds.

```
# Add fill and border layers to shape
tm_shape(watersheds) +
  tm_fill() +
  tm_borders()+
  tm_shape(states)+
```

```
tm_borders()
```



```
## Warning: The shape watersheds is invalid. See sf::st_is_valid
```



Okay, this is starting to look like a map! But we need to add elements to make it better.

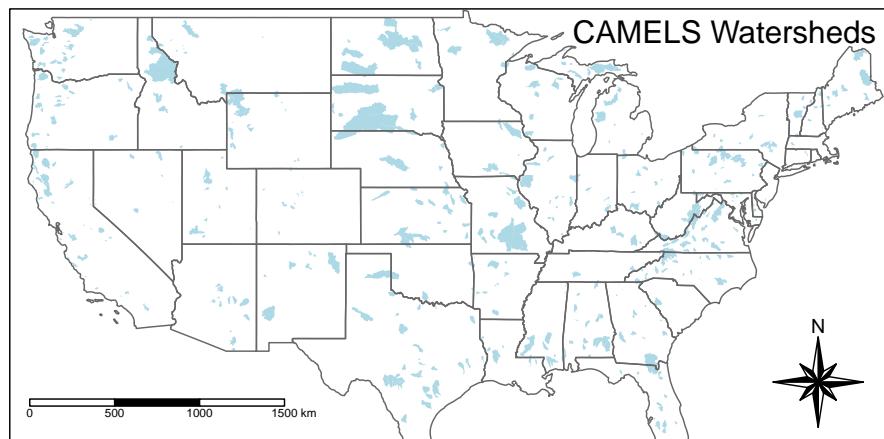
We could do this all in one statement, but you can also save your existing map as a kind of “basemap” and then add to it later, just like with a ggplot object. We will save the above map as *usa*.

Then we can use several built in geometries in tmap to add a compass, scale, and title. Note the syntax for specifying the position of the objects. Again, you could do this all in one statement too if you wanted.

```
#Save basic map object as "usa"
usa <- tm_shape(watersheds) +
  tm_fill(col = "lightblue")+
  tm_shape(states)+
  tm_borders()

usa +
  tm_compass(type = "8star", position = c("right", "bottom")) +
  tm_scale_bar(position = c("left", "bottom"))+
  tm_layout(title = "CAMELS Watersheds", title.position = c("right", "TOP"))

## Warning: The shape watersheds is invalid. See sf::st_is_valid
```



Below is an example of how to edit the “symbology” of the map. In other words, we want to color each of the polygons depending on a variable. Here we make the watersheds darker blue if they have a higher elevation.

The syntax below is basically (pseudo code):

```

Represent watersheds as shapes +
color the shapes based on elev_mean, use 10 colors, use the Blues
palette
add a legend in the bottom right, add some space for the title,
define the title, position the title
add a compass at the bottom left
add a scale bar at the bottom left

```

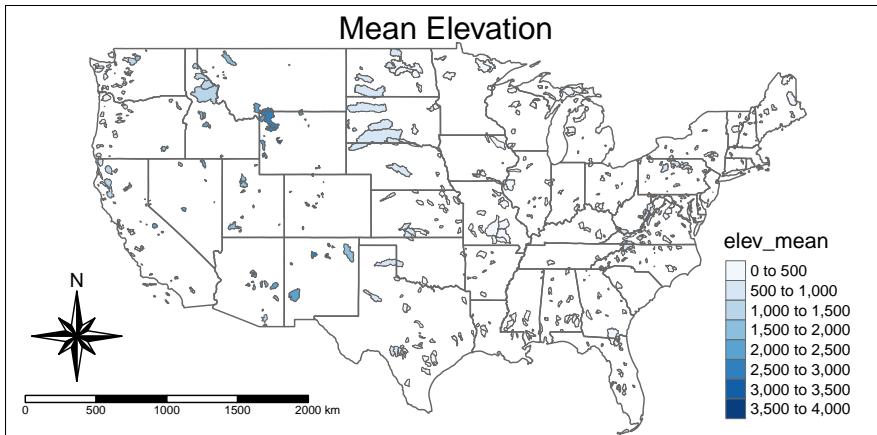
```

tm_shape(watersheds) +
  tm_fill(col = "elev_mean", n = 10, palette = "Blues")+
  tm_borders(lwd = 0.2)+
  tm_shape(states)+  

  tm_borders()+
  tm_layout(legend.position = c("right", "bottom"),
            inner.margins = 0.1,
            title = "Mean Elevation",
            title.position = c("center", "TOP"))+
  tm_compass(type = "8star", position = c("left", "bottom")) +
  tm_scale_bar(position = c("left", "bottom"))

```

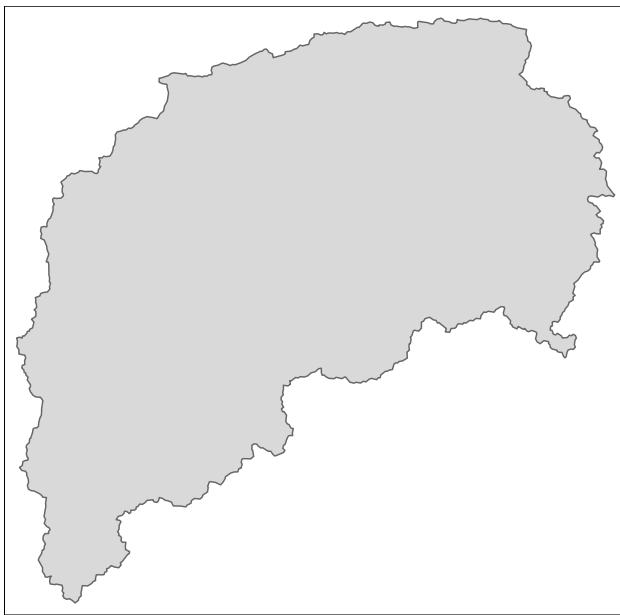
```
## Warning: The shape watersheds is invalid. See sf::st_is_valid
```



13.3 Data wrangling with tidyverse principles

You can use the same techniques as with other data to change or filter the spatial data. Below we filter to show just watershed number 3164000, which will be in quotes because the column is a character datatype. Note when we looked at the `watersheds` object above there is a column called `hru_id` for the watershed ids.

```
watersheds %>% filter(hru_id == "3164000") %>%
  tm_shape() +
  tm_fill() +
  tm_borders()
```



```
## Add non-spatial data to spatial data with a join
```

We have been using the watershed shapefile. The other dataset we read in “gageinfo” has more data, but it is just a regular tibble, not a geospatial file.

We need to attach data from gageinfo object to the watershed geospatial object based on watershed ID. How in the world will we do that?

A JOIN!!

In watershed shapefile the watershed IDs are in a column called *hru_id* and in gageinfo tibble they are in a column called *GAGE_ID*. So when we do the join, we need to tell R that these columns are the same and we want to use them to match the values. We will do a left join to accomplish this.

BUT. TWIST!

Let’s look at the first value in *hru_id* in watershed: “1013500” Now the same one in *GAGE_ID* in gageinfo: “01013500”

They’re both the same IDs, but one has a leading zero. And because they are character format, “1013500” does not equal “01013500”. We can address this a couple of ways, but the best is probably to keep them as characters, but add a leading 0 to the *hru_id* column.

We will do this by creating a new column with a mutate and using the paste0() function to add a leading 0. To make the join easier, we will also give this new column the same name as in the gageinfo tibble: “*GAGE_ID*”.

To review joins, check out chapter 7

```

#add leading 0 to hru_id column and save it as GAGE_ID
watersheds <- watersheds %>%
  mutate(GAGE_ID = paste0("0", hru_id))

#join watersheds and gageinfo using the GAGE_ID column as the key
watersheds_info <- watersheds %>%
  left_join(gageinfo, by = "GAGE_ID")

head(watersheds_info)

## Simple feature collection with 6 features and 16 fields
## Geometry type: MULTIPOLYGON
## Dimension: XY
## Bounding box: xmin: -71.34071 ymin: 44.59213 xmax: -67.69059 ymax: 47.31557
## Geodetic CRS: GCS_unknown
##   hru_id ann_P lon_cen lat_cen      AREA elev_mean ave_T july_T Perimeter
## 1 1013500    0 -68.56551 47.01169 2303988087  277.4935    NA     NA  647993
## 2 1022500    0 -68.07313 44.79691  620387273  103.6042    NA     NA  312624
## 3 1030500    0 -68.14985 45.80527 3676154745  174.4339    NA     NA  662248
## 4 1031500    0 -69.58119 45.23568  766544703  304.7435    NA     NA  309614
## 5 1047000    0 -70.16213 44.98744  904956194  379.7800    NA     NA  310157
## 6 1052500    0 -71.17197 44.96168 396110305  646.0736    NA     NA  172588
##   GAGE_ID HUC_02          GAGE_NAME      LAT
## 1 01013500    01 Fish River near Fort Kent, Maine 47.23739
## 2 01022500    01 Narraguagus River at Cherryfield, Maine 44.60797
## 3 01030500    01 Mattawamkeag River near Mattawamkeag, Maine 45.50097
## 4 01031500    01 Piscataquis River near Dover-Foxcroft, Maine 45.17501
## 5 01047000    01 Carrabassett River near North Anson, Maine 44.86920
## 6 01052500    01 Diamond River near Wentworth Location, NH 44.87739
##   LONG DRAINAGE_AREA_KM2 Elevation_m      geometry
## 1 -68.58264        2252.70    250.31 MULTIPOLYGON (((-68.06259 4...
## 2 -67.93524        573.60     92.68 MULTIPOLYGON (((-67.97836 4...
## 3 -68.30596        3676.17    143.80 MULTIPOLYGON (((-68.09162 4...
## 4 -69.31470        769.05    247.80 MULTIPOLYGON (((-69.31629 4...
## 5 -69.95510        909.10    310.38 MULTIPOLYGON (((-70.10847 4...
## 6 -71.05749        383.82    615.70 MULTIPOLYGON (((-71.10862 4...

```

And now we can plot this formerly non-spatial data on our map.

In this case, we can now add the name of the watershed to the map rather than the number.

```

#now the gageinfo columns are available for us to use when mapping
watersheds_info %>% filter(hru_id == "3164000") %>%
  tm_shape() +

```

```
tm_fill() +
tm_borders() +
tm_text("GAGE_NAME", size = 1)
```



We can also subset vector data to create new datasets or plot. Below we will use filter statements to grab all the polygons in Virginia from the watersheds shapefile and then the Virginia state outline from the states shapefile.

The method we will use for getting the Virginia watersheds is a little new. We are going to use the grepl() function to grab any gages that include the text ", VA" since there isn't a state column, but the watershed names include the state.

Why are we doing ", VA" and not just "VA"? Good question!

", VA" will most likely only show up in gage names that are in Virginia because they'll be in the format "Gage name, VA". If we just say "VA" we might get some gages that have "VA" as part of their name but are not in Virginia.

Once we successfully filter to Virginia, we will then make a nice map of the CAMELS watersheds in Virginia!

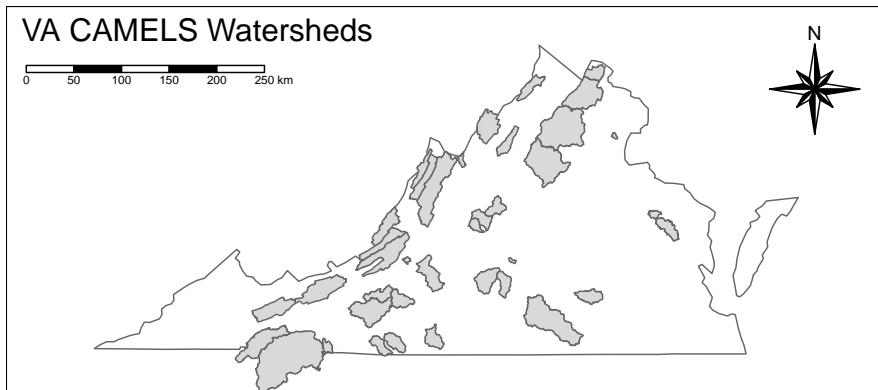
```
va_watersheds <- filter(watersheds_info, grepl(", VA", GAGE_NAME))
va_outline <- filter(states, NAME == "Virginia")

va_outline %>%
  tm_shape() +
```

```

tm_borders() +
tm_shape(va_watersheds) +
tm_fill() +
tm_borders() +
tm_layout(inner.margins = 0.1,
          title = "VA CAMELS Watersheds",
          title.position = c("left", "TOP"))+
tm_compass(type = "8star", position = c("right", "top")) +
tm_scale_bar(position = c("left", "top"))

```



In addition to filtering, we can use the data in the attribute table to calculate additional parameters, just like with a normal object. Below we calculate the ratio of Area to Perimeter by changing the perimeter datatype to numeric and then dividing area by perimeter.

```

va_watersheds <- va_watersheds %>%
  mutate(PerimeterNum = as.numeric(Perimeter),
        Area_Perimeter = AREA / PerimeterNum)

head(va_watersheds)

## Simple feature collection with 6 features and 18 fields
## Geometry type: MULTIPOLYGON
## Dimension:      XY

