An Introductory Note on R

Bowen Chen, PhD Candidate
Department of Agricultural Economics, Kansas State University
bwchen@ksu.edu

August, 2018

I write this lecture note to help you learn R at the beginning stage. In this note, I document and explain some rudimentary yet important R codes. These are also the ones that I have frequently used for doing my applied economics research in the last three years. You should likely be able to handle many research tasks in R, if you understand all these codes. Yet, I shall remind you of that this is not a comprehensive list. You have to learn more to do more sophiscated and interesting works, such as economic modeling, econometric estimation, machine learning, text mining and so on. But this note will certainly help you build foundations for doing those.

I want to emphasize that this note will not help you unless you practice with it. Based on my experience, there is no better way of becoming proficient at R than practicing it a lot. You can only become a better R programmer after writing many R codes and sometimes reading some R codes. There will be discouraging error messages, and many of them could look "stupid" (or inefficient) at the next time you read them. Remember, this is the learning process that a beginner has to go through. So, please type the codes discussed in the notes and excecute them in R for your own benefits.

The note is organized as follows. Section 1 introduces some basic R codes. I wrote them just based on my poor memory, and I might miss some important ones. More would be discussed in the class. Section 2 focuses on the R toolkits for data management. An applied economics researcher could spend much time of cleaning the data. But as you will see, R has provided some powerful tools for us to clean the data. The third section introduces the codes useful for data visualization. I will teach you how to plot in R like a painter, to whom only creativity forms the boundary. The last section provides some advices that might help you to proceed to advanced levels.

1. R basics

The codes covered in this section are basic and fundamental R codes. You really have to know all of these before you go further. At this point, I assume that you have installed R and

Rstudio on you computer and that you know where to and how to excecute R codes. If you have not installed them, here is a short guidance.

1.1 Starting codes

The first thing that I would like to introduce is the working directory. The working directory, simply speaking, is the folder in your computer where you want to save and read data (or codes). Once you specify the working directory, R will read or save data automatically in that folder. Note that you can still save and read data in other place but you have specify it. This is code that I usually put at the first line of R script.

```
# Set the working directory.
setwd('C:/BWCHEN/2_Course/MathReview/Bowen2018/Lecture11')
```

The content 'C:/BWCHEN/2_Course/MathReview/Bowen2018/Lecture11' is where I want to save my works in my computer. In case that your forget the working directory that you have specified, use the codes below to figure it out.

```
# Return the working directory.
getwd()
```

The next thing I usually do is to clean the objects saved in R memory before I initiate a R project. The reason for doing this is that we do not mess up with other projects. Sometimes, it could be a chaos and cause problems if you save too many objects in R.

```
# Remove everything saved in R.
rm(list = ls())
# or to remove the single object
x <- 3 # Assign the numeric value 3 to the object called x.
rm('x')
```

The sign "<-" is for assigning the values to an object. By doing this, you create an object in R, and this object would be saved in the R memory. You could also use "=" for assignment, but it is discouraged. Note that, unlike C or C++, you do not have to specify the type while creating the object. R does it for you.

Like many other softwares, R has third-party packages that can be used for specific tasks. A package is merely a collection of a bunch of codes. For instance, the package "systemfit" allows you to run system of equations in R. To use packages, you have to install them first manually, and you just have to install them once. But you have to call the package every time you restart R. Here is an example.

```r
# Install packages
install.packages('dplyr')
# Call the package
library(dplyr)
# Or, you can use the function below.
require(dplyr)
```

As mentioned above, a package is a collection of codes. In R, you can have direct access to the codes inside the packages. The way of doing this is excecute the code in R.

```r
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x000000001a86a350>
## <environment: namespace:graphics>
```

A package might be used for doing many different tasks. For instance, inside the "dplyr" package, there are functions to create column names and to delete missing values (details to be covered in section 2). To know what are available and which one to use, you can google the name of the package, such as "dplyr r". Then you should find a PDF file named "Package 'dplyr' - CRAN-R". This file documents all the information about the package written by the package developers. For your information, CRAN (or Comprehensive R Archive Network) is the offical website that saves the officially recognized R packages. When you install packages in R, you are actually downloading them from CRAN.

Sometimes, we do not know how to use the functions in an unfamiliar package. You can use the codes below to read the help files. After running it, you will see the help file in the bottomright panel in R studio. That file usually contains detailed descriptions of the functions and sometimes examples.

```r
?systemfit
```

R provides multiple ways of reading data files into R. Possibly, the "csv" and "excel" type of data files are usually what we have to deal with. You can read and save data using the codes below.

```r
# Read CSV data.
dat1 <- data.table::fread('comtrade_trade_data.csv')
# Read Excel data
dat2 <- readxl::read_xls('UN Comtrade Country List.xls',
```

```
                          sheet = 1, range = 'A1:I294')
# Write the data in CSV format.
# You should see a csv file in your working directory
# after running the codes below.
write.csv(dat1, file = 'comtrade_trade_data_test.csv')
# Write the data in excel format.
xlsx::write.xlsx(dat2, file = 'comtrade_trade_data_test.csv',
                 sheetName = 'Sheet1')
```

Note that in the above codes, I use double colons to call the function "fread" from the package "data.table", "read_xls" from the package "readxl", and "write.xlsx" from the package "xlsx" (install these packages if you do not have them). There are alternative ways of reading and saving the data. But these codes are relatively more efficient than others, as far as I know. For your information, "read.csv" is often used for reading the CSV files. R can also read data files in other types. For instance, the "readstata13::read.dta13()" from the "readstata13" package allows you to read STATA dta-files.

1.2 Simple programming

To warm up, let's go through some simple examples first.

```
# Use R as an calculator
234/432
```

```
## [1] 0.5416667
```

```
7392347983 + 378923749
```

```
## [1] 7771271732
```

```
39749*203023
```

```
## [1] 8069961227
```

```
# Use R to print something
'Hello, world'
```

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

```
print('Hello, world')
```

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

```r
cat('Hello, world')
```

```
## Hello, world
```

Like what you see in the math classes, you can use "+", "-", "*","/" to do the basic math operations in R. Both "print"" and "cat" can be used to print things on your console. The two could be used interchangably, though their outputs might slightly differ. Remeber that a character, unlike numeric values, should have quotation marks.

The codes below show how to assign something to an object. The thing could be an numeric value, a character, or even a logical value. You will encounter an error when calling an object that has not been assigned before; the error would say "the object ... not found".

```r
x <- 3 # a numerical value
x
```

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

```r
x + 1 # Add 1 to the value in x
```

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

```r
x <- 'Kansas State University' # a character
x
```

```
## [1] "Kansas State University"
```

```r
x + 1 # Encounter an error, because x is an character
```

```
## Error in x + 1: non-numeric argument to binary operator
```

```r
x <- TRUE # a logical value
x
```

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

```r
y
```

```
## Error in eval(expr, envir, enclos): object 'y' not found
```

1.3 Dataset

A dataset has two basic elements: variables and observations. In most cases, you are expected to arrange the dataset in the long format, meaning that each row represents an observation and each column represents an variable. Here is an example:

```r
data(cars)
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

```r
tail(cars)
```

```
##    speed dist
## 45    23   54
## 46    24   70
## 47    24   92
## 48    24   93
## 49    24  120
## 50    25   85
```

"cars" is a laboratory dataset in R that records the speed (mph) of cars (first column) and the distances (ft) taken to stop (second column) of 50 cars (rows). So, the dataset has 50 rows and 2 columns. A dataset might have hundreds or even millions of observaions (or rows). You use "head" function to see values of the first 10 observations and "tails" to see values of the last 10 observations. Some other useful codes are:

```r
ncol(cars) # Number of columns in the dataset
```

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

```r
nrow(cars) # Number of rows in the dataset
```

```
## [1] 50
```

```r
colnames(cars) <- c('SPEED', 'DISTANCE') # Specify the column names.
summary(cars) # Summary stats of the data
```

```
##      SPEED         DISTANCE
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
```

```
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

```
cars$type <- 'Chevy' # Make an another column.
cars[1, 2]
```

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

Note that data are usually saved as "data.frame" in R. The "$" sign is how you get a column in the data frame. Also, you can get value in a specific position using brackets, like "[,]". Many more codes for dealing with datasets would be discussed in section 2.

