Lab 1: Introduction to R

ENVIRON 710: Applied Statistical Modeling [*]

In this lab, you will upload software and packages needs to run R through R Studio and to use R Markdown, as well as learning the basics of coding in R. The learning goals of this lab are to:

- Install R, R Studio, R Markdown, etc.
- Become familiar with coding in R;
- Master the basics of visualizing and summarizing data - a skill that will be used in future labs.

The first task is to read and work through the below introduction to R (sections 1 - 14). If you want an additional introduction, there are many other introductions to R on the web. R functions included below are not always defined, so your task is to figure out what they do so that you can use them in the future. All functions used in this lab are basic functions necessary for daily use of R.

Also, note that this lab and most code in this class will be written in base R. There are other ways to write code, such as using `tidyr` and `dplyr` functions. Using these is absolutely fine and sometimes more efficient or intuitive. R is a language and there are many ways to achieve the same result, especially when it comes to organizing, wrangling, and visualizing data.

Upload a Word document under Assignments/Lab 1 on Sakai certifying that you can worked through the entire lab. After completing the lab, you should have installed all the necessary software and have the skills to start using R.

1. What is R?

R is powerful software for interacting with data. With R you can create sophisticated graphs, carry out statistical analyses, and develop and run simulations. R is also a programming language with an extensive set of built-in functions, so you can, with some experience, extend the language and write code to build your own statistical tools.

R is an open source implementation of the S language, which has been around and widely used for more than twenty years, first as S and then as the commercially available S-PLUS. A core team of statisticians and many other contributors work to update and improve R. Most importantly, R is free, high-quality statistical software that is useful for learning and doing statistics.

As described on the R project homepage:

> "R is a system for statistical computation and graphics. It consists of a language plus a run-time environment with graphics, a debugger, access to certain system functions, and the ability to run programs stored in script files."
>
> "The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions. Most of the user-visible functions in R are written in R. It is possible for the user to interface to procedures written in the C, C++, or FORTRAN languages for efficiency. The R distribution contains functionality for a large number of statistical procedures. Among these are: linear and generalized linear models, nonlinear regression models, time series analysis, classical parametric and nonparametric tests, clustering and smoothing. There is also a large set of functions which provide a flexible graphical environment for creating various kinds of data presentations. Additional modules are available for a variety of specific purposes."

[*]Created by John Poulsen with edits from TAs.

2. R Resources

There are hundreds of books, online resources and documents, blogs, and webpages on R. See the Resources file on the class Sakai webpage for a few useful R resources and shortcut reference cards. Or, look under Documentation on the R project homepage for a list of manuals, FAQ's, books, etc. This introductory document just scratches the surface of R, but will hopefully get you started.

Here are a few additional resources in addition to those on the class Sakai webpage.

https://www.rstudio.com/resources/cheatsheets/ - cheat sheets, especially helpful are the R Markdown cheat sheet and the data visualization cheat sheet (ggplot)
https://rfun.library.duke.edu/ - workshops and tutorials
https://r4ds.had.co.nz/ - R for Data Science book
http://www.cookbook-r.com/ - Cookbook for R – random helpful tasks
https://kapeli.com/cheat_sheets/LaTeX_Math_Symbols.docset/Contents/Resources/Documents/index - Math symbols used for R Markdown

3. Downloading R from the Web

Go the R project homepage at http://cran.us.r-project.org/.

Download R for Windows. Click on the link `Download R for Windows`, then click on the link `base`, and finally click on `Download R 4.2.2 for Windows` (or the most recent version, which could have a larger number). This begins the download of a file whose size is about 76MB. After the download is complete, double click on the downloaded file and follow the onscreen installation instructions.