```

```

## Bounding box: xmin: -79.32293 ymin: 38.31301 xmax: -77.55629 ymax: 39.2801
## Geodetic CRS: GCS_unknown
##   hru_id ann_P lon_cen lat_cen      AREA elev_mean ave_T july_T Perimeter
## 1 1620500     0 -79.27321 38.37592 61339485  913.1406    NA     NA    69535
## 2 1632000     0 -78.96078 38.70073 547286614  605.6940    NA     NA   175812
## 3 1632900     0 -78.72713 38.54450 249206633  410.4016    NA     NA   134796
## 4 1634500     0 -78.44939 39.06919 264494917  441.7808    NA     NA   151377
## 5 1638480     0 -77.67379 39.20060 233835164  175.1224    NA     NA   112488
## 6 1644000     0 -77.80391 38.99276 859599217  187.0871    NA     NA   237588
##   GAGE_ID HUC_02          GAGE_NAME      LAT      LONG
## 1 01620500     02 NORTH RIVER NEAR STOKESVILLE, VA 38.33763 -79.24004
## 2 01632000     02 N F SHENANDOAH RIVER AT COOTES STORE, VA 38.63706 -78.85280
## 3 01632900     02 SMITH CREEK NEAR NEW MARKET, VA 38.69345 -78.64279
## 4 01634500     02 CEDAR CREEK NEAR WINCHESTER, VA 39.08122 -78.32945
## 5 01638480     02 CATOCTIN CREEK AT TAYLORSTOWN, VA 39.25455 -77.57638
## 6 01644000     02 GOOSE CREEK NEAR LEESBURG, VA 39.01955 -77.57749
##   DRAINAGE_AREA_KM2 Elevation_m      geometry PerimeterNum
## 1                 44.74    786.35 MULTIPOLYGON (((-79.24689 3...
## 2                 543.36    543.76 MULTIPOLYGON (((-78.98026 3...
## 3                 245.05    417.29 MULTIPOLYGON (((-78.85841 3...
## 4                 263.98    344.44 MULTIPOLYGON (((-78.37821 3...
## 5                 232.00    155.98 MULTIPOLYGON (((-77.62354 3...
## 6                 859.17    187.25 MULTIPOLYGON (((-77.62354 3...
##   Area_Perimeter
## 1     882.1383
## 2     3112.9082
## 3     1848.7688
## 4     1747.2596
## 5     2078.7565
## 6     3618.0245

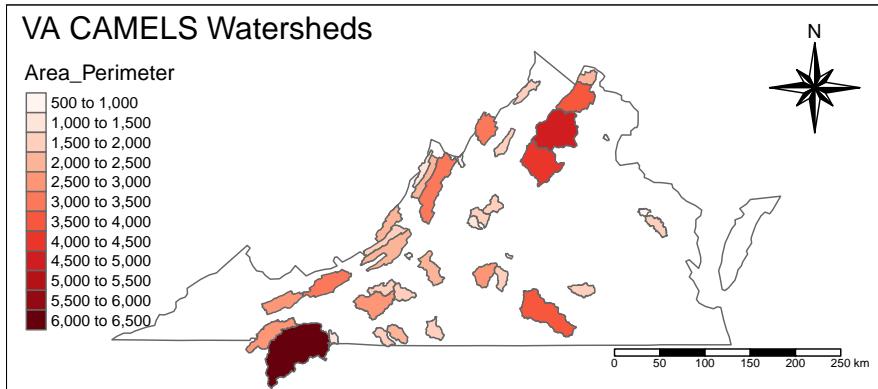
```

Now we can plot our newly calculated data by controlling color with that new column name.

```

va_outline %>%
  tm_shape() +
  tm_borders() +
  tm_shape(va_watersheds) +
  tm_fill(col = "Area_Perimeter", n = 10, palette = "Reds") +
  tm_borders() +
  tm_layout(inner.margins = 0.12,
            title = "VA CAMELS Watersheds",
            title.position = c("left", "TOP")) +
  tm_compass(type = "8star", position = c("right", "top")) +
  tm_scale_bar(position = c("right", "bottom"))

```

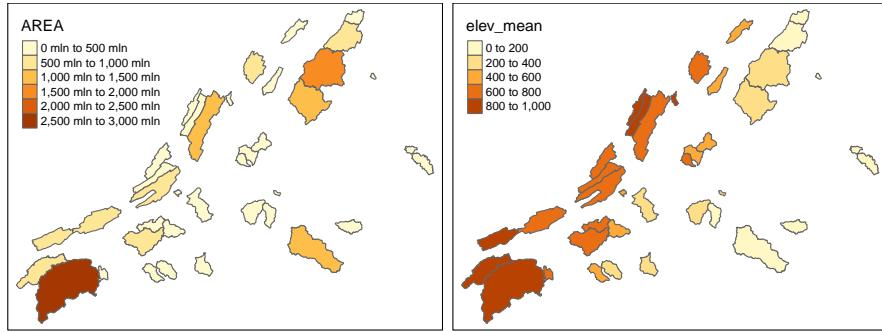


13.4 Plot maps side by side

Just like we can use facets in ggplot, we can use facets to show multiple maps. Below we color our map by AREA and elev_mean and put them next to each other using tm_facets.

```
facets = c("AREA", "elev_mean")

tm_shape(va_watersheds) +
  tm_polygons(facets) +
  tm_facets(ncol = 2, sync = FALSE)
```

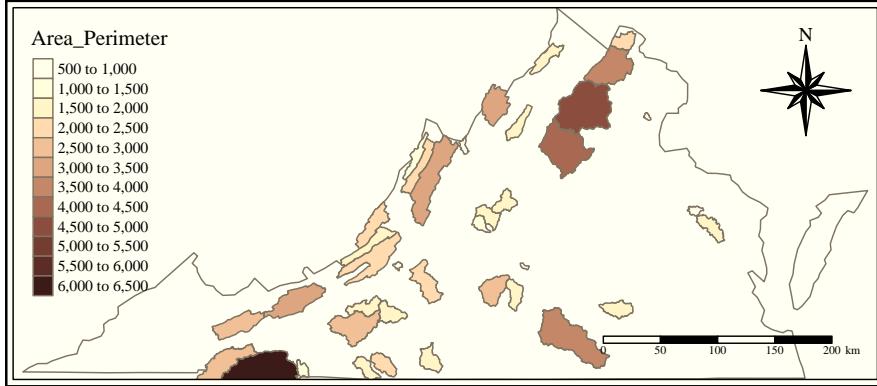


13.5 Built in styles, like themes in ggplot

Tmap also has built in styles, which are like themes in ggplot. We can use these styles with `tm_style`. Try “classic”, “coblat”, or “col_blind” below.

```
va_outline %>%
  tm_shape() +
  tm_borders() +
  tm_shape(va_watersheds) +
  tm_fill(col = "Area_Perimeter", n = 10, palette = "Reds") +
  tm_borders()+
  tm_layout(inner.margins = 0.12,
            title = "VA CAMELS Watersheds",
            title.position = c("left", "TOP"))+
  tm_compass(type = "8star", position = c("right", "top")) +
  tm_scale_bar(position = c("right", "bottom"))+
  tm_style("classic") #try cobalt, bw, col_blind
```

```
## Note that tm_style("classic") resets all options set with tm_layout, tm_view, tm_for
```



13.6 Interactive Maps

13.6.1 tmap

You can also generate maps that you can interact with, as opposed to static maps, that we have been using before. If you are generating a map for an app or webpage, this may be a good choice. But for a pdf report, the static maps are more appropriate.

In tmap all you have to do is run `tmap_mode("view")` and it will create an interactive map with the exact same syntax! To switch back to a static map, run `tmap_mode("plot")`

Also in this chunk we see how to add a basemap to a tmap object, using `tm_basemap`.

```
tmap_mode("view")

## tmap mode set to interactive viewing

usa <- tm_shape(watersheds_info) +
  tm_fill(col = "lightblue", alpha = 0.3) +
  tm_borders() +
```

```

tm_shape(states) +
  tm_borders() +
  tm_layout(title = "CAMELS Watersheds", title.position = c("right", "TOP"))

usa + tm_basemap(server = "OpenTopoMap")

## Warning: The shape watersheds_info is invalid (after reprojection). See
## sf:::st_is_valid

```

13.6.2 Leaflet

Leaflet is another way to make interactive maps. Its syntax is very different, as you can see below. But depending on what functionality you need, it could be a better choice.

```

leaflet(watersheds_info) %>%
  addTiles() %>%
  addPolygons(color = "#444444", weight = 1, smoothFactor = 0.5,
              opacity = 1.0, fillOpacity = 0.5,
              fillColor = ~colorQuantile("YlOrRd", elev_mean)(elev_mean))

## Warning: sf layer has inconsistent datum (+proj=longlat +datum=NAD83 +no_defs).
## Need '+proj=longlat +datum=WGS84'

```

Chapter 14

Summative Assessment 2

14.1 Info for assessment

To complete this assessment, go to the repository linked below and either copy it to your github account or download the repository, just as you do for other assignments and activities in class.

Github repo: <https://github.com/VT-Hydroinformatics/14-Summative2>

Chapter 15

Geospatial R Raster - Hydro Analyses

15.1 Introduction

The following activity is available as a template github repository at the following link: <https://github.com/VT-Hydroinformatics/13-Geospatial-Raster-Hydro.git>

For more: <https://geocompr.robinlovelace.net/spatial-class.html#raster-data>

To read in detail about any of the WhiteboxTools used in this activity, check out the user manual: https://jblindsay.github.io/wbt_book/intro.html

In this activity we are going to explore how to work with raster data in R while computing several hydrologically-relevant landscape metrics using the R package whitebox tools. Whitebox is very powerful and has an extensive set of tools, but it is not on CRAN. You must install it with the commented-out line at the top of the next code chunk.

Install/Load necessary packages and data:

```
library(tidyverse)
library(raster)
library(sf)
library(whitebox)
library(tmap)

#If installing/using whitebox for the first time
#install_whitebox()
```

```
whitebox::wbt_init()
theme_set(theme_classic())
```

15.2 Read in DEM

First, we will set tmap to either map or view depending on how we want to see our maps. I'll often set to map unless I specifically need to view the maps interactively because if they are all set to view it makes scrolling through the document kind of a pain: every time you hit a map the scroll zooms in or out on the map rather than scrolling the document.

For this activity we are going to use a 5-meter DEM of a portion of a Brush Mountain outside Blacksburg, VA.

What does DEM stand for? What does it show?
 What does it mean that the DEM is “5-meter”?

We will use raster() to load the DEM. We let R know that coordinate system is WGS84 by setting the crs argument equal to ‘+init=EPSG:4326’, where 4326 is the EPSG number for WGS84.

Next, an artifact of outputting the DEM for this analysis is that there are a bunch of errant cells around the border that don't belong in the DEM. If we make a map with them, it really throws off the scale. So we are going to set any elevation values below 1500 ft to NA. Note how this is done as if the dem was just a normal vector. COOL!

```
tmap_mode("view")
## tmap mode set to interactive viewing
dem <- raster("McDonaldHollowDEM/brushDEMsm_5m.tif", crs = '+init=EPSG:4326')
writeRaster(dem, "McDonaldHollowDEM/brushDEMsm_5m_crs.tif", overwrite = TRUE)
dem[dem < 1500] <- NA
```

15.3 Plot DEM

Now let's plot the DEM. We will use the same syntax as we did in the previous lecture about vector data. Give tm_shape the raster, then visualize it with tm_raster. We will tell tmap that the scale on the raster is continuous, which

color palette to use, whether or not to show the legend, and then add a scale bar.

```
tm_shape(dem)+  
  tm_raster(style = "cont", palette = "PuOr", legend.show = TRUE)+  
  tm_scale_bar()
```

15.4 Generate a hillshade

Since the DEM is just elevation values, it is tough to see much about the landscape. We can see the stream network and general features, but not much more. BUT those features are there! We just need other ways to illuminate them. One common way to visualize these kind of data is a hillshade. This basically “lights” the landscape with a synthetic sun, casting shadows and illuminating other features. You can control the angle of the sun and from what direction it is shining to control the look of the image. We will position the sun in the south-south east so it illuminates the south side of Brush Mountain well.

We will use the whitebox tools function `wbt_hillshade()` to produce a hillshade.

15.4.1 How whitebox tools functions work

The whitebox tools functions work can be a little tricky to work with at first. You might want to pass R objects to them and get R objects back, but that's not how they are set up.

Basically for your input, you well the `wbt` function the name of the file that has the input data.

For output, you tell it what to name the output.

The `wbt` function then outputs the calculated data to your working directory, or whatever directory you give it in the `output` argument.

This means if you want to do something with the output of the `wbt` function, you have to read it in separately.

In the chunk below we will read in the brush mountain 5m DEM and output a hillshade with `wbt_hillshad()`.

We will then read the output hillshade in with `raster()` and make a map with `tmap`. We will use the “Greys” palette in revers by adding a negative sign in front of it. This is just to make the hillshade look nice.

Notice how much more you can see in the landscape!

```
wbt_hillshade(dem = "McDonaldHollowDEM/brushDEMsm_5m_crs.tif",
               output = "McDonaldHollowDEM/brush_hillshade.tif",
               azimuth = 115)

hillshade <- raster("McDonaldHollowDEM/brush_hillshade.tif")

tm_shape(hillshade) +
  tm_raster(style = "cont", palette = "-Greys", legend.show = FALSE) +
  tm_scale_bar()
```

15.5 Prepare DEM for Hydrology Analyses

Alright, we are cooking now.

But we have to cool our jets for a second. If we are going to do hydrologic analyses, we have to prep our DEM a bit.

Our hydrologic tools often work on the premise of following water down the hillslope based on the elevation of the cells in the DEM. If, along a flowpath, there is no cell lower than a location, the algorithm we are using will stop there. This is called a sink, pit, or depression. See the figure below.

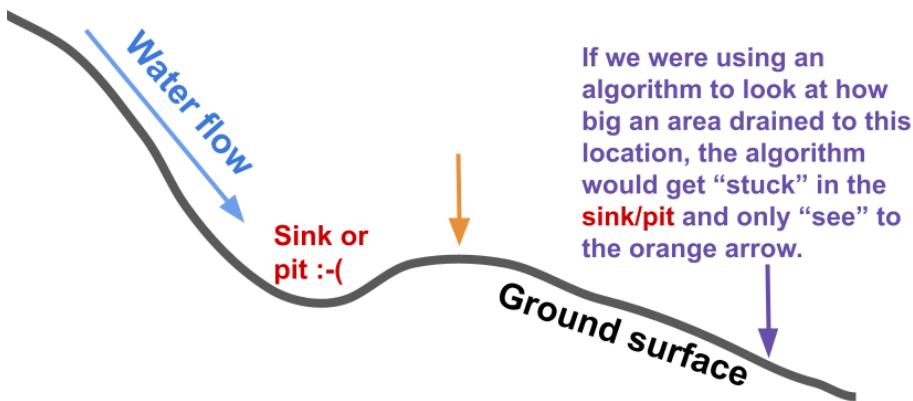


Figure 15.1: Sink

We can deal with these features two ways. The first is to “fill” them. Which means the dead end cells will have their elevations raised until the pit is filled and water can flow downhill. See below.

The second way we can deal with these features is to “breach” them. This means the side of the feature that is blocking flow will be lowered to allow water to flow downhill. See below.