1.4 Write functions

R is a functioning language. This means that R programmers write a lot of functions, and so should you. A function, from the mathematic perspective, can be written as $y = f(x)$, where y is the output, x is the input, and f represents the function. The universal syntax for writing function in R is similar to this, i.e.,

```
func <- function(x) {
  y = ...
  return(y)
  }
```

In this case, I use x as inputs. You can change this to k, m or any other thing. Then you code the function ('...') within the curly brackets and "return"" it as outputs. "func" is also a name that is upon your choice. Now I provide an example.

```
func <- function(x) {
  y = x^2  # square of x.
  return(y)
  }
y <- func(x = 2)
y
```

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

Here I write a function to calcluate the square of a number. You can also have multiple inputs, since we have multivariate functions. Here is an example.

```r
func2 <- function(x1, x2){
  y = (x1 + x2)^2
  return(y)
}
y <- func2(x1 = 2, x2 = 4)
y
```

```
## [1] 36
```

"func2" takes the squares of sum of two numbers. A function could take various forms.

```r
func <- function(x) {
  is.character(x) # Is the input a character?
}
func('J')
```

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

```r
func(1)
```

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

```r
func(TRUE)
```

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

```r
func <- function(x) {
  max(x) # What is the maximum number in the vector?
}

set.seed(101)
y <- rnorm(100, 0, 1)
func(y)
```

```
## [1] 1.852148
```

```r
func_max <- function(x) {
  max(x[-which(x == max(x))]) # What is the second-largest number in the vector?
}

y <- rnorm(10, 0, 1)
func_max(y)
```

```
## [1] 1.68596
```

There are several things to be noted. The "rnorm" function is to simulate numbers out of a normal distribution. The first element is the number of values to be simulated; the second is the mean; and the third is the standard deviation. You can also choose to sample from other distrbutions. Type "?rnorm" to see the details. When you simulate something, "sow" the seed in advance, so that you reap the same numbers.

The "func_max" function is bit more complicated but can be understood. In this function, I want to obtain the second-largest number in the vector. The logic of this function is to track where the largest number is located at and then delete it from the vector. The maximum number of the rest is what we need. This is not the smartest way of doing this. But I cannot show you "which" function and "==" function if I do not do this.

"==" is a test on equality. If the number on the left-hand side equals to the number on the right-hand side, the returned value is TRUE and, otherwise, FALSE. The functions for inequality tests include "<=", ">=", "<", ">". I think that the usages of these are self-explanatory. "which" is to return the locations of the TRUE values.

With the above being said, you know that "which(x == max(x))" means "returning the location where x equals to the maximum value of x". "x[.]" is to call the element of x. When you code "x[3]", R returns the value of the third element of x. When you have "-" sign in front of it, R returns the values except for the one at the third (this third element is deleted from x).

A relatively more efficient way is illustared below. The logic is to order the numbers first and then capture the value at the second. "sort" is a function for ordering the numbers.

```
func_max <- function(x) {
  sort(x, decreasing = TRUE)[2] # What is the second-largest number in the vector?
}

set.seed(101)
y <- rnorm(10, 0, 1)
func_max(y)
```

```
## [1] 0.9170283
```

The last example is:

```
func <- function(x1, x2) {
  y = x1 + x2
  out <- list(y = y, x1 = x1, x2 = x2)
```

```
  return(out)
}

y <- func(2, 3)
y[[1]]; y[['y']]
```

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

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

```
y[[2]];y[['x1']]
```

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

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

I want to introduce "list" to you through this example. "list" could be very important for advanced programming, but now I want you to know how to make a list and how to call element of a list. Within the function, I return the object as a list, named "out". In the list, I save three variables x1, x2 and y. When I excecute the function, the output is of course of list. You can return each one of them by calling the number or the name using double brackets "[[ ]]".

1.5 "if-else" condition

The "if-else" is frequently used in programming. Now I show you how to use it to play the coin flipping game in R. Here is the rule. The computer will randomly choose between head and tail. Then you take a guess. You win if your guess matches with the one chosen by the computer; otherwise, you loss.

```
guessfunc <- function(i){
  computerguess <- sample(c('head', 'tail'), 1)
  if(i == computerguess){
    print('You win')
  }else{
    print('You loss')
  }
}
set.seed(1011)
guessfunc('head')
```

```
## [1] "You loss"
```

```
guessfunc('tail')
```

```
## [1] "You loss"
```

Now we play the same game but with money involved. You get 10 dollars if you win and get nothing if you loss. You pay 6 dollars every time you want to play.

```
set.seed(1012) # Now we play with another machine.
guessfunc2 <- function(i){
  computerguess <- sample(c('head', 'tail'), 1)
  cost <- 6
  if(i == computerguess){
  revenue <- 10
  }else{
  revenue <- 0
  }
  netincome <- revenue - cost
  return(netincome)
}
guessfunc2('head')
```

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

```
guessfunc2('tail')
```

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

The "if-else" syntax could be simplied as follows. It is helpful when you have simple outcomes like the ones we have.

```
set.seed(1012)
guessfunc3 <- function(i){
  computerguess <- sample(c('head', 'tail'), 1)
  cost <- 6
  revenue <- ifelse(i == computerguess, 10, 0) # simplified version of if-else.
  netincome <- revenue - cost
  return(netincome)
}
guessfunc3('head')
```

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

```
guessfunc3('tail')
```

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

1.6 String analysis

String is an alternative word for character in R. Data do not necessarily be numeric values, and they could also be characters. Here are some basic codes for string analysis. To begin with, let's specify a character vector.

```
StringSample <- c('I like Kansas State University. I am proud of being a Wildcat.')
test <- strsplit(StringSample, split = ' ') # Split the string by space.
test[[1]]
```

```
##  [1] "I"           "like"        "Kansas"      "State"       "University."
##  [6] "I"           "am"          "proud"       "of"          "being"
## [11] "a"           "Wildcat."
```

```
paste(test[[1]], collapse = ' ') # Return to original format.
```

```
## [1] "I like Kansas State University. I am proud of being a Wildcat."
```

```
test <- strsplit(StringSample, split = 'a') # Split the string by the letter "a".
test[[1]]
```

```
## [1] "I like K"          "ns"                "s St"
## [4] "te University. I " "m proud of being " " Wildc"
## [7] "t."
```