Download R for (Mac) OS X. Click on the link `Download R for Mac OS X`, then click on `R-4.2.2.pkg` (or the latest version) to begin the download. Once it is downloaded in your Downloads folder or on your Desktop, double click on the downloaded file. Your Mac will unstuff the downloaded file and create an R folder. Inside this folder, there are many files including one with the R logo. Drag a copy of this to your Desktop and then drag the whole R folder to the Applications folder (located on the hard drive). After completing this, you can drag the original downloaded file to your trash bin.

R Studio After setup, you should see the R icon on your desktop. Clicking on the icon would open the standard interface, but many users prefer the RStudio interface. RStudio makes R easier to use, by including a code editor, debugging and visualization tools. It also facilitates printing R code and output as pdf, html, and Word files (we will come back to this in a future lab).

To install RStudio, go to http://www.rstudio.com/, click on "Download RStudio"", and then scroll down and click on"RStudio Desktop" - the free version. Make sure to choose the Open Source License, not the $995 Commercial Licence! Download the software as above, and then explore. Note that if you click on the RStudio icon, you do not also have to open the standard R program - it interfaces for you.

MiKTeX After installing RStudio, go to https://miktex.org/download and download the appropriate version of MiKTeX for either Windows or Mac. MiKTeX installs many of the things needed to use TeX for typesetting. You'll want to download the Installer first on the page by clicking "Download".

After doing the above, open RStudio and type the following in the console:

```
install.packages("rmarkdown").
```

This should ensure that you are ready to publish pdf's and fully reproducible documents.

4. Getting Started in R

This document will describe some of the basic functionality of R. Usually, you interact with R through a command-line interface (CLI) - type or paste a command in the console and R responds. Alternatively, commands can be written in a text editor and submitted to the console using the 'source' command. Once you have mastered a few commands, the CLI gives you control of an extremely powerful tool for interacting with data. Gaining mastery of a few R commands does take some learning and patience as R is finicky. You will undoubtedly experience some challenges as you work to learn a new skill that is not wholly intuitive, but it is well worth the effort. Onward!

The CLI version of R provides us with two prompts, that is, two signals that it is ready for input. They are the caret:

`>` which means that R is ready for a new command, and the plus sign:

`+` which means that R is waiting for you to finish the current command. A principal advantage of the CLI is that it simplifies the development and use of scripts. These allow us to keep a permanent, documented text record of the steps that we have done. Also note that `#` can be used to annotate code within a text editor or text file. Anything text or code on the same line after the `#` will not be run.

The below commands provide a glimpse of what R can do. Work through each of the examples below, typing the commands into R to see the results for yourself.

5. Using R as a Calculator

You can use R as a calculator. What do each of the following lines do?

```
2 + 2
12 * 3 - 10/2 + sqrt(16)
3^2
1:10
sum(1:10)
mean(1:10)
sd(1:10)
log(10)
log10(10)
```

The colon operator `:` creates an array of numbers from the first number to the second number. R has a number of built-in functions such as `mean`, `sum`, and `sqrt` that have obvious meanings. For example, the command `sum(1:10)` calculates `1+2+3+...+10`. Note that the `log` is the logarithm in base e, whereas `log10` denotes a base of 10.

R can do arithmetic operations on arrays. If you multiply an array of numbers by a single number, the multiplication happens separately for each number. You can also add or multiply equal-sized arrays of numbers.

```
2 * (1:15)
(1:10) + (10:1)
```

Assign variables using the `=` or the key combination `<-` (created to look like an arrow) signs to create variables. For example, type the below to assign a vector of numbers to `a1`.

```
a1 = 1:10
a1
a1 <- 1:10
a1
```

Variable names can consist of letter and numbers, but not symbols.

The `c` function in R concatenates the values (links them in a series). Because `c` is a reserved function name in R, it is preferable not to use `c` as the name of a variable lest you or R gets confused.

```
a1 <- 1:10
b1 <- 15:20
```

```
c1 <- c(b1, a1, b1)
c1
```

6. Using functions

R has many built in functions and you can also write your own. To see a function, just type its name. If you want to use the function, you need to add parentheses at the end, usually with arguments in the parentheses.

```
mean
mean(c1)
```

To get information on the way to parameterize a function, type `?` before the function. For example, `?mean` brings up a pop-up box that shows its full usage: `mean(x, trim = 0, na.rm = FALSE, ...)`.