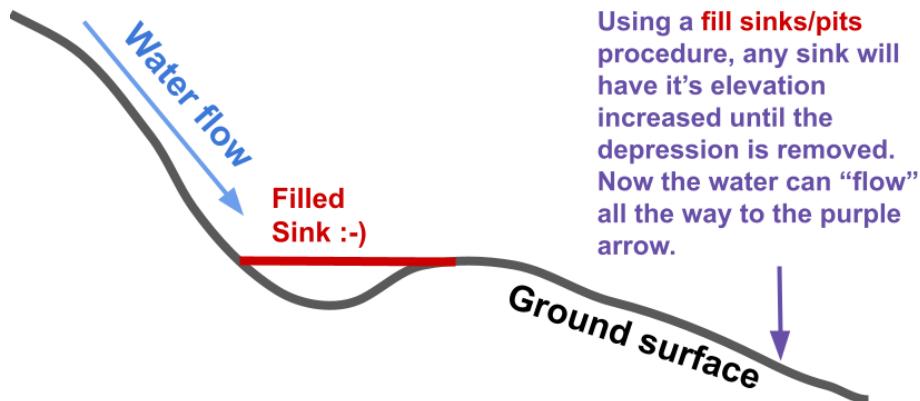


Figure 15.2: Filled

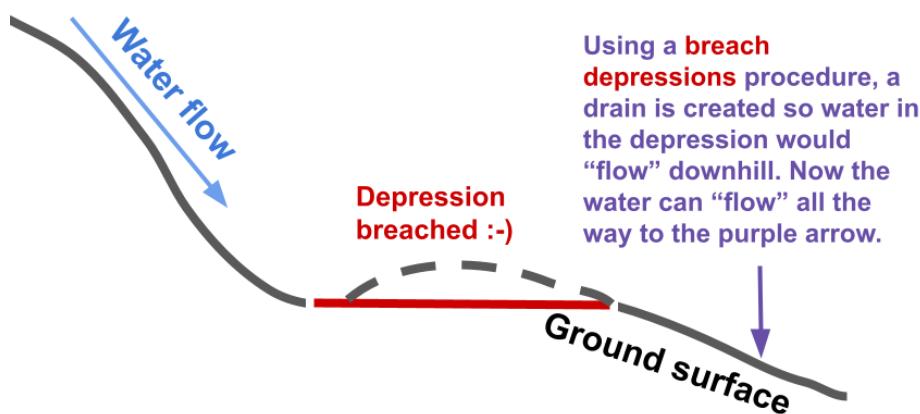


Figure 15.3: Breached

We are going to do both to prep our DEM. We will first breach depressions using `wbt_breach_depressions_least_cost()`, which will lower the elevation of the cells damming depressions.

To use this function we will also give it a maximum distance to search for a place to breach the depression (`dist`) and tell with whether to fill any depressions leftover after it does its thing (`fill`).

Then, because this algorithm can leave some remaining depressions, we wil use `wbt_fill_depressions_wang_and_liu()` to clean up any remaining issues.

Be careful to give `wbt_fill_depressions_wang_and_liu()` the result of the breach depressions function, not the original DEM!

```
wbt_breach_depressions_least_cost(
  dem = "McDonaldHollowDEM/brushDEMsm_5m_crs.tif",
  output = "McDonaldHollowDEM/bmstationdem_breached.tif",
  dist = 5,
  fill = TRUE)

wbt_fill_depressions_wang_and_liu(
  dem = "McDonaldHollowDEM/bmstationdem_breached.tif",
  output = "McDonaldHollowDEM/bmstationdem_filled_breached.tif"
)
```

15.6 Visualize filled sinks and breached depressions

Now let's look at what this did. This is a great example of how easily you can use multiple rasters together.

If we want to see how the fill and breach operations changed the DEM, we can just subtract the filled and breached DEM from the original. Then, in areas where nothing changed, the values will be zero, areas that were filled will be positive, and areas that were decreased in elevation to “breach” a depression will be negative.

To more easily see where stuff happened, we will set all the cells that equal zero to NA. Then we will plot them on the hillshade.

Where where changes made?

What do you think these pit/depression features represent in real life?

```
filled_breached <- raster("McDonaldHollowDEM/bmstationdem_filled_breached.tif")

## What did this do?
```

```

difference <- dem - filled_breached

difference[difference == 0] <- NA

tm_shape(hillshade) +
  tm_raster(style = "cont", palette = "-Greys", legend.show = FALSE) +
  tm_scale_bar() +
tm_shape(difference) +
  tm_raster(style = "cont", legend.show = TRUE) +
  tm_scale_bar()

## Variable(s) "NA" contains positive and negative values, so midpoint is set to 0. Set midpoint

```

15.7 D8 Flow Accumulation

The first hydrological analysis we will perform is the D8 flow accumulation algorithm. In whitebox this is `wbt_d8_flow_accumulation()`. We give the function the DEM, and for each cell, it determines the direction water fill flow from that cell. To do this it looks at the elevation of the surrounding cells relative to the current cell. In the D8 algorithm, the flow direction can be one of 8 directions, show in the figure below. All flow from the current cell is moved to the cell to which the flow direction points.

Using another function, we can just output the flow direction for each cell. This will be important when we delineate a watershed, but for visualization and many analysis purposes, we just want to look at the flow accumulation.

This tool outputs a raster that tells us how many cells drain to each cell. In other words, for a given cell in the raster, its value corresponds to the number of cells that drain to it. As a result, this highlights streams quite well.

The `wbt_d8_flow_accumulation()` function takes an input of a DEM or a flow direction (pointer) file. We will pass it out filled, breached DEM. The default output is the number of cells draining to each cell, but you can also choose specific contributing area or contributing area.

We will visualize our output by plotting the log of the D8 flow accumulation grid over the hillshade with an opacity of 0.5 using the alpha parameter in `tm_raster`. Mapping with the hillshade helps us see the flow accumulation in the context of the landscape. Plotting the log values helps us see differences in flow accumulation, because the high values are so much higher than the low values in a flow accumulation grid.

Where are the highest values? Where are the lowest values?

D8 Flow Direction

- Which direction will water flow from the target (blue) cell?
- 8 possible directions (arrows below)
- Determined by elevation of surrounding cells
- All flow from target cell goes in one direction (red cell)

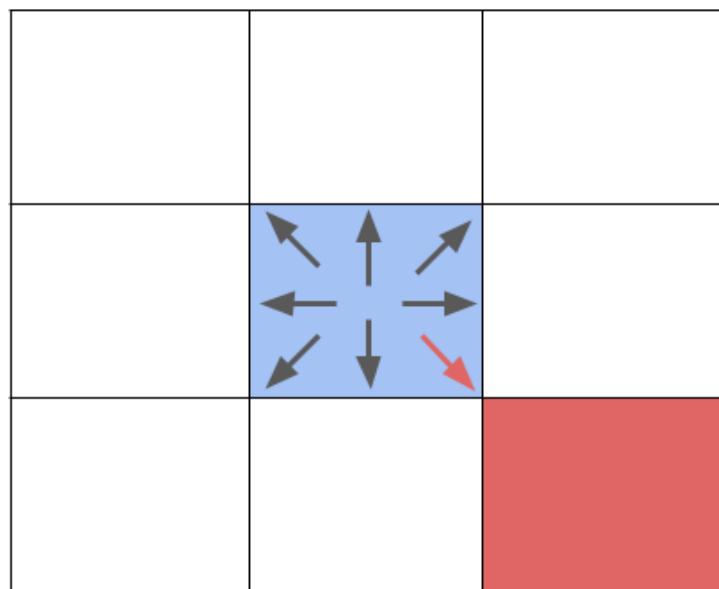


Figure 15.4: D8 Flow Direction

```
wbt_d8_flow_accumulation(input = "McDonaldHollowDEM/bmstationdem_filled_breached.tif",
                         output = "McDonaldHollowDEM/D8FA.tif")

d8 <- raster("McDonaldHollowDEM/D8FA.tif")

tm_shape(hillshade)+
  tm_raster(style = "cont", palette = "-Greys", legend.show = FALSE)+
tm_shape(log(d8))+
  tm_raster(style = "cont", palette = "PuOr", legend.show = TRUE, alpha = .5)+
  tm_scale_bar()
```

15.8 D infinity flow accumulation

Another method of calculating flow accumulation is the D infinity algorithm. This operates similarly to the D8 algorithm but with some important differences.

With D infinity, the flow direction for each cell can be any angle. Endless possibilities!

If the angle does not point squarely at one of the neighboring cells, the flow from the focus cell can be SPLIT between neighboring cells. We will see the resulting difference this makes when we look at the output.

The function for d infinity is `wbt_d_inf_flow_accumulation()` and like the D8 function it will take a flow direction (pointer) file or a DEM. We will give it out filled and breached DEM.

As with the D8 data, we will plot the logged accumulation values over a hillshade with 50% opacity.

How does this look different from the D8 results?

Which do you think represents reality better? Why?

```
wbt_d_inf_flow_accumulation("McDonaldHollowDEM/bmstationdem_filled_breached.tif",
                            "McDonaldHollowDEM/DinfFA.tif")

dinf <- raster("McDonaldHollowDEM/DinfFA.tif")

tm_shape(hillshade)+
  tm_raster(style = "cont", palette = "-Greys", legend.show = FALSE)+
tm_shape(log(dinf))+
  tm_raster(style = "cont", palette = "PuOr", legend.show = TRUE, alpha = 0.5)+
  tm_scale_bar()

## Variable(s) "NA" contains positive and negative values, so midpoint is set to 0. Set midpoint
```

D-infinity Flow Direction

- Which direction will water flow from the target (blue) cell?
- Possible flow direction is continuous (infinite)
- Determined by elevation of surrounding cells
- Depending on direction (black arrow), flow can be split between multiple cells (red cells)

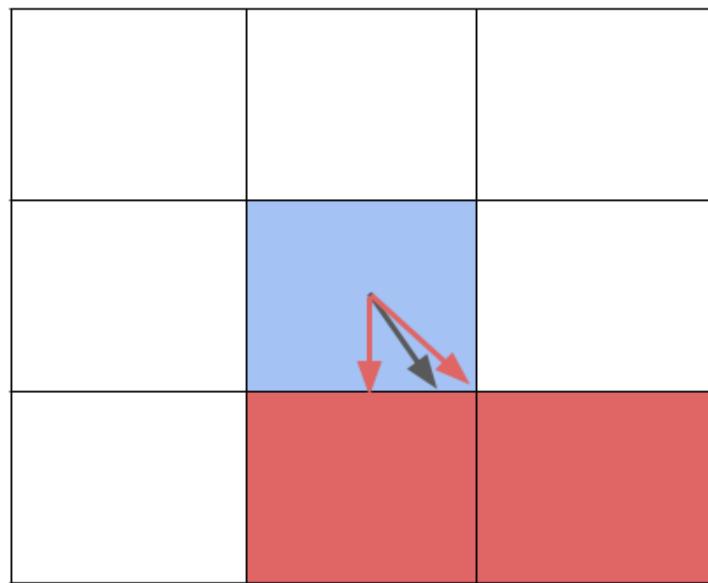


Figure 15.5: D inf Flow Direction

15.9 Topographic Wetness Index

The topographic wetness index (TWI) combines flow accumulation with the slope of each cell to calculate a index that basically corresponds to how likely an area is to be wet. If we think about a landscape, an area with a lot of contributing area and a flat slope is more likely than an area with a lot of contributing area and a steep slope to be wet. That's what TWI measures.

The formula for TWI is

$$TWI = \ln(As/\tan(Slope))$$

Where As is the specific contributing area and Slope is the slope at the cell. Specific contributing area is contributing area per unit contour length.

This is the first function we will use that doesn't just take a DEM as input. The wbt_wetness_index() function takes a raster of specific contributing area and another of slope. We will need to create each of these first and then pass them to the function (since we didn't calculate specific contributing area when we used the flow accumulation algorithms above)

We will then plot TWI with the same approach we used for the flow accumulation data.

We get some funkiness around the edges of the DEM, so we will also filter the twi output so the visualization looks better.

```
wbt_d_inf_flow_accumulation(input = "McDonaldHollowDEM/bmstationdem_filled_breached.tif",
                             output = "McDonaldHollowDEM/DinfFAsca.tif",
                             out_type = "Specific Contributing Area")

wbt_slope(dem = "McDonaldHollowDEM/bmstationdem_filled_breached.tif",
           output = "McDonaldHollowDEM/demslope.tif",
           units = "degrees")

wbt_wetness_index(sca = "McDonaldHollowDEM/DinfFAsca.tif",
                  slope = "McDonaldHollowDEM/demslope.tif",
                  output = "McDonaldHollowDEM/TWI.tif")

twi <- raster("McDonaldHollowDEM/TWI.tif")

twi[twi > 0] <- NA

tm_shape(hillshade)+
  tm_raster(style = "cont", palette = "-Greys", legend.show = FALSE)+
  tm_shape(twi)+
  tm_raster(style = "cont", palette = "PuOr", legend.show = TRUE, alpha = 0.5)+
  tm_scale_bar()
```

15.10 Downslope TWI

Another index of interest is the downslope index. The downslope index is the measure of slope between a cell and then another cell a specified distance downslope. We can replace the regular slope number in the TWI calculation with downslope index to calculate TWId or Downslope TWI.

$$TWI = \ln(As/\tan(DownslopeIndex))$$

As is specific contributing area, and the tan of the downslope index can be output from the whitebox downslope index tool. We have to do this calculation with the rasters because there isn't a specific whitebox tool that calculates TWId.

Essentially, this index captures the fact that you should expect a 50 meter long bench on a hillslope to be wetter than a 1 meter long bench, because the 1 meter one would drain a lot faster.

Similar to TWI we get some edge issues so we will filter the result to make the visualization look better.

```
wbt_downslope_index(dem = "McDonaldHollowDEM/bmstationdem_filled_breached.tif",
                     output = "McDonaldHollowDEM/downslope_index.tif",
                     out_type = "tangent")

downslope_index <- raster("McDonaldHollowDEM/downslope_index.tif")

dinfFA <- raster("McDonaldHollowDEM/DinfFAsca.tif")

twid <- log(dinfFA / downslope_index)

twid[twid > 40] <- NA

tm_shape(hillshade) +
  tm_raster(style = "cont", palette = "-Greys", legend.show = FALSE) +
tm_shape(twid) +
  tm_raster(style = "cont", palette = "PuOr", legend.show = TRUE, alpha = 0.5) +
  tm_scale_bar()
```

15.11 Map Stream Network

One neat and often very useful thing we can do with the flow accumulation grids we calculated, is map the stream network in a watershed. If we look at either flow accumulation grid we can see the highest values are in the streams. Therefore if we determine the flow accumulation value at the highest place on the

streamnetwork with consistent flow, we can set all cells with a flow accumulation lower than that to NO and we will only have cells that are in the stream.

Often, we actually want our stream network to be represented as lines, so we then have to convert that raster to a vector format.