Note that "strsplit" returns a list. "paste" is usually used to coerce the object into characters (try paste(1)). It can also be used to combine the characters.

```
nchar(StringSample) # Number of characters in the string
```

```
## [1] 62
```

```
grep('Kansas', StringSample) # Returns the index of element where "Kansas" is
```

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

```
grepl('Kansas', StringSample) # Is "Kansas" in the vector?
```

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

```r
sub('I', 'i', StringSample) # Replace "I" by "i", for the first time that "I" appears.
```

```
## [1] "i like Kansas State University. I am proud of being a Wildcat."
```

```r
gsub('I', 'i', StringSample) # Replace all "I"s by "i".
```

```
## [1] "i like Kansas State University. i am proud of being a Wildcat."
```

```r
substr(StringSample, 1, 4) # Choose the string from 1 to 4.
```

```
## [1] "I li"
```

```r
toupper(StringSample) # Capitalize all the letters
```

```
## [1] "I LIKE KANSAS STATE UNIVERSITY. I AM PROUD OF BEING A WILDCAT."
```

```r
tolower(StringSample) # De-capitalize all the letters
```

```
## [1] "i like kansas state university. i am proud of being a wildcat."
```

1.7 Matrix operations

R can handle matrix operations very easily. Here I present some the commonly used codes for matrix algebra. These are frequently used for programming the econometric models.

```r
set.seed(10)
A = matrix(sample(c(1:20), 9), nrow = 3, ncol = 3, byrow = TRUE)
A
```

```
##      [,1] [,2] [,3]
## [1,]   11    6    8
## [2,]   12    2    4
## [3,]   15   14   18
```

```r
set.seed(11)
B = matrix(sample(c(1:20), 9), nrow = 3, ncol = 3, byrow = TRUE)
B
```

```
##      [,1] [,2] [,3]
## [1,]    6    1   10
## [2,]   19    2   15
## [3,]   16    4   11
```

```r
# Matrix operations
A + B
```

```
##      [,1] [,2] [,3]
## [1,]   17    7   18
## [2,]   31    4   19
## [3,]   31   18   29
```

```r
A - B
```

```
##      [,1] [,2] [,3]
## [1,]    5    5   -2
## [2,]   -7    0  -11
## [3,]   -1   10    7
```

```r
A %*% B # Note that it is different to A*B
```

```
##      [,1] [,2] [,3]
## [1,]  308   55  288
## [2,]  174   32  194
## [3,]  644  115  558
```

```r
solve(A) # The inverse of A
```

```
##            [,1]        [,2]       [,3]
## [1,]  0.3846154 -0.07692308 -0.1538462
## [2,]  3.0000000 -1.50000000 -1.0000000
## [3,] -2.6538462  1.23076923  0.9615385
```

```r
B2 <- matrix(sample(c(1:20), 3), nrow = 3, ncol = 1, byrow = TRUE)
solve(A, B2) # Solve Ax= B2
```

```
##            [,1]
## [1,] -0.3846154
## [2,] -5.0000000
## [3,]  4.6538462
```

```r
crossprod(A, B) # = A'B
```

```
##      [,1] [,2] [,3]
## [1,]  534   95  455
## [2,]  298   66  244
## [3,]  412   88  338
```

```r
crossprod(A) # = A'A
```

```
##      [,1] [,2] [,3]
## [1,]  490  300  406
## [2,]  300  236  308
## [3,]  406  308  404
```

```r
t(A) # Transpose of A
```

```
##      [,1] [,2] [,3]
## [1,]   11   12   15
## [2,]    6    2   14
## [3,]    8    4   18
```

```r
diag(A) # diagonal matrix with diagonal elements in A
```

```
## [1] 11  2 18
```

```r
eigen(A) # eigenvalues of A
```

```
## eigen() decomposition
## $values
## [1] 30.684048  1.469330 -1.153377
##
## $vectors
##            [,1]       [,2]        [,3]
## [1,] -0.4363386 -0.2178706  0.02110212
## [2,] -0.3007986 -0.6383192 -0.81494639
## [3,] -0.8480147  0.7382960  0.57915204
```

1.8 The "apply" family and the "for" loop

Functions in "apply" family are used for iterations. For instance, you might want to conduct Monte Carlo simulations with 5,000 runs. The "for" loop is often used in many other softwars to achieve the goal. Yet, I want you to use the "apply" type of functions to do it in R. The reasons are twofold. First, "for" loop is, for somes reasons, slow in R. You could speed up your codes using the "apply" functions. Second, "apply" functions can be used in so-called parallell computing, which is a powerful tool for dealing with large datasets. You need to understand the "apply" functions before you get hands on those more advanced techniques. In the following examples, I use the "apply" functions to iterate the calculation of standard

deviations of all the column variables.

(1) "apply"

```
set.seed(101)
dat <- data.frame(matrix(rnorm(100, 0, 1), 20, 5)) # Create a matrix with 20 rows and 5 colu
# The data are from a standard normal distribution.
apply(dat, 2, sd) # Compute column means
```

```
##        X1        X2        X3        X4        X5
## 0.8667844 0.9730288 0.9738892 0.9049027 0.9792187
```

The "apply" function runs the "sd" (a base function calculating standard deviations) over the dataset. The first element is where you specify the data ("Dat" here). The second element specify the row or column. You use "1" for rows and "2" for columns. The last element is the function that you want to iterate with.

(2) "lapply"

```
out1 <- lapply(c(1: ncol(dat)), function(i) sd(dat[, i]))
out1
```

```
## [[1]]
## [1] 0.8667844
##
## [[2]]
## [1] 0.9730288
##
## [[3]]
## [1] 0.9738892
##
## [[4]]
## [1] 0.9049027
##
## [[5]]
## [1] 0.9792187
```

The "lapply" function is very different to the "apply" function as it has a different syntax. The "lapply" requires two arguments. The first argument is to specify the elements that you

wish to iterate over. In this case, I use "c(1: ncol(dat))", meaning that I want to iterate the function over the first to the last column of the dataset called "dat". The second argument is the function that you want to iterate with. You should understand now that this function is to calcualte the standard deviation of "i"th column of "dat". The returned object is a list. When you write the such functions, you should have clear ideas about the these two arguments in your minds. Otherwise, you might encounter issues of doing this.

(3) "sapply"

```
out2 <- sapply(c(1: ncol(dat)), function(i) sd(dat[, i]))
out2
```

```
## [1] 0.8667844 0.9730288 0.9738892 0.9049027 0.9792187
```

Once you understand "lapply", the "sapply" would be easy to understand, because it is just the simplified version of "lapply". By saying simplified, I mean the returned output is not a list anymore, but a vector. Now I show how to do the same thing using the "for" loops.