You can also create functions of your own. For example, here is a simple function to compute the area of a rectangle. The function is named `findArea` and requires inputs for `x` and `y`. The second line of code calls the function, sending it values for `x` and `y`. What happens if you change the values for `x` and `y` below?

```
findArea = function(x, y){x * y}
findArea(x = 13, y = 4)
```

7. Changing your workspace

The workspace is your current R working environment and includes any user-defined objects (vectors, matrices, data frames, lists, functions).

Use the function, `ls`, to get a list of objects in your workspace. If you want to remove objects from your workspace, you can do so by removing them individually with `rm` or wipe all the objects (data, functions, variables) in your workspace with `rm(list = ls())`.

```
ls()
rm(c1)
rm(list=ls())
```

Use `Ctrl-C` or `Esc` to stop processing. This will work most of the time, and you will rarely need to ask your operating system to intercede.

Quit R altogether by typing the command `q()`. When you do so, R will ask you if you want to save your workspace. Saving your workspace allows you to have variables you have previously defined available without creating them again from scratch. Generally, whether or not you choose to save the workspace depends on your workflow. Personally, I almost never do it. I prefer to write scripts, which are a complete record of the analysis performed, and if I need to recreate the analysis, I resource the scripts.

If you choose to save the workspace, then a compressed image of the objects, called `.RData`, will be saved into your working directory. To easily access these objects again in a future session, use the load function (e.g., `load("mydata.RData")`).

#8. Working Directory The working directory is the location to and from which R writes and reads files by default. In Windows, there is a menu item, `File --> Change dir...`, that allows for selecting the working directory. In Mac OS, go to `Misc --> Change Working Directory`. In the CLI, you can use command lines.

```
getwd()
```

One easy way to keep your files organized is to have a `setwd` function at the top of your scripts. Run the function before running the script so that all the data, etc. can be easily located. For example, at the top of the script file type `setwd("c:/path/to/my/directory/")`. Here, `setwd()` is a function and `c:/path/to/my/directory/` is an argument for the function. Functions tell R what we want it to do, arguments are what we want R to do the function on. The forward slashes are used regardless of the underlying operating system. If you want to read or write to a different location, you must explicitly say so. Life is therefore much easier if all the data and scripts for an analysis are kept in a single (frequently backed up!) location.

A hint for Windows-users in setting the working directory: open Windows Explorer to your desired working directory, right click in the address bar and click `copy address as text`. Paste the address into the `setwd` command. Make sure to change \'s to /'s and execute the `setwd()` function.

For Mac-users, right-click a file, then hold the `option` key to see the 'Copy "file.name.csv" as Pathname' option, which will get the file path.

9. New Project

An alternative to setting your directory and workspace as described above is to create a Project in RStudio. To do this, click the `File` button on the RStudio toolbar and select `New Project` and follow the prompts to create a new directory or to add your project to an existing file. For more information, see: https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects. The benefit of a project is that by saving all your data and scripts in the same file as the project the working directory is automatically set. Open up the project and you are ready to go. For example, you could create a project for Lab 1 that includes the data and scripts needed for the lab.

10. Convention and Shortcuts

There are many different ways to do things in R. There are no official conventions on how the language should be used, but the following pointers may prove useful in communicating with other R-users.