Whitebox Tools has two handy functions to let us do this: `wbt_extract_streams()` makes a raster of the stream network by using a threshold flow accumulation you give it. It takes a D8 flow accumulation grid as input.

Then `wbt_raster_streams_to_vector()` will take the output from `wbt_extract_streams()` and a D8 pointer file and output a shapefile of your stream network.

Below we extract the streams, generate a D8 pointer file, and then convert the raster streams to vector. We will then plot the streams on the hillshade.

```
wbt_extract_streams(flow_accum = "McDonaldHollowDEM/D8FA.tif",
                    output = "McDonaldHollowDEM/raster_streams.tif",
                    threshold = 6000)

wbt_d8_pointer(dem = "McDonaldHollowDEM/bmstationdem_filled_breached.tif",
               output = "McDonaldHollowDEM/D8pointer.tif")

wbt_raster_streams_to_vector(streams = "McDonaldHollowDEM/raster_streams.tif",
                            d8_pntr = "McDonaldHollowDEM/D8pointer.tif",
                            output = "McDonaldHollowDEMstreams.shp")

streams <- st_read("McDonaldHollowDEMstreams.shp")

## Reading layer `streams' from data source
##   `/Users/jpgannon/My Drive (jpgannon@vt.edu)/CLASSES/SPRING Hydroinformatics/Hydroinformatics
##   using driver `ESRI Shapefile'
## Simple feature collection with 43 features and 2 fields
## Geometry type: LINESTRING
## Dimension:      XY
## Bounding box:  xmin: -80.49736 ymin: 37.23593 xmax: -80.46384 ymax: 37.25638
## CRS:           NA

tm_shape(hillshade)+
  tm_raster(style = "cont", palette = "-Greys", legend.show = FALSE) +
  tm_shape(streams) +
  tm_lines(col = "blue") +
  tm_scale_bar()

## Warning: Correct projection of shape streams unknown. Long-lat (WGS84) is
## assumed.
```

15.12 Extract raster values to point locations

Now let's say you have some sample or monitoring sites in this study area. You may want to know what the values of the rasters we just made are for your sites. Below we will read in the coordinates of a few sites and extract data from a single raster and then multiple at once.

15.12.1 Import and plot points

First, we will read in a csv of point locations.

Then we have to convert it to a spatial datatype. We will use `SpatialPoints()` to do this. We need to give it our points, and then tell it what projection our data is in. We are using geographic coordinates (longlat) and the WGS84 datum (most gps' use this).

Then we will plot the points just to make sure they show up where we expect.

```
points <- read_csv("McDonaldHollowDEM/points.csv")

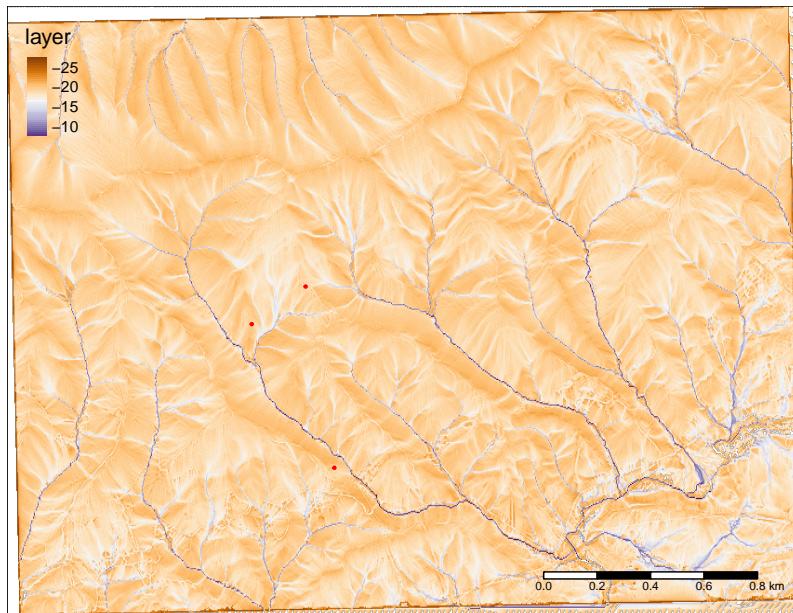
## Rows: 3 Columns: 2
## -- Column specification -----
## Delimiter: ","
## dbl (2): lon, lat
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

pointsSP <- SpatialPoints(points, proj4string = CRS("+proj=longlat +datum=WGS84"))

tmap_mode("plot")

## tmap mode set to plotting

tm_shape(twid) +
  tm_raster(style = "cont", palette = "PuOr", legend.show = TRUE) +
  tm_scale_bar() +
  tm_shape(pointsSP) +
  tm_dots(col = "red")
```



```
#### Extract values from a single raster
```

We can extract values from any raster using the `extract()` function in the `raster` package. We give this function the raster we want to pull data from (`x`) and the points where we want data (`y`). Then we specify how we want it to grab data from the raster. “Simple” just grabs the value at the locations specified. If we specify “bilinear” it will return an interpolated value based on the four nearest raster cells.

This function just spits out a vector of values disconnected from our points, so next we will add the column to our existing points object. NOT the geospatial one, just the datafram/tibble.

```
twidvals <- extract(x = twid, #raster
                      y = pointsSP, #points
                      method = "simple")
points$twid <- twidvals

points

## # A tibble: 3 x 3
##       lon     lat    twid
##   <dbl> <dbl> <dbl>
## 1 -80.5  37.2 -19.9
## 2 -80.5  37.2 -19.6
## 3 -80.5  37.2 -18.3
```

15.12.2 Extract values from multiple rasters at once

Maybe you want a bunch of topographic information for all your sample sites. You could repeat the process above for each one, or we can stack the rasters of interest and then pull out values for each using the same method as below. WHOA!

So: we give stack() each raster we want data from, using the rasters we read in earlier in the activity

Then: use the same exact syntax as when we extracted values from a single raster, but give the extract function the stacked raster.

We can then use cbind() to slap the extracted data onto our existing dataframe (gain, NOT the spatial one, just the original regular one)

```
slope <- raster("McDonaldHollowDEM/demslope.tif")

raster_stack <- stack(twi, twid, slope, dem)

raster_values <- extract(x = raster_stack, #raster
                          y = pointsSP, #points
                          method = "simple")

points <- cbind(points, raster_values)

points

##          lon      lat      twid      TWI      layer demslope brushDEMsm_5m
## 1 -80.48355 37.24087 -19.90985 -8.347021 -19.90985 40.47319      2130.564
## 2 -80.48705 37.24570 -19.64899 -8.164754 -19.64899 57.50642      2301.624
## 3 -80.48477 37.24697 -18.29477 -6.953844 -18.29477 32.81537      2430.992
```

15.13 View raster data as a PDF or histogram

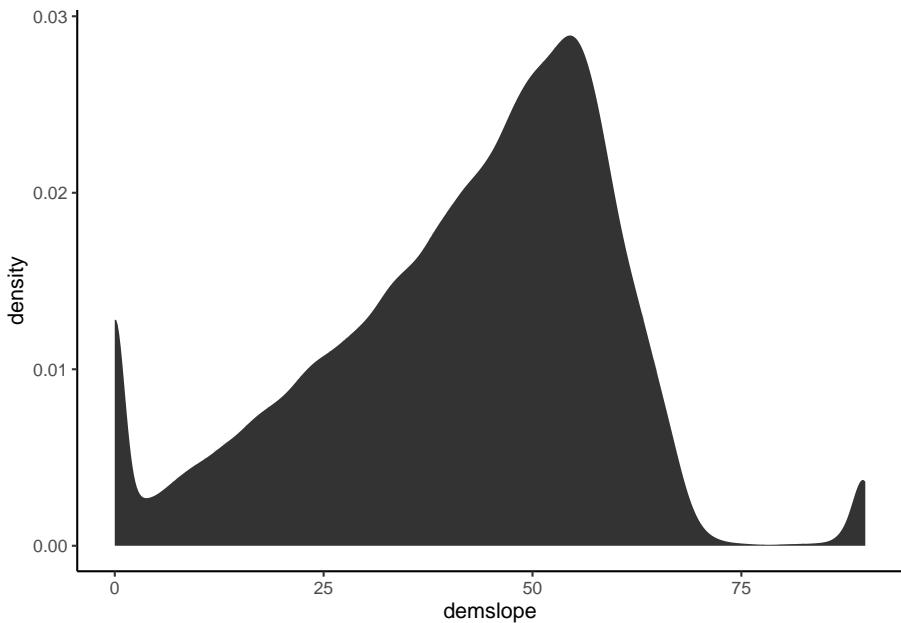
Often it can be useful to look at a summary of different topographic characteristics in an area or watershed outside of a map. One way we can do this is to look at a histogram or pdf of the values in our map by converting the raster values to a dataframe and plotting with ggplot.

We will use as.data.frame to convert our slope raster to a data frame and then plot the result in ggplot with stat_density.

It would be very difficult to accurately describe the differences in slope between two areas by looking at a map of the values, but this way we can do it quite effectively.

```
slopedf <- as.data.frame(slope)

ggplot(slopedf, aes(demslope)) +
  stat_density()
```



15.14 Subsetting a raster for visualization

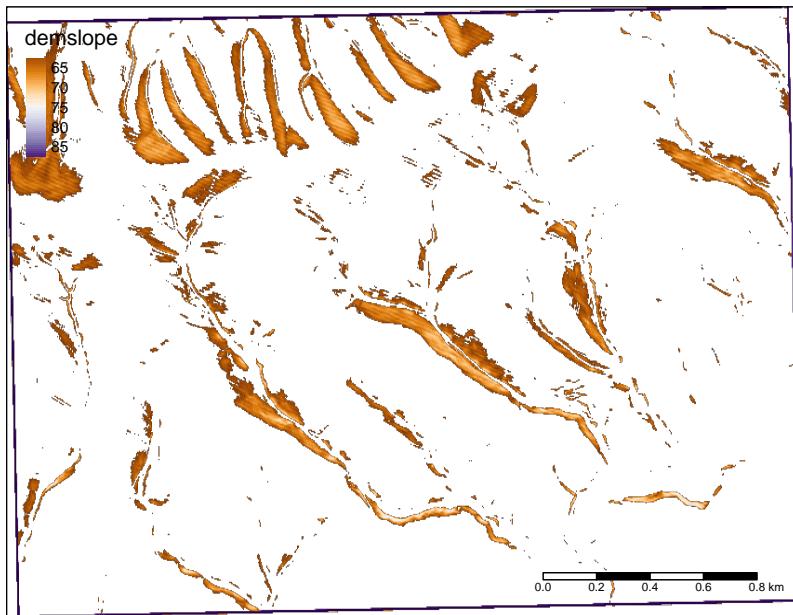
Similar to how we set junky values to NA earlier in this activity, we can also use this as a visualization tool. We will create a new raster with areas with a slope of less than 60 percent set to NA. When we plot this, it will just show slope where slope is greater than 60.

At its heart this is a reclassification. You could use the same strategy to classify slopes into bins. For example, make slopes from 0 - 60 percent equal to 1, for low slope, and then > 60 equal to 2 for high slope... or any number of bins. This can be really useful for finding locations that satisfy a several criteria. Think: where might we find the right habitat for a certain tree species, or bird, or sasquatches?

```
slope2 <- slope
slope2[slope2 < 60] <- NA

tm_shape(slope2)+
```

```
tm_raster(style = "cont", palette = "PuOr", legend.show = TRUE)+  
tm_scale_bar()
```



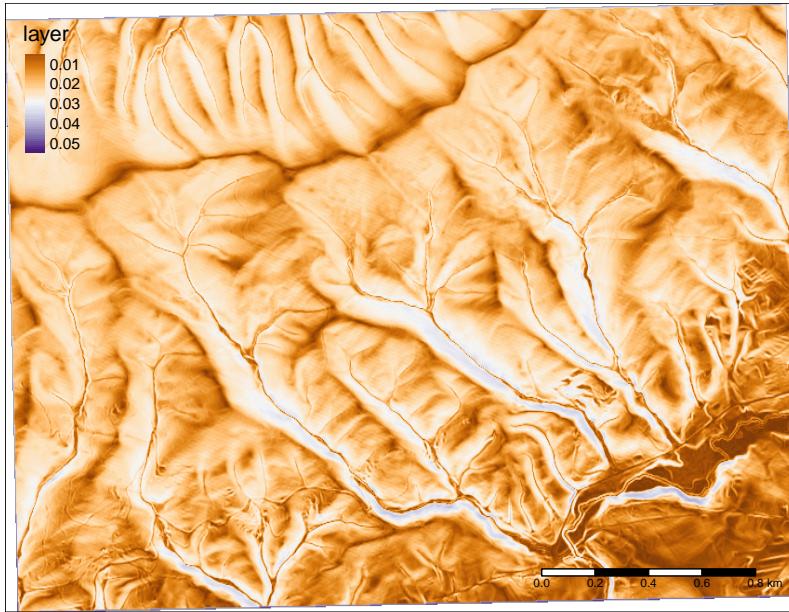
15.15 Raster Math

Super quick: you can also do math with your rasters!

If you've made model that predicts the likelihood of the location or something, you can just plug your rasters in like a normal equation and it'll do the math and you can map it! SO. COOL.