(4) "for" loop

```
for (i in 1:ncol(dat)){
  print(sd(dat[, i]))
}
```

```
## [1] 0.8667844
## [1] 0.9730288
## [1] 0.9738892
## [1] 0.9049027
## [1] 0.9792187
```

Sometimes, you might need a "break" while you are looping. The "break" is needed when you want to stop the loops if some conditions are met. For instance, you might want to stop the looping if the values are two large or too small. In the example below, I want to print all the numbers in the first column of "dat". I stop printing once the value is greater than 0.1. You can either use "break" or "stopifnot".

```
colValues <- dat[, 1] # Values in the first column

for (i in 1:length(colValues)){
```

```
  print(colValues[i])
  if(colValues[i] > 0.1) break
}
```

```
## [1] -0.3260365
## [1] 0.5524619
```

```
for (i in 1:length(colValues)){
  print(colValues[i])
  stopifnot(colValues[i] <= 0.1)
}
```

```
## [1] -0.3260365
## [1] 0.5524619
```

```
## Error: colValues[i] <= 0.1 is not TRUE
```

2. Data management

An empirical researcher often has to spend a big chunk of the time to clean the data. The raw data, which might be from survey or other sources, could be messy. R is very powerful when cleaning the data. And the most powerful tools in R are the functions included in the "dplyr" and "tidyr" packages. I encourage all of you to learn these two packages and master the functions in them. You can save plenty of time in the future if you spend enough time learning them now. Actually, these functionas are self-explanatory and easy to learn.

In the example below, I provide the codes that I have written for accomplishing a research task. Specifically, the objective of the research task is to show the allocation of China's soybean import shares across source exporting countries. I think it is a good example of showing you how to use the two packages.

2.1 Read the data

First, I went to the UN Comtrade database (a popular database for getting trade data that are provided by the United Nations) to the download the annual bilateral trade data. You can download the dataset from my GitHub website (see here)

```
# Read CSV data.
dat <- data.table::fread('comtrade_trade_data.csv')
head(dat)
```

```
##    Year Reporter Code Trade Flow Code Partner Code Classification
## 1: 2012           156               1            0             H4
## 2: 2012           156               1           32             H4
## 3: 2012           156               1           76             H4
## 4: 2012           156               1          124             H4
## 5: 2012           156               1          152             H4
## 6: 2012           156               1          156             H4
##    Commodity Code Quantity Unit Code Supplementary Quantity Netweight (kg)
## 1:           1201                 8            58382620493    58382620493
## 2:           1201                 8             5896225801     5896225801
## 3:           1201                 8            23891336539    23891336539
## 4:           1201                 8              630236688      630236688
## 5:           1201                 8                     60             60
## 6:           1201                 8                 104850         104850
##          Value Estimation Code
```

```
## 1: 34976644185                0
## 2:  3685112780                0
## 3: 14260026885                0
## 4:   401958481                0
## 5:          743                0
## 6:        27261                0
```

The original data are in CSV format. The "head" function returns the first 6 rows in the dataset. Of course, you can change the number of rows that you wan to see, using 'head(dat, n = 10)' (say, you want to see the first 10 rows). You can see that the format of the data is not bad. But there are redundant information that I do not need, such as the "Unit Code". Now I transforam the data into desired (long) format and get the needed information.

2.2 Data wrangling

Note that this data contain "Reporter Code" and "Partner Code". These are country codes used by the United Nations to identify countries. For instance, 156 is the code for China. In this dataset, all "Reporter Code" are 156 because I only download China's import data. To get the country names that we know, I match the trade data with another dataset provided by the United Nations that records the country names and country codes. The data can also be downloaded from here.

```
library(dplyr, warn.conflicts = FALSE)
```

```
## Warning: package 'dplyr' was built under R version 3.4.3
```

```
library(tidyr, warn.conflicts = FALSE)
```

```
## Warning: package 'tidyr' was built under R version 3.4.4
```

```
CtyCodeDat <- readxl::read_xls('UN Comtrade Country List.xls',
                               sheet = 1, range = 'A1:I294') %>%
  dplyr::select(1, 2)

cleanDat <- dat %>%
  dplyr::select(1, 4, 9, 10) %>%
  left_join(., CtyCodeDat, by = c('Partner Code' = 'ctyCode')) %>%
  filter(`Partner Code` != 0) %>%
  dplyr::rename(Country = `cty Name English`,
                Quantity = `Netweight (kg)`) %>%
```

```
mutate(Country = ifelse(Country %in% c('Brazil', 'Argentina', 'USA'),
                        Country, 'ROW')) %>%
group_by(Country, Year) %>%
summarise(Value = sum(Value), Quantity = sum(Quantity)) %>%
mutate(Value = as.numeric(Value),
       Quantity = as.numeric(Quantity)) %>%
arrange(Year)
```

Now I explain the codes line by line.

(1) The first line is to read the excel file into R using the "read_xls" function. I call the dataset "CtyCodeDat".

(2) The "%>%" is called pipeline operations. It is called "and then", meaning you do the followings on the same dataset. So you do not have specify the name of the dataset in the following codes.

(3) The "select" is a function for selecting columns in the dataset. Here I select the first and second columns in the "CtyCodeDat" dataset. You can also select based on the names of the columns.

(4) The trade dataset "dat" contains 11 variables in total, but I only need four of them: "Year", "Partner Code", "Netweight (kg)" and "Value". The other variables are redundant for the analyses and are then discarded. Again, I use the "select" function based on their column numbers, which are 1, 4, 9 and 10.

(5) The "left_join" function is to merge the "dat" dataset with the "CtyCode" dataset. I merge them by the country codes, which are called differently in the two datasets. You just have to use the "by" argument. "Partner Code" is the name in "dat", and "ctyCode" is the name in "CtyCodeDat". The "." calls the data that you are dealing with. With "left_join", the dataset on the left is to be used as the benchmark, and every observations contained in the right and not in the left are discarded. The dataset on the right would be used as benchmark if you use "right_join". No information would be discarded if you use "full_join". Upon merging, another column "cty Name English" would appear in the dataset.

(6) The "filter" function is used to delete the observations. Here I delete the observations with "Partner Code" equals to 0, because these observations are for "World", which gives the total import values. I only need bilateral import data.

(7) I dislike two column names used in the dataset, so I change them using the "rename" function.

(8) The "mutate" function is the one that you use for changing or creating a new column in the dataset. Here I change the values in column "Country". All countries except for Brazil, Argentina, and USA are considered one region named "ROW" (or rest of the world).

(9) The next thing I do is to take the sum of imports of countries in "ROW". I group the country by "Country" first and then take the sum for each group.

(10) Lastly, I order the dataset by "Year".

2.3 Data analysis

I am interested in that which country exported the most soybean to China. The statistics that I decide to look at is the average import share from 2014 to 2016. The country with highest import share is the biggest exporter to China.

```
ImportShare <- cleanDat %>%
  filter(Year >= 2014, Year <= 2016) %>%
  group_by(Year) %>%
  mutate(Total = sum(Value)) %>%
  mutate(Share = Value/Total) %>%
  group_by(Country) %>%
  summarise(Share = mean(Share))
ImportShare
```

```
## # A tibble: 4 x 2
##   Country    Share
##   <chr>      <dbl>
## 1 Argentina 0.0970
## 2 Brazil    0.469
## 3 ROW       0.0445
## 4 USA       0.389
```

(1) I only keep the data during 2014 and 2016 by using the "filter" function.

(2) To compute for the import share, I have to calcuate the total import value for each year. Then I group the data by year and calculate the "sum" of the values for each year.

(3) Then I create the "Share" column using "mutate". This is the import share, which equals to the ratio of the imports to total imports.

(4) The last thing is to compute for average import share for each country. I group the data by country first and then calculate the "mean" for each country.