- Although `=` works for assignment of variables, it is also used for other commands like passing arguments. By contrast, the arrow `<-` is only used for variable assignment. Most experienced R-users use the arrow.
- Spaces are cheap. Use spaces liberally between arguments and between objects and arithmetic operators.
- Call your objects useful names. Do not call your model "model", or your dataframe "data". Use names that you will be able to understand at a later date (months later when you need to revise your manuscript for publication).
- You can terminate your lines with semi-colons, but don't.
- You cannot (and should not) use an R-defined function (e.g., `mean`) as a variable name.
- Remember, R is case sensitive.
- To run an individual line of code from a text editor in Windows, place the cursor at the beginning of the line and hit, `Ctrl-r`. On a Mac, hit `Command-return`. Alternatively, type the command into R, or copy and paste the command into R.
- If you hit the up arrow in the console, the command that you entered previously will appear.
- `Ctrl-e` moves the cursor to the end of the line.
- `Ctrl-a` moves the cursor to the beginning of the line.
- To get help on a command type `help()` or `?` with the command of interest in the parentheses.

11. Data Manipulation and Storage

There are several different ways to store data in a workspace, but the following four options are essential for data analysis: vectors, strings, dataframes and matrices.

Vectors are 1-dimensional strings of data. To assign a data vector, use `c()`. Here we create a vector called 'emissions' and insert some values.

```r
emissions <- c(52.134, 82.741, 57.978, 100.0, 93.702)
emissions
```

R also allows character strings (vectors of words and characters).

```
Simpsons <- c("Homer", "Marge", "Bart", "Lisa", "Maggie")
Simpsons
```

Vectors can be combined and sorted.

```
emissions1 <- c(45.6, 53.2, 67.3, 29.1, 76.4)
emissions2 <- c(91.3, 89.1, 72.9, 95.1, 85.3)
emiss.total <- c(emissions1, emissions2)
emiss.total

sort(emiss.total, decreasing = F)
sort(emiss.total, decreasing = T)
```

What happens when you change `decreasing` from `FALSE` to `TRUE` in the `sort()` function? (Note that I abbreviate `TRUE` with `T` out of laziness. This is not necessarily a good habit.)

R also allows vector math.

```
emiss.total*pi
round(emiss.total, digits=0)
```

Vectors of data can be created using `rep()` and `seq()`. What does each line of code do?

```
rep(0, times = 5)
rep(c(1,2,3), each = 5)
seq(-5,5, by=0.1)
rep(seq(1,10), 3)
```

You can also use functions on a data vector.

```
sum(emiss.total)
length(emiss.total)
cumsum(emiss.total)
```

R has several summary statistics that are extremely useful for data analysis. Use `emiss.total` as the argument to the following functions: `mean`, `sort`, `min`, `max`, `range`, `median`, `summary`, `sd`, `var`. What do each of these functions do?

A dataframe is a two-dimensional collections of variables of many different types (character strings, numbers, factors, etc.). Dataframes are extremely useful and flexible for statistical data analysis.

```
dframe <- data.frame(Number=c(1:20),
           Emissions=emiss.total,
            School=c(rep("Fuqua", 10), rep("NSOE", 10)))
dframe
```

A matrix is similar to a dataframe, but more restrictive. It is a two-dimensional set of numbers (cannot include character strings).

```
matbin <- matrix(c(1,0,0,0,0,1,1,0,1,0,1,1), ncol=3,
           dimnames = list(c("R.1", "R.2", "R.3", "R.4"),
            c("C.1", "C.2", "C.3")))
matbin

mrandnums <- matrix(rnorm(100, mean=10,sd=100), nrow=20)
mrandnums
```

In `matbin`, we assign the vector of 0's and 1's to a matrix with 3 columns and define the column and row labels using `dimnames`. In `mrandnums`, we create a list of random numbers from a normal distribution, with the function `rnorm`, and then assign them to a matrix with 20 rows.

12. Graphing

R can create extremely sophisticated and attractive graphs and figures. Below are some simple plots using the base plotting package to get started. Other packages like `lattice` and `ggplot2` have additional graphing features. In this class, we will mostly use `ggplot2`. If you look up the R Graph Gallery on the Internet you can get a taste of the diversity of figures that can be created with R.