Here's a super simple example to just illustrate that you can do this: we will just divide slope by the elevation (dem)

```
demXslope <- slope / dem  
  
tm_shape(demXslope)+  
  tm_raster(style = "cont", palette = "PuOr", legend.show = TRUE)+  
  tm_scale_bar()
```



15.16 Extra: plot topo characteristics against one another

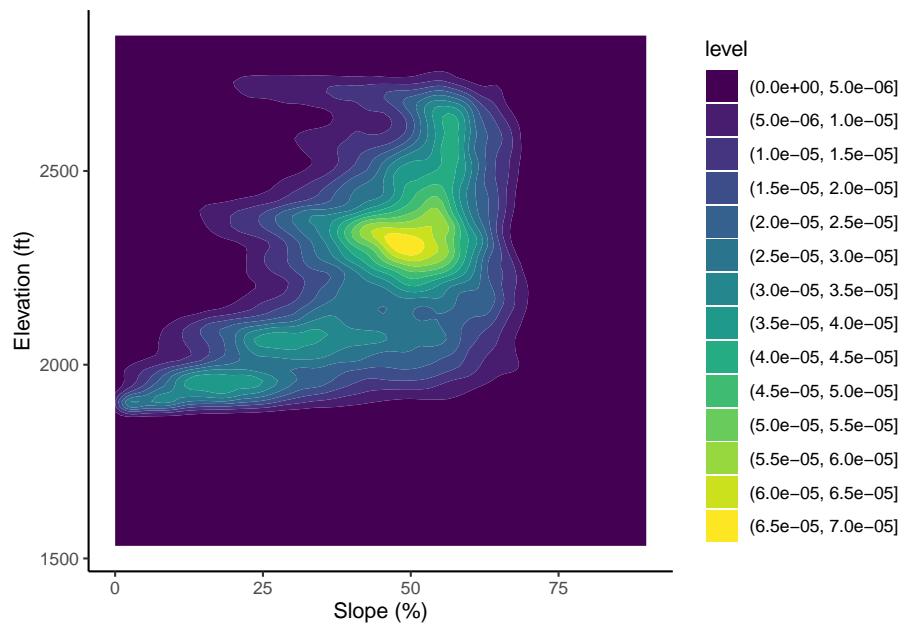
Here we convert slope and elevation each to a dataframe, join them, and then plot a 2 dimensional density plot.

```
slopedf <- as.data.frame(slope)
elevdf <- as.data.frame(dem)

slopeelev <- cbind(slopedf, elevdf)

ggplot(slopeelev, aes(x = demslope, y = brushDEMsm_5m))+
  geom_density_2d_filled()+
  ylab("Elevation (ft)")+
  xlab("Slope (%)")

## Warning: Removed 19206 rows containing non-finite values
## (stat_density2d_filled).
```



Chapter 16

Geospatial R Raster - Watershed Delineation

16.1 Introduction

The following activity is available as a template github repository at the following link: <https://github.com/VT-Hydroinformatics/15-Watershed-Delineation.git>

For more: <https://geocompr.robinlovelace.net/spatial-class.html#raster-data>

To read in detail about any of the WhiteboxTools used in this activity, check out the user manual: https://jblindsay.github.io/wbt_book/intro.html

This activity is adapted from/inspired by: <https://matthewrvross.com/active.html> and other code from Nate Jones.

Goals for this activity:

*Use R to delineate the watershed for several pour points on a DEM
Extract topographic information from those watersheds*

Check out the list of packages for this exercise. If you have trouble getting rayshader or rgl going, don't fret too much, these are for the 3d watershed visualization at the end that is basically just something fun to check out.

```
#install.packages("whitebox", repos="http://R-Forge.R-project.org")  
  
library(tidyverse)  
library(raster)
```

```

library(sf)
library(whitebox)
library(tmap)
library(stars)
library(rayshader)
library(rgl)

whitebox::wbt_init()

knitr::knit_hooks$set(webgl = hook_webgl)

theme_set(theme_classic())

```

16.2 The watershed delineation tool/process

With watershed delineation, it is helpful to take a step back and think about how the whole process works before we dive in. The way things are presented below is pretty linear, but there is no way you would arrive at it by just trying to fire up the watershed delineation function in whitebox. It took some planning to figure out this process! So let's work through it...

We will use the `wbt_watershed()` function to delineate our watersheds. This function looks upslope from a given point or points (the pour point(s)) and figures out what area drains to that point. To run this function you need a *D8 pointer grid* and *a pour point, or pour points*. Sounds simple. But we have to do a few things to generate those two inputs.

Generating the D8 pointer grid: this is a grid where each cell specifies what direction water will flow out of that cell. So if our DEM has any pits or depressions without outlets in it, there will be flow dead-ends. This will result in an incorrect watershed delineation. (This was covered in the last chapter) So we have to prepare our DEM to get rid of pits/depressions.

We will do this in two stages. First we will *fill single cell pits* and then *breach larger depressions*. It is important to remember that for this to work you have to fill the single cell depressions, then pass the *resulting DEM* from that process to the breach depression function. Once you have done this, you can run the D8 pointer function to generate your pointer grid.

Making pour points: The key here is that your pour points MUST.... *MUST* be on ones of the cells in the “stream”, according to your DEM. If it is even one cell over from the high flow accumulation cells that denote the stream on your DEM, you will just get a weird, small sliver of a watershed.

The first step is to get the coordinates of your pour point(s). You could grab them from google earth, use a GPS in the field, or maybe you have coordinates

for a known location/structure that defines your “outlet”. This could be a flume, weir, gaging station, etc. *AHEM EVEN IF* your coordinates came from the super fanciest GPS-iest post-processed differential nanometer accuracy wonder GPS, you **STILL** need to use the process described below to be sure they are on your flow network. Your points might be in the right place, but that doesn’t mean that’s where the stream is according to your DEM!

You can read your points in as a csv or just define them in your code if there aren’t many (that’s what we will do) and then turn them into a spatial object using `SpatialPoints()`, and then export them as a shapefile. If you are reading in a shapefile with your points, you don’t need to convert/export them.

With our points in hand, we then need to make sure they are on our stream network. To do that we will use Jensen snap pour points function in wbt. This tool takes your points, searches within a defined area for the stream network, then moves the points to the stream network. This function needs your *pour points* and a *raster stream file*. So.... we need to make the raster stream file! We do this using the extract streams wbt function. This function takes a D8 flow accumulation grid, so we need to create that first. Then we can run the stream network function and the snap pour points function.

A note on units: be sure you know what the distance units are in your data. In our case everything is in decimal degrees, so we need to specify how far the pour point function will search in degrees. If we were using UTM data, that number would be in meters, and if we were in state plane (in VA) that number would be in feet. You can 100% crash R and send wbt into a death spiral if you mix your units up and send the snap pour points function off searching for the biggest stream within 10,000 miles.

SO! Our process will look like this:

Read in DEM

Fill single cell sinks then breach larger sinks

Create D8 flow accumulation and D8 pointer grids

Read in pour points

Create stream raster

Snap pour points to stream raster

*Run watershed function

There will be some extra steps in there just to aid in visualization, but if you were just writing code to perform the analysis, the above is the ticket

16.3 Read in DEM

The first several steps are review from the previous activity.

First we will read in the raster, set its CRS (not always needed), set values below 1500 to NA since they are artefacts around the edges, and plot the DEM to be sure everything went ok.

```
tmap_mode("view")

## tmap mode set to interactive viewing

dem <- raster("McDonaldHollowDEM/brushDEMsm_5m.tif", crs = '+init=EPSG:4326')

writeRaster(dem, "McDonaldHollowDEM/brushDEMsm_5m_crs.tif", overwrite = TRUE)

dem[dem < 1500] <- NA

tm_shape(dem) +
  tm_raster(style = "cont", palette = "PuOr", legend.show = TRUE) +
  tm_scale_bar()
```

16.4 Generate a hillshade

Next, we will generate a hillshade to aid in visualization and then plot it to be sure it turned out ok.

```
wbt_hillshade(dem = "McDonaldHollowDEM/brushDEMsm_5m_crs.tif",
              output = "McDonaldHollowDEM/brush_hillshade.tif",
              azimuth = 115)

hillshade <- raster("McDonaldHollowDEM/brush_hillshade.tif")

tm_shape(hillshade) +
  tm_raster(style = "cont", palette = "-Greys", legend.show = FALSE) +
  tm_scale_bar()
```

16.5 Prepare DEM for Hydrology Analyses

This step is review from last time as well, but it is important to point out that it is crucial for this analysis. Basically we are looking upslope for all DEM cells that drain to a specific spot, so if there are any dead-ends, we will not get an accurate watershed.

In order to be sure all of our terrain drains downlope, fill single cell pits and then breach any other depressions using the `wbt_breach_depressions_least_cost()` function from `wbt`.

There is a much more in-depth discussion of why we are doing this in the previous chapter.

From now on in the analysis *be careful* to use the filled and breached DEM.

```
#wbt_fill_single_cell_pits()
#
#           dem = "McDonaldHollowDEM/brushDEMs_m_5m_crs.tif",
#           output = "McDonaldHollowDEM/bmstationdem_filled.tif")

wbt_breach_depressions_least_cost(
    dem = "McDonaldHollowDEM/brushDEMs_m_5m_crs.tif",
    output = "McDonaldHollowDEM/bmstationdem_breached.tif",
    dist = 5,
    fill = TRUE)

wbt_fill_depressions_wang_and_liu(
    dem = "McDonaldHollowDEM/bmstationdem_breached.tif",
    output = "McDonaldHollowDEM/bmstationdem_filled_breached.tif"
)
```

16.6 Create flow accumulation and pointer grids

The watershed delineation process requires a D8 flow accumulation grid and a D8 pointer file. There were both discussed last chapter. The flow accumulation grid is a raster where each cell is the area that drains to that cell, and the pointer file is a raster where each cell has a value that specifies which direction water would flow downhill away from that cell.

Below, create these two rasters using the *filled and breached* DEM.

```
wbt_d8_flow_accumulation(input = "McDonaldHollowDEM/bmstationdem_filled_breached.tif",
                           output = "McDonaldHollowDEM/D8FA.tif")

wbt_d8_pointer(dem = "McDonaldHollowDEM/bmstationdem_filled_breached.tif",
               output = "McDonaldHollowDEM/D8pointer.tif")
```

16.7 Setting pour points

The last thing we need is our pour points. These are the point locations for which we will delineate our watersheds. It is *crucial* that these points are on the stream network in each watershed. If the points are even one cell off to the side, you will not get a valid watershed. Instead you will end up with a tiny sliver that shows the area that drains to that one spot on the landscape.

170CHAPTER 16. GEOSPATIAL R RASTER - WATERSHED DELINEATION

Even with highly accurate GPS locations, we still need to check to be sure our pour points are on the stream network, because the DEM might not line up perfectly with the points.

Fortunately, there is a wbt function that will make sure our points are on the stream network. `wbt.jenson_snap_pour_points()` looks over a defined distance from the points you pass it for closest stream and then moves the points to those locations. So to use this function we also neet to create a stream network.

We will follow the following process to get our pour points set up:

- Create dataframe with pour points
- Convert data frame to shapefile
- Write the shapefile to our data directory
- Move points with snap pour points function

Perform the first two operations above in this chunk, the pour points are given. I just grabbed them from google earth.

```
ppoints <- tribble(  
  ~Lon, ~Lat,  
  -80.482778, 37.240504,  
  -80.474464, 37.242990,  
  -80.471506, 37.244512  
)  
  
ppointsSP <- SpatialPoints(ppoints, proj4string = CRS("+proj=longlat +datum=WGS84"))  
  
shapefile(ppointsSP, filename = "McDonaldHollowDEM/pourpoints.shp", overwrite = TRUE)
```

Now, following the process from last chapter, we will create a raster stream grid using a threshold flow accumulation of 6000 using the D8 flow accumulation grid.

Then finally, we will use the Jenson snap pour points function to move the pour points to their correct location.

The parameter `snap_dist` tells the function what distance in which to look for a stream. The units of the files we are using are decimal degrees, so we have to be careful here! Use a value of 0.0005, which is about 50 meters. If you were to put 50, it would search over 50 degrees of lat and lon!!! (I did this when making this activity and there was a lot of crashing)

After you get the streams and snapped pour points, read them into your R environment and plot them to be sure the pour points are on the streams.

```
wbt_extract_streams(flow_accum = "McDonaldHollowDEM/D8FA.tif",
                    output = "McDonaldHollowDEM/raster_streams.tif",
                    threshold = 6000)

wbt_jenson_snap_pour_points(pour_pts = "McDonaldHollowDEM/pourpoints.shp",
                            streams = "McDonaldHollowDEM/raster_streams.tif",
                            output = "McDonaldHollowDEM/snappedpp.shp",
                            snap_dist = 0.0005) #careful with this! Know the units of your data

pp <- shapefile("McDonaldHollowDEM/snappedpp.shp")
streams <- raster("McDonaldHollowDEM/raster_streams.tif")

tm_shape(streams)+
  tm_raster(legend.show = TRUE, palette = "Blues")+
tm_shape(pp)+
  tm_dots(col = "red")
```

16.8 Delineate watersheds

Now we are all set to delineate our watersheds!

Use wbt_watershed(), which takes as input a D8 pointer file (d8_pntr) and our snapped pour points (pour_pts). It will output a raster where each watershed is populated with a unique value and all other cells are NA.

Read the results of this function back in and plot them over the hillshade with alpha set to 0.5 to see what it did.

```
wbt_watershed(d8_pntr = "McDonaldHollowDEM/D8pointer.tif",
               pour_pts = "McDonaldHollowDEM/snappedpp.shp",
               output = "McDonaldHollowDEM/brush_watersheds.tif")

ws <- raster("McDonaldHollowDEM/brush_watersheds.tif")

tm_shape(hillshade)+
  tm_raster(style = "cont", palette = "-Greys", legend.show = FALSE)+
tm_shape(ws)+
  tm_raster(legend.show = TRUE, alpha = 0.5, style = "cat")+
tm_shape(pp)+
  tm_dots(col = "red")
```

16.9 Convert watersheds to shapefiles

For mapping or vector analysis it can be very useful to have your watersheds as polygons. To do this we will use the stars package. `st_as_stars()` converts our watershed raster into an object that the stars package can work with, and then `st_as_sf()` converts the raster stars object to a vector sf object. We also need to set `merge` to TRUE, which tells `st_as_sf` to treat each clump of cells with the same value (our watersheds) as its own feature.

Now we can plot the vector versions of our watersheds, and also use `filter()` to just show one at a time, or some combination, rather than all three.

```
wsshape <- st_as_stars(ws) %>% st_as_sf(merge = T)

ws1shp <- wsshape %>% filter(brush_watersheds == "1")

tm_shape(hillshade)+
  tm_raster(style = "cont", palette = "-Greys", legend.show = FALSE)+
  tm_shape(ws1shp)+
  tm_borders(col = "red")
```

16.10 Extract data based on watershed outline

Now, just like we looked at the distribution of different landscape data over an entire DEM in the last chapter, we can look at landscape data for each watershed. To do this we will use the `extract()` function to extract elevation data for just the watershed shapes (vector version). Then we will grab the data for each watershed, since the output here is a list, and plot them in separate geoms in ggplot.