Next I calculate the correlation between the soybean imports from the U.S. and the soybean imports from Brazil, because the two countries compete with each other in China's soybean market.

```
ImportCor <- cleanDat %>%
  select(-Quantity) %>%
  filter(Country %in% c('USA', 'Brazil')) %>%
  tidyr::spread(Country, Value)
cor(ImportCor$Brazil, ImportCor$USA)
```

```
## [1] 0.9458747
```

The codes above use an important function called "spread". The "spread" function coverts the dataset from long format into short format. Its counterpart, "gather", would convert the dataset from short format into long format. The correlation is very high, but could be spurious. Because the imports from both countries have been increasing over time. The time trends are possibly the reason that the two numbers are highly positively correlated. As a remedy, I take first-order difference of the data and calcluate the correlated of differenced import data. The "drop_na" function is used for deleting the rows with "NA" values.

```
ImportCorDif <- ImportCor %>%
  mutate(LagBrazil = lag(Brazil), LagUSA = lag(USA)) %>%
  mutate(USADif = USA - LagUSA, BrazilDif = Brazil - LagBrazil) %>%
  tidyr::drop_na()
cor(ImportCorDif$BrazilDif, ImportCorDif$USADif)
```

```
## [1] 0.2117795
```

There are other useful functions contained in the two packages. The "dplyr cheatsheet" (google it) provides a good summary of them.

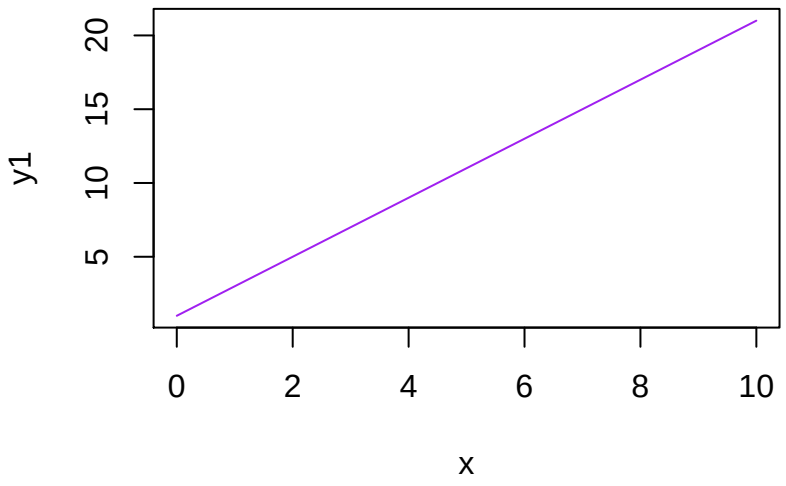
3. Data visualization

The "plot" and "ggplot" functions are two commonly used ones for data visualization in R. "plot" function is a base function, and I usually use it for producing graphs that are with simple designs. "ggplot" is the function from the "ggplot2" package, and I usually use it for producing "complicated" graphs. You should propably understand why after you go through the examples below.

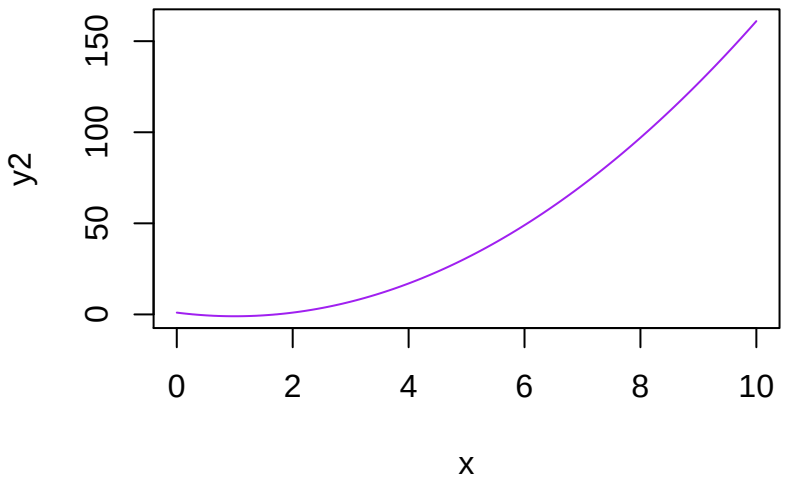
3.1 The "plot" function

First, I want to share with you the codes that I have used for the math reviews. These are also the typical ways of using the plot function. As a reminder, these are the functions that aim to produce the the graph of different functions.

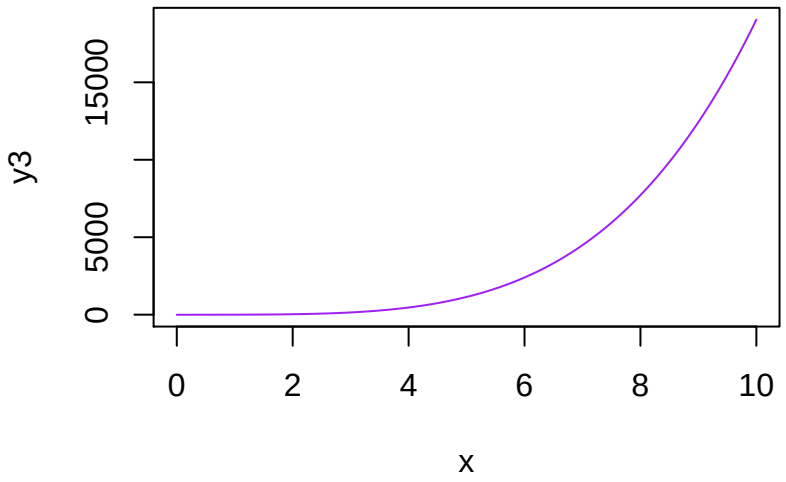
```
# Specify the domain.
x <- seq(0, 10, by = 0.01)

# A linear function.
y1 <- sapply(x, function(i) 2*i + 1)    # y = 2x + 1
plot(x, y1, type = 'l', col = 'purple')
```



```r
# A quadratic function.
y2 <- sapply(x, function(i) 2*i^2 - 4*i + 1)  # y = 2x^2 - 4x + 1
plot(x, y2, type = 'l', col = 'purple')
```