Some of the built-in functions for creating graphs include: `boxplot()`, `hist()`, `plot()`, `barchart()`, `pie()`. The following command constructs a boxplot.

```
boxplot(emiss.total, col = "darkblue", main = "",
        ylab = "y axis label", xlab = "x axis label",
          las = 1, cex.axis = 0.9, cex.lab = 1)
```

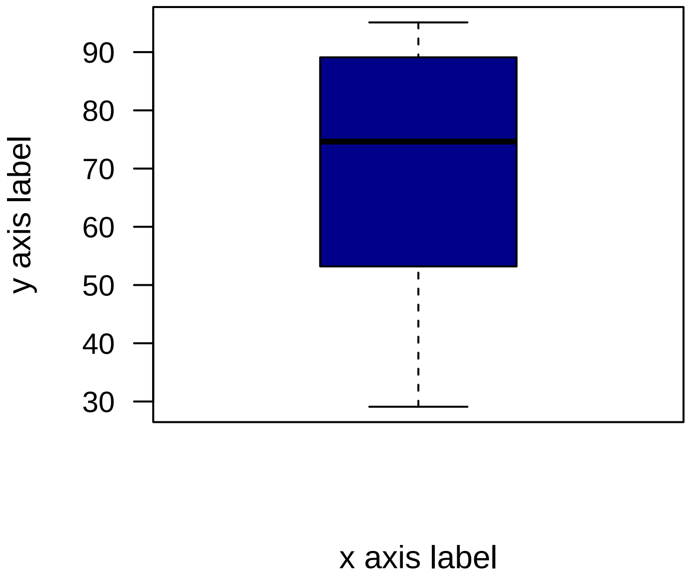

Figure 1: The figure legend should describe your figure.

Use `?boxplot` and `?par` to figure out what each of the arguments in the function does.

There are many commands to manipulate the graphical parameters of the plot. They can be found by typing `?par`.

You can save the plot in several ways, type `?savePlot()` to find all the possible formats (e.g., png, jpeg, tiff, bmp). Note that `savePlot` does not work on Macs. As an alternative, with the plot as an active window, go to `File --> Save As --> pdf`.

A note on saving graphs with MacOS.... It's tempting to just create graphics to the on-screen device (such as X11 on Linux or Quartz on MacOS) and then to use `Save As` from the menu. However, this doesn't allow you to explicitly set the options for the device, and on some platforms, you don't get to choose the file format. Also, if you resize the graphics window after you create the graph, you can get some unexpected results.

The best practice is to create a script file that begins with a call to the device driver (usually pdf or png), runs the graphics commands, and then finishes with a call to `dev.off()`.

```
png(file = "mygraphic.png", width = 400, height = 350)
 plot(x = rnorm(10), y = rnorm(10), main = "example")
  dev.off()
```

Not only will you often get better-looking results, you'll have the means to recreate the graphic file six months down the line, when you've long forgotten how you did it manually.

We can produce similar figures in `ggplot2`. You probably have to add this package with the `install.packages()` command and load it with `require()` (see below).

```
install.packages("ggplot2")
```

Now let's create the graph:

```
require(ggplot2)

emiss.total <- as.data.frame(emiss.total)

ggplot(emiss.total, aes(y = emiss.total)) + geom_boxplot(fill = "darkblue") +
  ylab("Emissions") + xlab("") +
   theme_bw()
```