Just like in last chapter you could do this for any of the topographic measures we calculated, including extracting multiple datasets and comparing them to one another. Cool!

```
wsElevs <- extract(dem, wsshape)

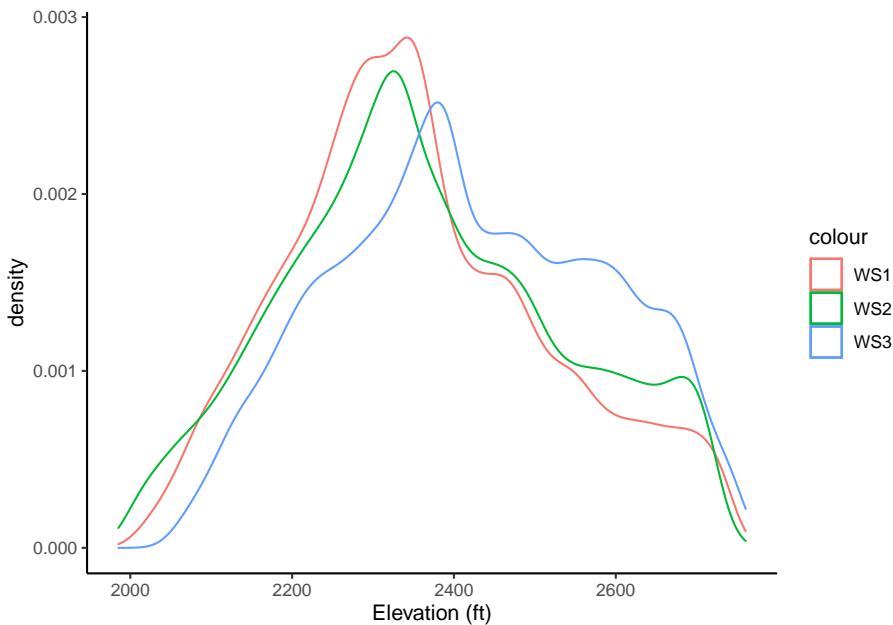
wsElevs1 <- setNames(wsElevs, c("WS1", "WS2", "WS3"))

WS1 <- as_tibble(wsElevs1$WS1)
WS2 <- as_tibble(wsElevs1$WS2)
WS3 <- as_tibble(wsElevs1$WS3)

ggplot() +
  geom_density(WS1, mapping = aes(value, color = "WS1"))+
```

16.11. BONUS: MAKE A 3D MAP OF YOUR WATERSHED WITH RAYSHADER173

```
geom_density(WS2, mapping = aes(value, color = "WS2"))+
  geom_density(WS3, mapping = aes(value, color = "WS3"))+
  xlab("Elevation (ft)")
```



```
#wsElevs1 %>% map_dfr(~as_tibble(.) %>% mutate(WS = names(.)))
```

16.11 BONUS: Make a 3d map of your watershed with rayshader

The following code is here just because it is cool. We clip the DEM to the watershed we want, convert it to a matrix, create a hillshade using rayshader (a visualization tool for 3d stuff), and then plot the output.

```
ws1_bound <- filter(wsshape, brush_watersheds == "1")

#crop
wsmask <- dem %>%
  crop(., ws1_bound) %>%
  mask(., ws1_bound)

#convert to matrix
```

174 CHAPTER 16. GEOSPATIAL R RASTER - WATERSHED DELINEATION

```
wsmat <- matrix(  
  extract(wsmask, extent(wsmask)),  
  nrow = ncol(wsmask),  
  ncol = nrow(wsmask))  
  
#create hillshade  
raymat <- ray_shade(wsmat, sunangle = 115)  
  
#render  
wsmat %>%  
  sphere_shade(texture = "desert") %>%  
  #add_shadow(raymat) %>%  
  plot_3d(wsmat, zscale = 10, fov = 0, theta = 135, zoom = 0.75, phi = 45,  
  windowsize = c(750,750))
```

```
## Warning in make_shadow(heightmap, shadowdepth, shadowwidth, background, :  
## `magick` package required for smooth shadow--using basic shadow instead.
```

```
#render as html  
rglwidget()
```

```
## Warning in snapshot3d(scene = x, width = width, height = height): webshot = TRUE  
## requires the webshot2 package and Chrome browser; using rgl.snapshot() instead
```

16.11. BONUS: MAKE A 3D MAP OF YOUR WATERSHED WITH RAYSHADER175



Chapter 17

Intro to Modeling - Getting Started with HBV

17.1 Introduction

The following activity is available as a template github repository at the following link: <https://github.com/VT-Hydroinformatics/16-Intro-Modeling-HBV>

Before we start with the code below, we will briefly discuss modeling in general, using the following reading as a guide: https://cfpub.epa.gov/si/si_public_record_report.cfm?dirEntryId=339328&Lab=NERL

Please be prepared to discuss the reading at the start of class.

Goals for this activity:

- Become familiar with the process of running a simple hydrologic model
- Try parameterizing the model manually
- Introduction to methods of assessing model success

```
library(tidyverse)
library(lubridate)
library(patchwork)
library(plotly)

theme_set(theme_classic())
```

17.2 Creating the HBV model function

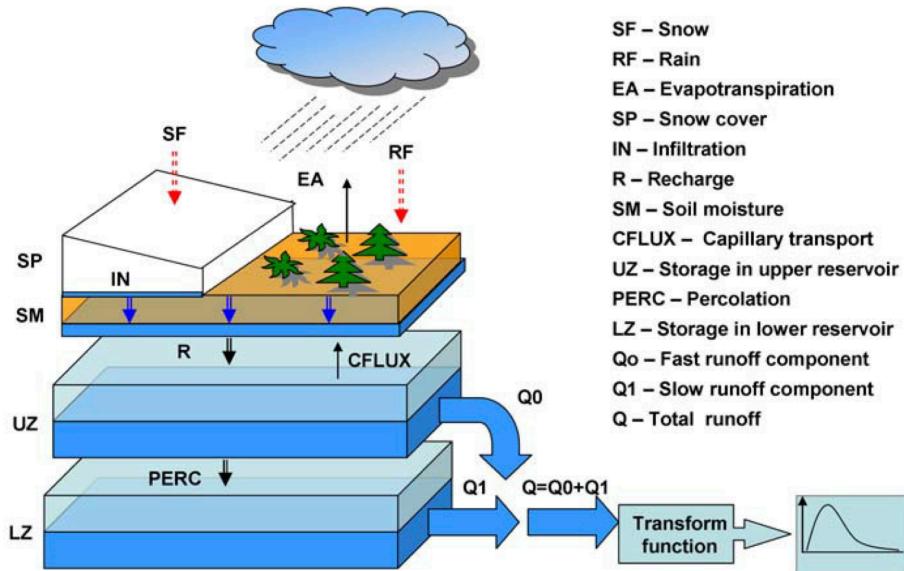
The code for the HBV model is in the HBV folder, it is called *HBV.R*. We will open that and look at the code a bit to discuss how HBV works.

The code in *HBV.R* creates a function that runs the HBV model. We can run that code by using the `source()` function in. When we give the location of the *HBV.R* file to `source()`, it runs the code and creates the HBV function. Run the line of code below, then type `HBV(` and you'll see the input parameters pop up just like any other function.

From this we see that HBV takes as input pars, P, Temp, PET, and routing.

`pars` is a vector of all the parameters that HBV uses to run (discussed below) `P` is a vector of precipitation values `Temp` is a vector of temperature values `PET` is a vector of PET values And `routing` is a parameter that controls how water is routes through the channel of larger watersheds. We will not use this because we are running the model on a small watershed.

HBV is a lumped conceptual model. We will use the diagram below and the *HBV.R* code to discuss how the model works and get an idea of what the parameters do.



This schematic was reproduced from: Durga Lal Shrestha & Dimitri P. Solomatine (2008) Data-driven approaches for estimating uncertainty in rainfall-runoff modelling, International Journal of River Basin Management, 6:2, 109-122, DOI: 10.1080/15715124.2008.9635341

```
source('HBV/HBV.R')
```

17.3 Read in Precip and Temp

Our objective today is basically to get the model running with data from watershed 3 at the Hubbard Brook experimental forest in NH, USA.

Watershed 3 is a hydrologic reference watershed at HBEF. This means it hasn't been experimentally manipulated, but discharge, temperature, and precipitation have been recorded there for a long time. This makes it a good candidate for some modeling! You can see more about watershed 3 here: <https://hubbardbrook.org/watersheds/watershed-3>

As we saw in the chunk above, we need P, Temp, PET, and a set of parameters to run the model.

We will talk parameters later, but for now we need the data that drives the model: P, Temp, and PET.

We have P and Temp. The data is in Pwd2009-2012.csv and Tdm2009-2012.csv.

So how do we get PET?

We calculate it! But before we do that let's bring in the precip and temp data and format them how the model wants them.

Our model function just wants a vector of values. No dates attached. So we must be careful that each set of input data is for the same amount of time with the same number of values. To help with that, we will start below by defining and start and end date for the model run.

Finally, we will read in the data, select the gage or station we want to represent the watershed, change the format of the date column, and then filter to our start and end date.

Then we will pull out just the data, so it can be passed to the model.

```
start <- mdy("01-01-2009")
end <- mdy("12-31-2012")

#Precip
P1 <- read_csv("HBV/Pwd2009-2012.csv") %>%
  select(DATE, WS_3) %>%
  mutate(DATE = ymd(DATE)) %>%
  filter(DATE >= start & DATE <= end)

## Rows: 1461 Columns: 10
## -- Column specification -----
```

```

## Delimiter: ","
## dbl (10): DATE, WS_1, WS_2, WS_3, WS_4, WS_5, WS_6, WS_7, WS_8, WS_9
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

P <- P1$WS_3

#Temp in deg C
Temp1 <- read_csv("HBV/Tdm2009-2012.csv")%>%
  select(DATE, STA_1) %>%
  mutate(DATE = ymd(DATE)) %>%
  filter(DATE >= start & DATE <= end)

## Rows: 1461 Columns: 9
## -- Column specification -----
## Delimiter: ","
## dbl (9): DATE, STA_1, STA_6, STA_14, STA_INT, STA_HQ, STA_23, STA_17, STA_24
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

Temp <- Temp1$STA_1

```

17.4 Calculate PET

Next, we need to calculate PET for the same time reange as our temperature and precipitation record.

We will use the Hamon method to calculate PET for each day of the record.

To do that we will take the latitude of the site in radians...

```

lat <- 43 + 57/60 #43 degrees and 57 minutes
latrad <- (lat/360) * 2 * pi #convert to radians

PET1 <- select(Temp1, DATE) %>%
  mutate(DOY = yday(DATE)) %>% #DOY for dates
  mutate(tempvar = (2 * pi / 365) * DOY) %>%
  #declination of the sun above the celestial equator in
  #radians on day JulDay of the year
  mutate(delta_h = 0.4093 * sin(tempvar - 1.405)) %>%
  #day length in h
  mutate(daylen = (2 * acos(-tan(delta_h) * tan(latrad)) / 0.2618)) %>%

```

```

  mutate(
    PET = 29.8 * daylen * 0.611 * exp(17.3 * Temp /
      (Temp + 237.3)) / (Temp + 273.2)) #PET Hamon method
  PET <- PET1$PET

```

17.5 HBV Parameters

Okay, so now we have our data set up. Next we need to talk parameters.

The table below shows the parameters of the HBV model, the minimum and maximum for those values, and a description of their function in the model.

Object Name	Min	Max	Description
FC	40	400	Max soil moisture storage, field capacity
beta	1	6	Shape coefficient governing fate of water input to soil moisture storage
LP	.3	1	Threshold for reduction of evaporation
SFCF	0.4	1.2	Snowfall correction factor
TT	-1.5	1.2	Threshold temperature
CFMAX	1	8	Degree-day factor
k0	0.05	0.5	Recession constant (upper storage, near surface)
k1	0.01	0.3	Recession constant (upper storage)
k2	0.001	0.15	Recession constant (lower storage)
UZL	0	70	Threshold for shallow storage
PERC	0	4	Percolation, max flow from upper to lower storage
MAXBAS	1	3	base of the triangular routing function, days"

To pass a set of parameters to the model, we just put them into a single vector in the order they are in the table above. In the chunk below, I've structured

this to make it easy to see a description of each parameter, but you could also just do it in one line without all the comments. Just be sure to get the order right!

The code below sets the routing to 0, which is what we want for a small catchment.

Then I just set each parameter to it's minimum value. We will use this parameter set to run the model and see what happens!

```

#when this term = 1, then triangular routing is invoked, or for no routing, routing = 0
#if routing = 0 then MAXBAS doesn't do anything
routing <- 0

#hard code parameters
params <- c(40,      #FCM ax soil moisture storage, field capacity
           1,       #beta Shape coefficient governing fate of water input to soil moisture
           0.3,    #LP Threshold for reduction of evap
           0.4,    #SFCF Snowfall correction factor
           -1.5,   #TT Threshold temperature
           1,       #CFMAX Degree-day factor
           0.05,   #k0 Recession constant (upper storage, near surface)
           0.01,   #k1 Recession constant (upper storage)
           0.001,  #k2 Recession constant (lower storage)
           0,       #UZLThreshold for shallow storage
           0,       #PERC Percolation, max flow from upper to lower storage
           1       #MAXBAS base of the triangular routing function, days
         )

```

17.6 First model run

Let's run the model! Remember the function is HBV(parameters, Precip, Temp, PET, routing)

We will set the output of the model equal to `ModelOutput` and then have a look at what it outputs.

How can we tell how this did at modeling flow in watershed 3?

```
ModelOutput <- HBV(params, P, Temp, PET, routing)
```

```
head(ModelOutput)
```

```
## # A tibble: 6 x 12
##       q   qs   qi     qb Storage    SWE    AET      SF      S1      S2 soil      w
```

```

##   <dbl> <dbl> <dbl> <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1    0    0    0    0      0    0    0    0    0    0    0    0    0
## 2    0    0    0    0     40  152.    0    0    0    0    40    0
## 3    0    0    0    0     40  152.    0    0    0    0    40    0
## 4    0    0    0    0     40  152.    0    0    0    0    40    0
## 5    0    0    0    0     40  154.    0  1.16    0    0    40    0
## 6    0    0    0    0     40  154.    0    0    0    0    40    0

```

17.7 Import observed streamflow data

`q` in the model output is discharge, and we also have a record of measured discharge from watershed 3 in the file called `SWD2009-2012.csv`

In order to compare the modeled discharge to the measured discharge, we will read in the data and process it the same way we did with the model inputs. Then we can attach it to the model output so we can compare the two.