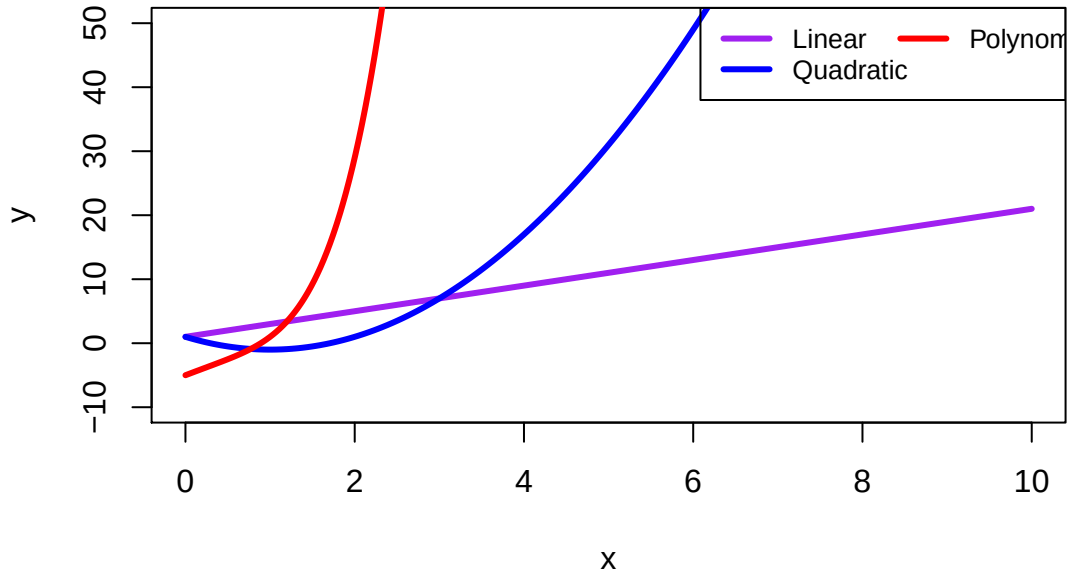



```r
# A polynomial function.
y3 <- sapply(x, function(i) 2*i^4 - i^3 + 5*i - 5) # y = 2x^4 - x^3 + 5x - 5
plot(x, y3, type = 'l', col = 'purple')
```




```
# Plot all the functions in one graph.
plot(x, y1, type = 'l', main = 'Figure. Graph of functions',
     col = 'purple', lwd = 3, xlim = c(0, 10), ylim = c(-10, 50),
     xlab = 'x', ylab = 'y')
lines(x, y2, col = 'blue', lwd = 3)
lines(x, y3,  col = 'red', lwd = 3)

legend('topright', legend=c("Linear", "Quadratic", "Polynomial"),
                   col=c("purple", "blue", 'red'),
                   lwd = 3, cex = 0.8, ncol = 2, text.width = 1)
```

Figure. Graph of functions



In the "plot" function, you have to specify the values for x-axis ("x" in the case) and values for y-axis ("y1", "y2", and "y3" in the case). After specifying them, you need to specify the "type", namely, the type of graph that you wish to draw. It can be a "point" (type = 'p'), "line" (type = 'l') or others (see details from ?plot).

The other arguments are:

"main" – title of the graph;

"sub" – subtitle of the graph;

"xlab" – name of the x-axis;

"ylab" – name of the y-axis;

"col" – color of the graph;

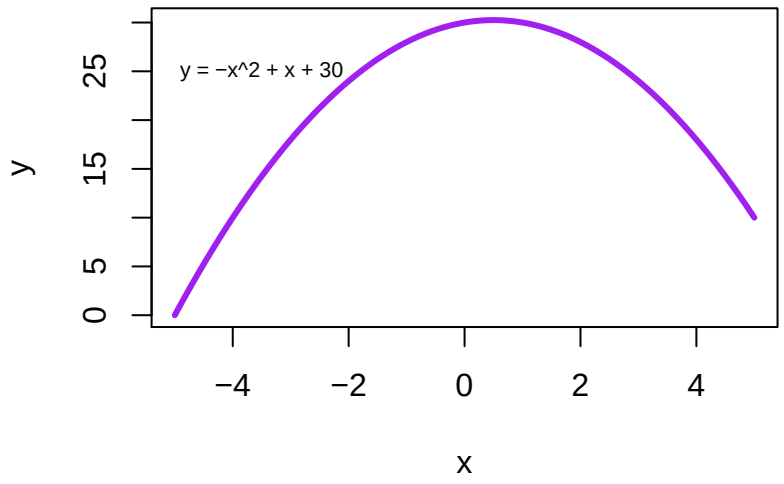
"lwd" – line width;

"xlim" – limits of the x-axis; "ylim" – limits of the y-axis.

This is an incomplete list of the arguments in the "plot" function that you can specify. You can see more of it by reading the help file ("?plot") or refer to this website (https://www.statmethods.net/advgraphs/parameters.html). Here is another example that I add some texts into the graph by using the "text" function. The numbers (-3.5 and 25) define the

locations where I want the text to be displayed. The "labels" specify the text to be displayed, and "cex" specify the size.

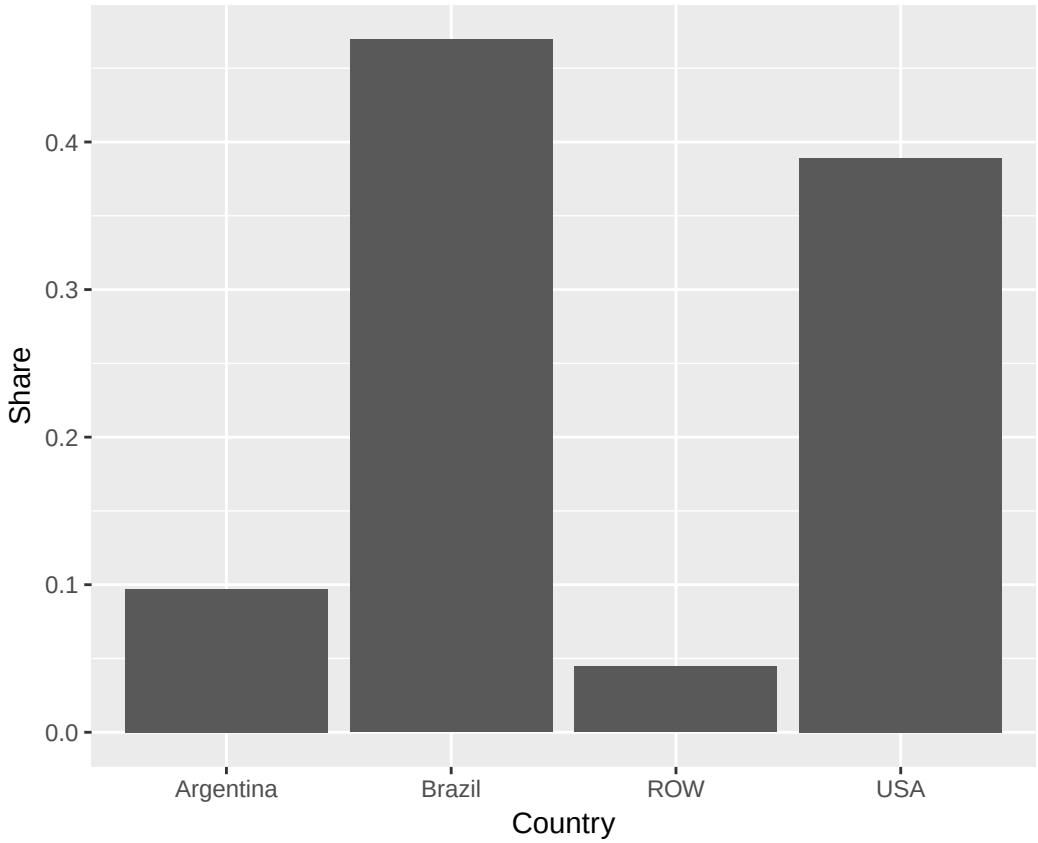
```
# Specify the domain
x <- seq(-5, 5, by = 0.01)

# Concave function
y1 <- sapply(x, function(i) -i^2+i+30)    # y = -x^2+x+30
plot(x, y1, type = 'l', col = 'purple', lwd = 3, xlab = 'x', ylab = 'y')
text(-3.5, 25, labels = 'y = -x^2 + x + 30', cex = 0.7)
```



3.2 "ggplot" function

Now I show you how to use the "ggplot" function. Unlike the "plot" function, you are supposed to put all the variables to be visualized in one data frame. I give an example first and then explain them to you with more details.