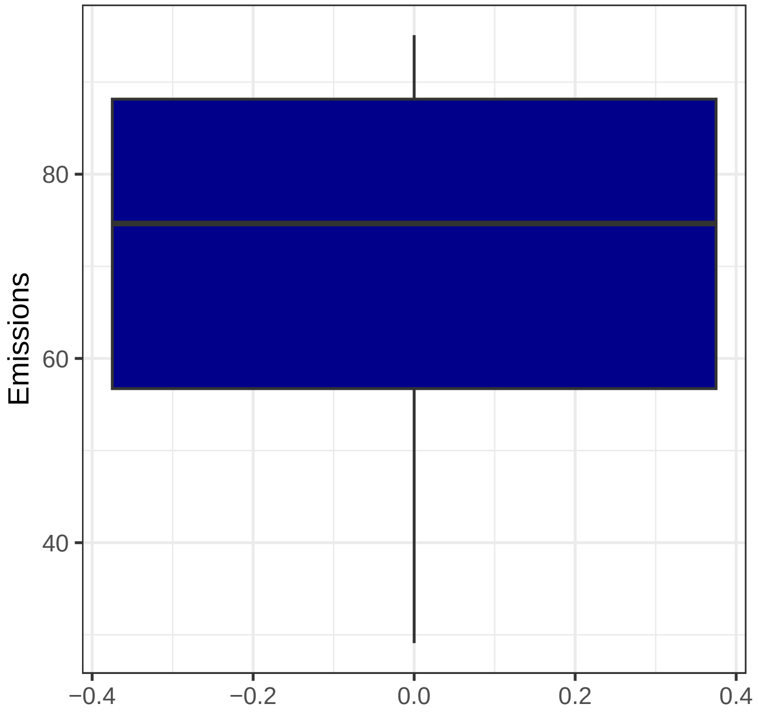

Figure 2: Boxplot created using ggplot.

Go to this link from to the Duke Library and watch all the videos related to `ggplot`: https://rfun.library.duke.edu/portfolio/ggplot_workshop/.

Note that for this class (and probably in your graduate career), the following features are required of figures:

- x and y-axes must be appropriately labeled with correct units;
- x- and y-axis tick labels must be on the correct scale and there should be enough tick marks to interpret the location of the data;
- y-axis tick marks should be perpendicular to the axis (see `las`);
- generally, the figure will NOT have a title, but will be described in a figure legend;
- all error bars should be described in the figure legend;
- figures should be clean and attractive (don't use bizarre colors or fonts).

13. Loading data into R

Now let's load data into R from an existing data file. In this example we will use the `mississippi.csv` data set found on Sakai. This dataset includes yearly water discharge volumes (in cubic kilometers) from the Mississippi River from 1954 to 2001. First, download the data file to your work file for the class and make sure your work directory is set correctly to locate the data.

```
mississippi <- read.csv("mississippi.csv", header = TRUE, sep = ",")

names(mississippi)
```

In the above code, we specified the type of data file (.csv) when we called the file name. `header = TRUE` defines the first line of the data file as a header (series of column names). `sep = ","` states that the columns are separated by commas. `names` lists all the variables in the data file.

`head()` shows the first several rows of a data.frame, matrix or table. `tail()` shows the last several rows of a data.frame, matrix or table.

```
head(mississippi)
tail(mississippi, 10)
```

14. Indexing

To select individual cells from a vector or dataframe, we define the values we are searching for in closed brackets. For example, the below commands provide (a) the first `Discharge` data point, (b) the first ten `Discharge` values, and (c) the 1st, 2nd, and 4th `Distcharge` values. Note here that the `$` denotes the column, `Discharge` from which we are indexing.

```
mississippi$Discharge[1]
mississippi$Discharge[1:10]
mississippi$Discharge[c(1,2,4)]
```

You can use indexing to select specific rows of data or specific data points. Remember, that when indexing a dataframe or matrix, it is specified as `data.frame[rows, columns]`. For example, below I do the following: (a) index the first row of data, (b) index the 3rd through 9th rows of data, (c) index just the discharge column.

```
mississippi[1,]
mississippi[3:9,]
mississippi[,2]
```

You can assign a column of data to a variable of its own:

```
misdis <- mississippi$Discharge
```

15 Calling variables from dataframes

The `attach()` command attaches the database to the R search path. This means that objects in the database can be accessed by simply typing their names. The `detach()` allows you to reverse the attach command. For example, if you `attach(mississippi)` you can work directly with the column names as variables.