NOTE: These data are all nicely processed. They have the same temporal resolution (daily), there are no NAs, and they are in the same units. These are all things to check if you are preparing data from another site.

```

#Streamflow mm/d
Qobs1 <- read_csv("HBV/SWD2009-2012.csv") %>%
  select(DATE, WS_3) %>%
  mutate(DATE = ymd(DATE)) %>%
  filter(DATE >= start & DATE <= end)

## Rows: 1461 Columns: 10
## -- Column specification -----
## Delimiter: ","
## dbl (10): DATE, WS_1, WS_2, WS_3, WS_4, WS_5, WS_6, WS_7, WS_8, WS_9
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

Qobs <- Qobs1$WS_3

ModelOutput <- bind_cols(ModelOutput, Qobs1)

```

17.8 Compare observed and modeled discharge graphically

Now that we have the observed values glued on to our model output, we can plot the two and see how they compare. Below we will plot the modelled and observed discharge as two lines on a ggplot.

Because the model starts out with its storages at zero, it takes some time for it to actually approximate real conditions. For this reason, you will typically set aside part of the modelled period as a “warm up” period. The duration of this period depends on the system, but for our purposes, we are doing to drop the first half of the model run and only look at the second half: 2011 - 2013. So we will filter the data to just look at that time.

How'd we do? Did the model do a good job of capturing runoff dynamics in watershed 3?

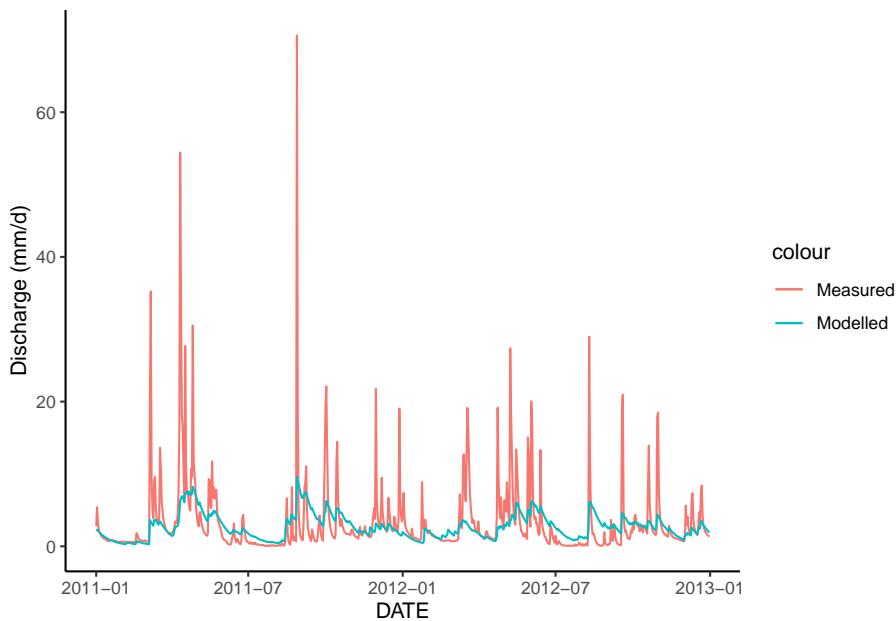
Describe what it looks like the model did well and what it did poorly?

When you are looking at results like this, it is often helpful to look at the full timeframe but to also zoom in and look at specific event dynamics to see what hydrologic processes the model is or is not capturing well. One helpful tool at doing that is plotly. Plotly allows you to create interactive plots. The code chunk after the ggplot creates a plotly graph. From here on out we will use plotly to look at our modeling results so we can investigate them more thoroughly.

```
OutputTrim <- filter(ModelOutput, DATE >= mdy("01-01-2011"))

ggplot(OutputTrim, aes(x = DATE, y = WS_3, color = "Measured"))+
  geom_line()+
  geom_line(aes(y = q, color = "Modelled"))+
  ylab("Discharge (mm/d)")
```

17.9. COMPARE OBSERVED AND MODELLED DISCHARGE WITH INTERACTIVE GRAPH185



17.9 Compare observed and modelled discharge with interactive graph

The code below creates a plot analogous to the ggplot above. Note the differences in syntax. All the same information is there, it is just a different recipe, and you use pipes instead of +.

You can click and drag a box over any time you want to look at to zoom, and if you hover over one of the lines, it will show you the values in a dialogue box. There is a lot of other functionality as well, which you can explore in the menu in the upper right.

```
OutputTrim %>% plot_ly(x = ~DATE) %>%
  add_trace(y = ~q, name = 'Modeled', type = 'scatter', mode = 'lines') %>%
  add_trace(y = ~WS_3, name = 'Measured', type = 'scatter', mode = 'lines')
```

17.10 Measure how well the model fits with NSE

Before we can start trying to tune our model to look more like the observed discharge record, it would be helpful to have some sort of quantified metric for how well our modeled data fits the measured data.

There are many different ways to do this, but discussion of the pros and cons of those approaches is beyond this quick introduction to modeling. So we are just going to look at one method, the Nash-Sutcliffe Efficiency (NSE).

Basically, the NSE looks at how much better your model run did than if you had just used the mean discharge for the data record as your “modelled results”. It does this by comparing how far off the observed values were from the mean discharge to how far off the modeled values were from the observed discharge.

Mathematically NSE is the sum of the squared differences between the modeled and observed discharge divided by the sum of the squared differences between the observed and mean discharge, subtracted by 1.

$$NSE = 1 - \frac{\sum_{t=1}^T (Q_m^t - Q_o^t)^2}{\sum_{t=1}^T (Q_o^t - \bar{Q}_o)^2}$$

Where Q_m^t is modeled discharge at time t, Q_o^t is observed discharge at time t, and \bar{Q}_o is mean observed discharge.

Below, we calculate NSE for the model run above. We will continue to exclude the warm up period.

An NSE over 0 means the model did better than the mean discharge at predicting discharge. An NSE of 1 would be a perfect model fit. How'd we do? Does this make sense with the timeseries we looked at above?

```
#Calculate NSE
NSE <- 1 - ((sum((OutputTrim$q - OutputTrim$WS_3) ^ 2)) /
               sum((OutputTrim$WS_3 - mean(OutputTrim$WS_3)) ^ 2))

NSE
## [1] 0.2135378
```

17.11 Assess model fit with a different measure: Snow

Before we jump into trying to parameterize this thing, I wanted to note that there are other things you can calibrate your model to other than discharge. In the case of watershed 3, for example, we also have a snow record. Snowmelt is a very important input driving our model so we might want to make sure that the model is doing a good job of capturing that as well!

Let's pull in snow, calculate NSE between the modeled and measured snow, and look at a plot of the data.

17.11. ASSESS MODEL FIT WITH A DIFFERENT MEASURE: SNOW187

Snow is not measured daily, so we will plot the measured values as points. Likewise, we will have filter our data to only times when we have a modeled snow amount AND a recorded snow amount when we calculate NSE.

```
#Read and prep snow data
snow <- read_csv("HBV/sno2009-2012.csv") %>%
  select(DATE, STA2) %>%
  mutate(DATE = ymd(DATE)) %>%
  filter(DATE >= start & DATE <= end)

## New names:
## Rows: 76 Columns: 25
## -- Column specification
## -----
## (1): DATE dbl (21): STA1, STA2, STA3, STA4, STA5, STA6, STA7, STA8, STA9,
## STA10, STA11... lgl (3): ...23, ...24, ...25
## i Use `spec()` to retrieve the full column specification for this data. i
## Specify the column types or set `show_col_types = FALSE` to quiet this message.
## * `` -> `...23`
## * `` -> `...24`
## * `` -> `...25`

## Warning: 8 failed to parse.

#Join measured snow to model output
OutputTrimSno <- left_join(OutputTrim, snow, by = "DATE")

#Plot modeled and measured snow
OutputTrimSno %>% plot_ly(x = ~DATE) %>%
  add_trace(y = ~SWE, name = 'Modeled', type = 'scatter', mode = 'lines') %>%
  add_trace(y = ~STA2, name = 'Measured', type = 'scatter', mode = 'markers')

## Warning: Ignoring 698 observations

#Drop rows where there isn't a measured snow amount
CompareSnow <- drop_na(OutputTrimSno, STA2)

#Calculate NSE for snow, SWE is modeled, STA2 is measured
NSEsno <- 1 - ((sum((CompareSnow$SWE - CompareSnow$STA2) ^ 2)) /
  sum((CompareSnow$STA2 - mean(CompareSnow$STA2)) ^ 2))

NSEsno

## [1] 0.1339054
```

17.12 Calibrate HBV manually

Woohoo! We can now run our model and assess how well it is working!

Now, let's see how well we can get it to work. The code below runs the model, produces a plot, and calculates the NSE based on discharge.

By changing the parameters, see how well you can get the model to fit. Take note of how the discharge changes when you change certain parameters. Let's see who can get the highest NSE!

```
#when this term = 1, then triangular routing is invoked, or for no routing, routing = 0
#if routing = 0 then MAXBAS doesn't do anything
routing <- 0

#hard code parameters
params <- c(40,      #FCM ax soil moisture storage, field capacity
          1,       #beta Shape coefficient governing fate of water input to soil mois
          0.3,    #LP Threshold for reduction of evap
          0.4,    #SFCF Snowfall correction factor
          -1.5,   #TT Threshold temperature
          1,       #CFMAX Degree-day factor
          0.05,   #k0 Recession constant (upper storage, near surface)
          0.01,   #k1 Recession constant (upper storage)
          0.001,  #k2 Recession constant (lower storage)
          0,       #UZL Threshold for shallow storage
          0,       #PERC Percolation, max flow from upper to lower storage
          1        #MAXBAS base of the triangular routing function, days
          )

#Run the model
Out <- HBV(params, P, Temp, PET, routing)

#Add observed output
Out <- bind_cols(Out, Qobs1)

#Trim out the warm up period
OutTrim <- filter(Out, DATE >= mdy("01-01-2011"))

#Calculate NSE
NSE <- 1 - ((sum((OutTrim$q - OutTrim$WS_3) ^ 2)) /
              sum((OutTrim$WS_3 - mean(OutTrim$WS_3)) ^ 2))

#Create plot with NSE in title
OutTrim %>% plot_ly(x = ~DATE) %>%
  add_trace(y = ~q, name = 'Modeled', type = 'scatter', mode = 'lines') %>%
```

```
add_trace(y = ~WS_3, name = 'Measured', type = 'scatter', mode = 'lines') %>%  
layout(title=paste("NSE: ", round(NSE,2)))
```


Chapter 18

Intro to Modeling - Calibrate HBV

18.1 Introduction

The following activity is available as a template github repository at the following link: <https://github.com/VT-Hydroinformatics/17-calibrate-model-for-loops>

Goals for this activity:

- Learn how for loops work
- Use for loops to calibrate the HBV model using the Monte Carlo technique
- Assess the results of the calibration, discuss equifinality

We saw in the last activity that coming up with the parameters that create a model that fits our watershed discharge best is hard. For this reason, there are several ways people have devised for automating the process. We are going to explore one: a Monte Carlo calibration.

To use this approach, we will create a ton of random parameter sets, run the model for each, and then see which one fits best. The idea is that by creating a bunch of random sets of parameters, we will end up with at least one that works well. As you might guess, this can take a while. We will just do this for 100 or 1000 runs here in this activity, but we will also explore some output from a 100,000 run Monte Carlo that I ran and saved.

There is one new package this time, GGally, just for creating a parallel coordinate plot to look at the results of the 100k run Monte Carlo at the end.

```
library(tidyverse)
library(lubridate)
library(plotly)
library(GGally) #for making parallel coordinate plot

theme_set(theme_classic())
```

In order to do what we are going to do, we need “for” loops.

A for loop is a way to run a chunk of code several times while changing a value in that chunk of code each time.

So using an example from class: you could run some code that downloads data from a USGS gage, and each time through the loop it changes which gage you download.

In R, the syntax for this is

```
for(x in sequence){
  x
}
```

Where x will change every time you go through the loop. The values x will have each time through the loop are those that are in “sequence” above. If you changed this to x in 1:3, x would have values of 1, 2, and 3 each time through. If you made it x in c(2,3,5) x would change to 2, then 3, and then 5 each time through the loop. You can make the sequence anything and x can have any name so “letter in c(a, b, c)” would have *letter* change to a, then b, then c each time through the loop.

The image below shows a short example of how a for loop works.

```
days <- c("mon", "tues", "wednes")
for(x in 1:length(days)){
  days[x] <- paste0(days[x], "day")
}
```

Run the code between the {} 3 times, changing the value of x each time to each of the values in c(1, 2, 3)

Loop 1 | **x = 1, so:**
days[1] <- paste0(days[1], "day") → days = "monday"

Loop 2 | **x = 2, so:**
days[2] <- paste0(days[2], "day") → days = "monday",
"tuesday"

Loop 3 | **x = 3, so:**
days[3] <- paste0(days[3], "day") → days = "monday",
"tuesday", "wednesday"

Let's try some examples

```
val <- 0

for(x in 1:10){
  val[x] <- x
}

val

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

days <- c("mon", "tues", "wednes")

for(x in 1:length(days)){
  days[x] <- paste0(days[x], "day")
}

days

## [1] "monday"    "tuesday"   "wednesday"

gages <- c("123", "ABC", "001")

for(g in gages){
```

```

    print(g)
}

```

```

## [1] "123"
## [1] "ABC"
## [1] "001"

```

18.2 Challenge: Write a for loop

Define a vector with values 5, 6, 7, 8, 9

Create a for loop that steps through this vector and adds 1 to each value.

Yes, you can do this by just typing the name of the vector + 1, but that's not the point!