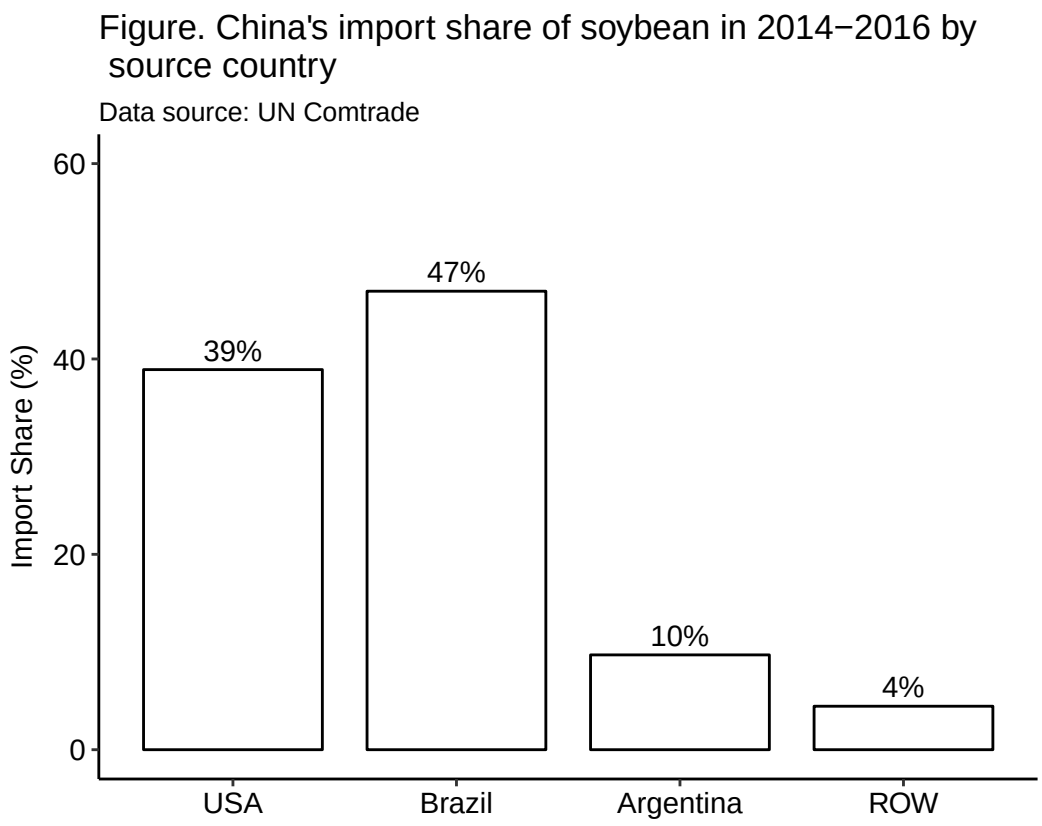
```
library(ggplot2)
g1 <- ggplot(data = ImportShare) +
  geom_bar(mapping = aes(Country, Share), stat = 'identity')
g1
```



The functions above are the main body of the "ggplot" function. The basic elements required for "ggplot" are the name of the data object ("ImportShare" here), the names of axes ("Country" and "Share" here), the type of graph you wish to display ("geom_bar" here). If you wish to draw a graph in other types, you have to change "geom_bar" to "geom_line", "geom_point" or others. This website provides a comprehensive list of the "geom"s included (see http://sape.inf.usi.ch/quick-reference/ggplot2/geom). Next, I improve on the graph based on my design.

```
g2 <- ggplot(data = ImportShare) +
  geom_bar(mapping = aes(Country, 100*Share), stat = 'identity', col = 'black',
           fill = 'white', width = 0.8) +
  geom_text(mapping = aes(Country, 100*Share + 2,
                          label = paste0(round(100*Share, 0), '%'))) +
  labs(x = '', y = 'Import Share (%)',
       title = "Figure. China's import share of soybean in 2014-2016 by \n source country",
       subtitle = "Data source: UN Comtrade") +
  theme_classic() +
  scale_y_continuous(limits = c(0, 60)) +
```

```
  scale_x_discrete(limits = c('USA', 'Brazil', 'Argentina', 'ROW')) +
  theme(axis.text = element_text(color = 'black', size = 11),
        axis.title = element_text(color = 'black', size = 11))
g2
```



The graph displayed above is the one that I have in my mind. I think this is a good one that conveys the information very clearly – the U.S. and Brazil are now the major soybean exporters to China, since the two countries took dominant imports shares during 2014-2016. Now I explain the codes used to generate the graph.

(1) In any "geom" function, you can specify the color and line width to be displayed. "col" defines the color of the border line, "fill" defines the color to be filled in the bars, and "width" defines the width of the bars.

(2) The numbers displayed above the bars were not in the default graph. I add them into the graph by using the "geom_text" function. The arguments in "aes()" define the locations where I want to display, and "label" defines the texts to be displayed. Here I use the "round" function to round the numbers and then use "paste0" function to

combine them with the percentage sign "%".

(3) In the "labs" function, you can specify the names of axes, and title and sub title.

(4) "theme_classic" is a way of choosing the type of theme. The classcial one is the one without any lines and colors in the background. This is my personal favorite.

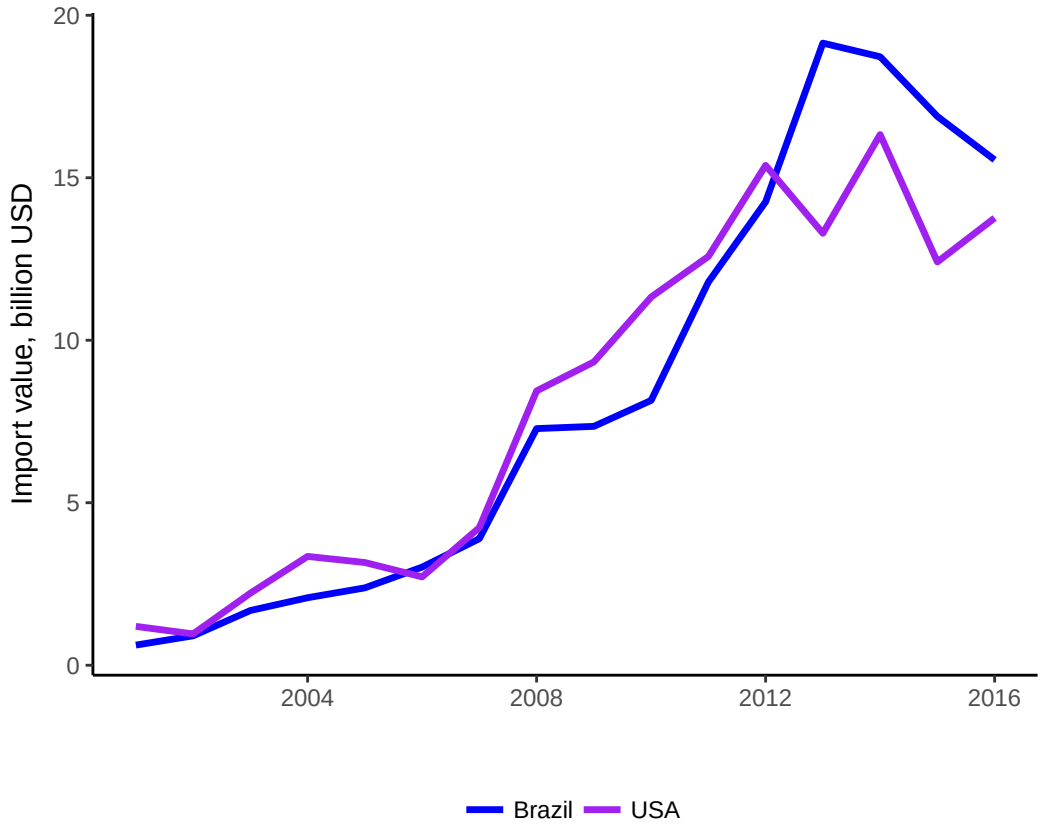
(5) The "scale_x" and "scale_y" functions are the ones through which you modify the settings of axes. You choose "discrete" when the data are discrete, and "continuous" when the data are continuous.