For example, if you attach *mississippi* then you can run functions on the columns by just using their names.

```
attach(mississippi)
 mean(Discharge)
```

```
## [1] 563.125
```

BE CAREFUL with `attach`: if you have objects with the same name in different files, it will remember the last time the object was named. Most experienced R-users do not use `attach` for this reason.

```
detach(mississippi)
```

Rather, you can call the variables in *mississippi* in multiple ways, including: - using the dataframe$variable format - using the `with` command.

```
mean(mississippi$Discharge)
```

```
## [1] 563.125
```

```
with(mississippi, mean(Discharge))
```

```
## [1] 563.125
```

In the first, you can call individual columns from dataframes. In the second, with the `with` command, you make the entire dataframe available in case you need to use multiple variables from the same dataframe.

16 Install R packages

Many, many of the functions that we will need to use in R come from different packages, which are a set of predefined functions as a library. A package is a collection of functions, sample data, and documentation that describes how to use them. For example, above we installed the `ggplot2` package to use it to create a boxplot. All packages must first be installed to your R library. This will be done automatically by using `install.packages()`: it installs packages from CRAN, a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R.

```
install.packages("ggplot2")
```

Note that you must wrap the package name in quotes to install it. When you install a package in R, you get more than just that package but any packages that it has a dependency with as well. This ensures that the package you are installing will be fully functional as soon as you have installed it. These dependencies occur because you can always use functions from one package to create new functions to be included with another in a new package.

After they are installed, R packages can be loaded to your workspace using the `library` or `require` functions.

```
library(ggplot2)
require(ggplot2)
```

And we can remove a package from the workspace with `remove.packages()`.

```
remove.packages("ggplot2")
```

If you just need to use a package for a single function you can also cite it before the function with two colons.

```
ggplot2::ggplot(mississippi, aes(x = Year, y = Discharge)) +
           geom_line() + theme_bw()
```

There is a lot more to learn about R, we'll build on this base in future labs.

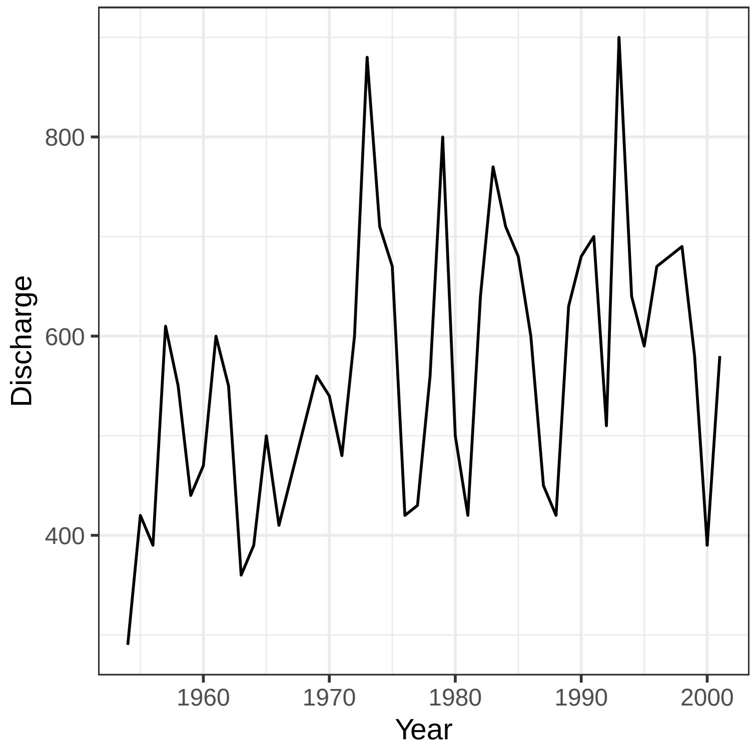

Figure 3: Annual water discharge from the Mississippi River from 1954 to 2001.