18.3 Prep data for HBV

Since we did this in the last chapter, we are just going to quickly run the code the preps our data to go into the model. Below we load the HBV model and make vectors of P, Qobs, and Temp.

```

#Create the HBV function with the HBV R file
source('HBV/HBV.R')

#set start and end dates for model to filter data
start <- mdy("01-01-2009")
end <- mdy("12-31-2012")

#Precip
P1 <- read_csv("HBV/Pwd2009-2012.csv") %>%
  select(DATE, WS_3) %>%
  mutate(DATE = ymd(DATE)) %>%
  filter(DATE >= start & DATE <= end)

## Rows: 1461 Columns: 10
## -- Column specification -----
## Delimiter: ","
## dbl (10): DATE, WS_1, WS_2, WS_3, WS_4, WS_5, WS_6, WS_7, WS_8, WS_9
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

```

```

P <- P1$WS_3

#Streamflow mm/d
Qobs1 <- read_csv("HBV/SWD2009-2012.csv") %>%
  select(DATE, WS_3) %>%
  mutate(DATE = ymd(DATE)) %>%
  filter(DATE >= start & DATE <= end)

## Rows: 1461 Columns: 10
## -- Column specification -----
## Delimiter: ","
## dbl (10): DATE, WS_1, WS_2, WS_3, WS_4, WS_5, WS_6, WS_7, WS_8, WS_9
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

Qobs <- Qobs1$WS_3

#Temp in deg C
Temp1 <- read_csv("HBV/Tdm2009-2012.csv")%>%
  select(DATE, STA_1) %>%
  mutate(DATE = ymd(DATE)) %>%
  filter(DATE >= start & DATE <= end)

## Rows: 1461 Columns: 9
## -- Column specification -----
## Delimiter: ","
## dbl (9): DATE, STA_1, STA_6, STA_14, STA_INT, STA_HQ, STA_23, STA_17, STA_24
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

Temp <- Temp1$STA_1

#Snow in SWE, mm
snow <- read_csv("HBV/sno2009-2012.csv") %>%
  select(DATE, STA2) %>%
  mutate(DATE = ymd(DATE)) %>%
  filter(DATE >= start & DATE <= end)

## New names:
## Rows: 76 Columns: 25
## -- Column specification

```

```

## ----- Delimiter: "," chr
## (1): DATE dbl (21): STA1, STA2, STA3, STA4, STA5, STA6, STA7, STA8, STA9,
## STA10, STA11... lgl (3): ...23, ...24, ...25
## i Use `spec()` to retrieve the full column specification for this data. i
## Specify the column types or set `show_col_types = FALSE` to quiet this message.
## * `` -> `...23`
## * `` -> `...24`
## * `` -> `...25`

## Warning: 8 failed to parse.

```

18.4 Calculate PET

Again, because we did this last time, we will just re-run this code to create our PET values.

```

lat <- 43 + 57/60 #43 degrees and 57 minutes
latrad <- (lat/360) * 2 * pi #convert to radians

PET1 <- select(Temp1, DATE) %>%
  mutate(DOY = yday(DATE)) %>% #DOY for dates
  mutate(tempvar = (2 * pi / 365) * DOY) %>%
    #declination of the sun above the celestial equator in
    #radians on day JulDay of the year
  mutate(delta_h = 0.4093 * sin(tempvar - 1.405)) %>%
    #day length in h
  mutate(daylen = (2 * acos(-tan(delta_h) * tan(latrad)) / 0.2618)) %>%
  mutate(
    PET = 29.8 * daylen * 0.611 * exp(17.3 * Temp /
      (Temp + 237.3)) / (Temp + 273.2)) #PET Hamon method

PET <- PET1$PET

```

18.5 Monte Carlo step 1: generate random parameter sets

Below is a table of the parameters for the HBV model and their suggested min and max values for our application. What we basically want to do is mimic you sitting down at the computer and randomly changing all the parameters, running the model, and checking to see how well it did. Over, and over, and over.

18.5. MONTE CARLO STEP 1: GENERATE RANDOM PARAMETER SETS 197

You can do this by generating the set of parameters each time you run the model, but we are going to do it separately so we can see the Nashe-Sutcliff efficiency (NSE) for each set of parameters.

Object Name	Min	Max	Description
FC	40	400	Max soil moisture storage, field capacity
beta	1	6	Shape coefficient governing fate of water input to soil moisture storage
LP	.3	1	Threshold for reduction of evaporation
SFCF	0.4	1.2	Snowfall correction factor
TT	-1.5	1.2	Threshold temperature
CFMAX	1	8	Degree-day factor
k0	0.05	0.5	Recession constant (upper storage, near surface)
k1	0.01	0.3	Recession constant (upper storage)
k2	0.001	0.15	Recession constant (lower storage)
UZL	0	70	Threshold for shallow storage
PERC	0	4	Percolation, max flow from upper to lower storage
MAXBAS	1	3	base of the triangular routing function, days”

To make our parameter sets, we first have to decide how many model runs we want to try. Let's try 100 so we don't burn down anyone's computer.

Then we will use the `runif()` function to generate the same number of random values as the number of model runs we want to perform for each parameter, within the bounds of the min and max values above.

Remember, `runif()` creates a set of random numbers in a uniform distribution with a specified length with a specified minimum and maximum value. A uniform distribution is one where each number has the same probability of occurrence as the next, so it's basically a set of random numbers between your min and max values.

We will also create a blank vector of the same length which we will populate

with the NSE values for each model run, and then slap them all together with cbind() (column bind).

```
#number of runs
N <- 100

# PARAMETERS RANGE and generate set
FC <- runif(N, min = 40 , max = 400) #Max soil moisture storage, field capacity
beta <- runif(N, min = 1 , max = 6) #Shape coefficient governing fate of water
LP <- runif(N, min = 0.3 , max = 1) #Threshold for reduction of evap
SFCF <- runif(N, min = 0.4 , max = 1.2) #Snowfall correction factor
TT <- runif(N, min = -1.5 , max = 1.2) #Threshold temperature
CFMAX <- runif(N, min = 1 , max = 8) #Degree-day factor
k0 <- runif(N, min = 0.05 , max = 0.5) #Recession constant (upper storage, near surface)
k1 <- runif(N, min = 0.01 , max = 0.3) #Recession constant (upper storage)
k2 <- runif(N, min = 0.001, max = 0.15) #Recession constant (lower storage)
UZL <- runif(N, min = 0 , max = 70) #Threshold for shallow storage
PERC <- runif(N, min = 0 , max = 4) #Percolation, max flow from upper to lower
MAXBAS<- rep(1, N) #base of the triangular routing function, days
#MAXBAS is just 1's because routing will be set to zero, so the parameter isn't used

NSE <- rep(NA, N) #create NSE column, to be filled in for loop

routing <- 0

pars <- cbind(FC, beta, LP, SFCF,
              TT, CFMAX, k0, k1,
              k2, UZL, PERC, MAXBAS, NSE)
```

18.6 Run the model for each parameter set

Alright, let's get at this. We now have our random sets of parameters. So we will run the model for each and see how it does.

Here's how we will do that:

First: Define the period for which we want to calculate NSE. Remember we are going to drop the first 40% of the model run to let the model "warm up". Since we are starting with nothing in the storages we need to let the boxes "fill up" and settle in.

Then we run a FOR LOOP!

Our for loop will step through a sequence of numbers from 1 to the total number of parameter sets we have. So in this case: 100. We will use i as our variable. So i will be 1, 2, 3.... all the way to 100.

In the code portion of the loop, it will grab the parameters from row i of the parameter set we created. And pass that to the HBV function with the Precip, Temp, and PET data. Remember routing is just set to zero.

After we run the model (still in the for loop) we will stick our observed values onto the results and then trim them to the period we want to evaluate. (drop the first 40%)

Finally, we calculate NSE for that individual model run, and then add it into the NSE column we made before. So each run will just tack on an NSE value to the end of the parameter set. We don't need to save the model results, because we can just rerun the model with the same parameters and it'll give us the same output.

The figure below illustrates how the for loop will work:

i = 1 Model pars: pars[i, 1:3]
NSE = pars[i, 4]

pars			
par1	par2	par3	NSE
1	198	0.03	0.5
1.3	230	0.02	NA
1.1	175	0.04	NA

i = 2 Model pars: pars[i, 1:3]
NSE = pars[i, 4]

par1	par2	par3	NSE
1	198	0.03	0.5
1.3	230	0.02	0.3
1.1	175	0.04	NA

i = 3 Model pars: pars[i, 1:3]
NSE = pars[i, 4]

par1	par2	par3	NSE
1	198	0.03	0.5
1.3	230	0.02	0.3
1.1	175	0.04	0.7

Figure 18.1: monte carlo for loop

```

#trim the first 40% (warm up) of Qobs off for NSE calculation
EvalStart <- floor(length(Qobs) * 0.4)
EvalEnd <- length(Qobs)

for (i in 1:N){

  #call model with i parameter set generated above
  results <- HBV(pars[i,1:12], P, Temp, PET, routing)

  #add the Qobs to results
  results <- cbind(results, Qobs)

  #trim the first 40% of the record so it isn't included in the NSE calculation
  results <- results[EvalStart:EvalEnd,]

  #Calculate NSE and add to parameter set
  pars[i,13] <- 1 - ((sum((results$Qobs - results$q) ^ 2)) / sum((results$Qobs - mean(results$Qobs))^2))
}

}

```

18.7 Find the best parameter set

Alright, now we have a NSE value that tells us how well each parameter set did at estimating runoff. So we need to find the best one!

To do this, we will convert the parameter sets to a tibble, use filter to grab the row with the maximum NSE, then use slice to pull just that row out of the tibble. Finally, we will run as.numeric on the data to be sure everything is numbers.

Now we have our best parameter set, let's see how it did!

We will do what we did in the previous lesson: run the model with the parameter set, slap on the observed data, and filter it to the time period we want to look at. (after 08-07-2010 in this case, that's the 40% cutoff in this case)

```

#find best parameters
pars <- as_tibble(pars)

bestparams <- pars %>% filter(NSE == max(NSE)) %>%
  slice(1) %>%
  as.numeric()

#run with best parameters

```

```

modeloutput <- HBV(bestparams, P, Temp, PET, routing)

#add observations for plotting
modeloutput <- bind_cols(modeloutput, Qobs1)

#trim out warm up period for plotting
OutputTrim <- filter(modeloutput, DATE >= mdy("08-07-2010"))

```

Below we will look at the model run using `plotly`. First, we calculate the NSE again, just to be sure, and then plot the same way we did in the last lesson.

How did the model do? Did the 100 run Monte Carlo do better or worse than your manual attempts from last lesson?

```

#Calculate NSE
NSE <- 1 - ((sum((OutputTrim$q - OutputTrim$WS_3) ^ 2)) /
               sum((OutputTrim$WS_3 - mean(OutputTrim$WS_3)) ^ 2))

#Create plot with NSE in title
OutputTrim %>% plot_ly(x = ~DATE) %>%
  add_trace(y = ~q, name = 'Modeled', type = 'scatter', mode = 'lines') %>%
  add_trace(y = ~WS_3, name = 'Measured', type = 'scatter', mode = 'lines') %>%
  layout(title=paste("NSE: ", round(NSE,2)))

```

18.8 Investigating a much bigger Monte Carlo

The next couple of code chunks make plots from a Monte Carlo I ran with 100,000 parameter sets.

We will look at the two plots and discuss.

How much difference was there in NSE in the top 100 runs of the 100,000 run set? How much would you expect the parameters to vary between these runs? What do you think it means if they did or did not vary much?

```

#the [,-1] drops the first column, which is just row numbers
pars100 <- read_csv("HBV/parsFrom100kRun.csv") [,-1]

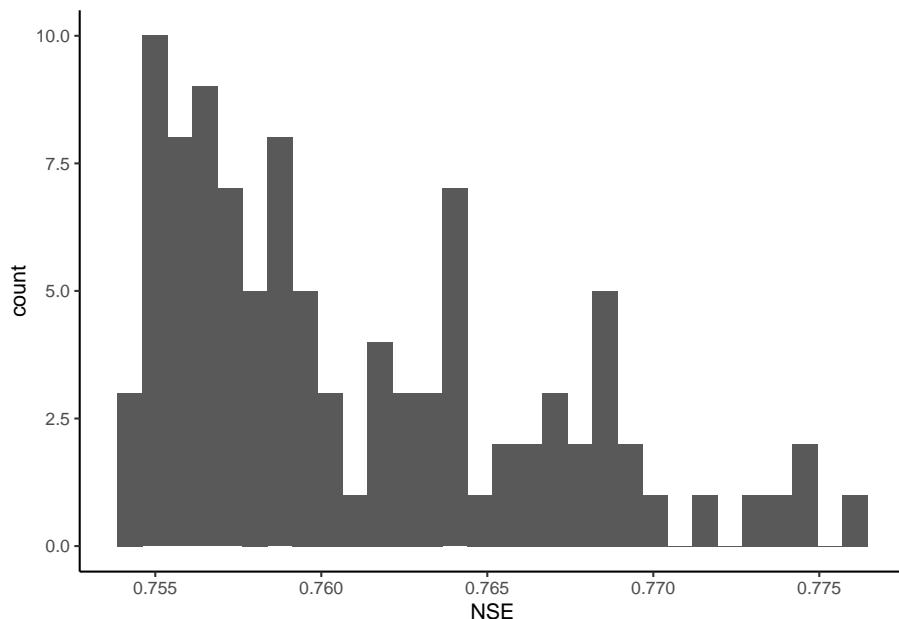
## New names:
## Rows: 100 Columns: 15
## -- Column specification
## ----- Delimiter: "," dbl
## (15): ...1, FC, beta, LP, SFCF, TT, CFMAX, k0, k1, k2, UZL, PERC, MAXBAS...
## i Use `spec()` to retrieve the full column specification for this data. i

```

```
## Specify the column types or set `show_col_types = FALSE` to quiet this message.
## * `` -> `...1`
```

```
ggplot(pars100, aes(NSE)) +
  geom_histogram()
```

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



The chunk below creates a parallel coordinate plot. Each parameter is shown, but normalized, where 1 is its maximum value from our random sets, and 0 is the minimum value.

Each connected line is a set of parameters. The green line is the set that had the highest NSE.

Look at this plot for a bit and discuss.

Are there any parameters that were always in the same range for these runs? What does that tell you about them? Are there any parameters that look like they spanned almost the whole range for these runs? What does that tell you about them? What does this plot tell you about the model's representation of the real world?

```
#filter the 100,000 run to the top ten
rankedpars <- filter(pars100, ranks <= 10)
```

```
#set the min and max values for each parameter for the plot
mins <- c(40, 1, 0.3, 0.4, -1.5, 1, 0.05, 0.01, 0.001, 0, 0, 1, 0, 0)
maxs <- c(400, 6, 1, 1.2, 1.2, 8, 0.5, 0.3, 0.15, 70, 4, 3, 0, 0)

#add maxes and mins to the parameter sets
rankedpars <- rbind(rankedpars, mins, maxs)

#Make NSE a character so the legend will work properly
rankedpars <- rankedpars %>% mutate(NSE = as.character(round(NSE, 2)))

#Make the NSE equal "Min/Max" for appropriate legend labels
rankedpars$NSE[rankedpars$NSE == "0"] <- "Min/Max"

#Make the ranks a factor datatype
rankedpars <- rankedpars %>% mutate(ranks = factor(ranks))

#Create parallel coordinate plot
rankedpars %>% ggparcoord(columns = 1:11,
  groupColumn = 13,
  showPoints = TRUE,
  scale = "uniminmax") +
  theme_minimal() +
  ggtitle("Top 10 parameter sets from 100,000 run Monte Carlo")+
  ylab("normalized parameter values")+
  xlab("parameters")
```