(6) "theme" is where you can do more things with the graph. Here I specify the color and the size.

The codes above show the typical way of using the "ggplot" function for data visualization. It is a fundamental framework that you can use to produce graphs with more sophiscated components. The "ggplot" function, by design of the developers, follow certain graphic grammers, or principles of drawing graphs. Based on the grammer, a plot is made up of the following four basic elements: (1) data and aesthetic mappings, (2) geometric objects; (3) scales; (4) facet specification. With the four elements, you can describe a wide range of graphics; and by modifying the four elements, you can create a graph whatever you want it to look like.

To enhance your understanding of the function, I present one more example. Here I show the historic changes in China's soybean imports from the U.S. and Brazil during 2001-2016.

```
USBrazilImport <- filter(cleanDat, Country %in% c('Brazil', 'USA'))

g3 <-
  ggplot(data = USBrazilImport) +
  geom_line(mapping = aes(Year, Value/1000000000, color = Country), size = 1.2) +
  theme_classic() +
  labs(x = '', y = 'Import value, billion USD') +
  scale_color_manual(values = c('blue', 'purple')) +
  theme(legend.position = 'bottom',
        legend.title = element_blank())

g3
```

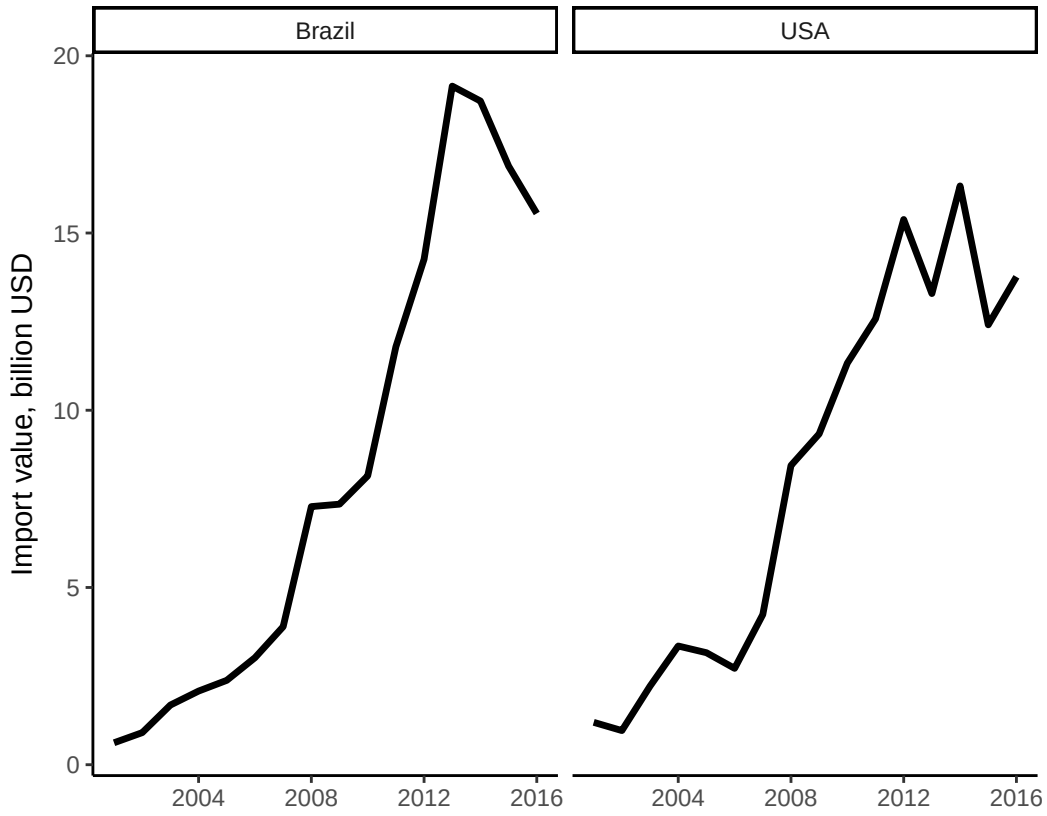


The objective of the above example is to show you how to visualize the data that are group-specific. In this example, we have import data for the U.S. and Brazil. One way of showing the two series of data is to use two different colors for each of them, so readers can tell the differences. To do this, you specify "color" in the "geom_line" function. Then you can change the colors using the "scale_color_manual" function.

Another way of doing this is to use two facets, and we use one facet for one country. This is done by using the "facet_wrap" function.

```
g4 <-
  ggplot(data = USBrazilImport) +
  geom_line(mapping = aes(Year, Value/1000000000), size = 1.2) +
  theme_classic() +
  labs(x = '', y = 'Import value, billion USD') +
  facet_wrap(~Country) +
  theme(legend.position = 'bottom',
        legend.title = element_blank())
```

```
g4
```


Brazil
USA
Import value, billion USD
20
15
10
5
0
2004
2008
2012
2016
2004
2008
2012
2016

4. The way ahead

When you understand all the codes discussed above, you would be at a critical point of advancing to intermediate levels. Here are some suggestions for you to advance your R skills when you reach that point. I do not recommed you to take these suggestions unless you are serious about R.

4.1 Read R blogger

The R blogger (https://www.r-bloggers.com/) is a nice website that collects R news and tutorials that are contributed by worldwide R progammers. The information are overwhelming, because they update at daily basis. Be patient and stay hungry. I was eager to improve my R skills about two years ago. During the period of time, I read several blogs every day and try to run some of the R codes discussed in those blogs. I benefit enormously from the exercises. I still read the bloggers occasionally, but the marginal benefit of doing this is a bit low now.

4.2 Use R

As mentioned above, you can never become proficient at R without enough practices. If you have data works, do them in R and try to avoid excel as much as you can. Do not take me wrong on this. I do not dislike Excel. My point is that you never improve as a R user by using excel or other softwares so frequently.

I have used R to program many economic and econometric models that I met when taking the guraduate courses, including STAT 712 (Optimization), ECON 930 (Econometrics II), ECON 720 (Microeconomic theory), and ECON 981 (International Trade) and so on. It was quite challenging. But it turns out that it also benefits me a lot – I improved my R skills and gained deeper understanding of those economic and econometric models. You might want to give it a try. As a reminder, this strategy might takes time and energy.

4.3 Learn from other R programmers

R is an open source, so you can have access to the codes embeded in the R packages. Get the functions, print them out, and read them line by line. Challenge yourself by asking the followig question – "what are codes for". If you understand the codes and their purposes, you can write them later.