STOR 565 Fall 2019 Homework 1

Hunter Finger

*Remark.* This homework aims to introduce you to the basics in **R**, which would be the main software we shall work on throughout this course. It might look like a long homework but it has only 13 problems. The rest are explanations regarding basic things in R. **Total number of points**: 120. Recall all Homeworks are worth the same weight when I compute your final grade. I will convert each HW to out of 100 then and compute the average of your HW scores.

**Instruction.** Download the whole folder for this homework and unzip the file. Open the **RMarkdown** document with surfix “.rmd” via **RStudio**. Click Knit to create a PDF document (remember to install the necessary packages in your **R**). The “knit” option can be changed to pdf *or* html (or even Word). When submitting the HW, you **will knit to a pdf document print the output and handover in class**. The html file in the folder has clickable links for the various references below on Latex etc. By removing the results='hide' and fig.keep='none' options in the code chunks, the code outputs and the plots will display in the created file. For more information about the **RStudio**, refer to the section **Getting Started**; about the **RMarkdown**, refer to [the online tutorial](http://rmarkdown.rstudio.com/lesson-1.html) and [the online manual of knitr](https://yihui.name/knitr/) by Yihui Xie from **RStudio, Inc.**

**Important caveat**: In the past some students have had issues when knitting the file to pdf. If your work does not show up, knit to html or doc (this seems to be fine) and print the output. At the end of the day all we need to grade your HW is the paper copy of your HW.

**Latex:** If the file does not compile properly in the intial go around you might need to install Latex onto your system. See [Latex installation](https://www.latex-project.org/get/) for details. You might then need to reinstall RStudio. Latex is a fantastic framework that everyone in the computational world uses to write technical documents. See [Tex exchange](https://tex.stackexchange.com/questions/1756/why-should-i-use-latex) or [Medium](https://medium.com/@marko_kovic/why-i-write-with-latex-and-why-you-should-too-ba6a764fadf9) article for more details as to why you should use Latex. See [this guide](http://www.docs.is.ed.ac.uk/skills/documents/3722/3722-2014.pdf) for a beginners introduction. I do not envision you having to use a ton of Latex in the course but this is important for me when making these assignments for the assignment to typeset properly especially with reference to math symbols.

**diagram package**: For displaying some of the pictures in the file we used an R package called diagram. You will need to install this before the file knits properly.

**Please turn off the display of example code chunks (by specifying include=FALSE), complete the exercise code chunks (remember to turn on the eval option), fill in your name and create a PDF document, then print and submit it.**

# Getting Started

**R** is one of the most common coding language in data analysis nowadays. It is free, open-source and powerful software environment for statistical computing and graphics. A nice description of **R** can be found in [the course website](http://data.princeton.edu/R/) of German Rodriguez from Princeton. To download and install **R**, click into [the CRAN (Comprehensive R Archive Network) Mirrors](https://cran.r-project.org/mirrors.html) and choose a URL that applies to you. To run with **R**, click “RGui.exe” in Windows system or “R.app” in Mac OS.

The console of **R** is not the most friendly interface to work with, as compared to the more handful editor **RStudio**. You can download and install it via [its website](https://www.rstudio.com/products/rstudio/download/) and choose the free version of **RStudio Desktop**. Note that when you run with **RStudio**, the mirror of **R** should have been installed in your system, even though you are not accessing to it directly.

The ensuing sections lead you to the essentials of **R**. To explore more about how **R** works, please refer to (Dalgaard 2008, @venables2017) and [the tutorial](http://data.princeton.edu/R/) of German Rodriguez.

# Basics in **R**

**R** is an object-oriented language. Hence the “data” we work on are formatted as a particular object that meets some structural requirements (subjected to a particular class). Thus one should first understand which class of object he/she has on hand, and then figure out the applicable operations on it.

In a hierarchical manner, the more advanced class consists of ingredients from more fundamental classes. Vectors, matrices, lists, data frames and factors are the most commonly used fundamental classes in data analysis.

## Vectors and Matrices

Vector is a collection of “data” that share the same type (numeric, character, logic or NULL). Matrix arranges “data” of the same type in two dimensions. Note that there doesn’t exist a “scalar” object, which would be treated as a vector of length 1.

### Create a Vector

The concatenation function c( ) can be used to manually create a vector in **R**. When using the c( ) function, numbers are entered as a list with commas between each new entry. For example, x <- c(1, 2) creates a vector and assigns it to the variable x.

To create a vector that repeats times, we can use the replication function rep( , n). For example, a vector of five TRUE’s can be obtained by x <- rep(TRUE, 5).

Finally, we can create a consecutive sequence of numbers using the sequence generating function seq(from = , to = , by = ). Here, the from, to and by arguments specify where the sequence begins, ends, and by how much the sequence increments. For example, the vector can be obtained using x <- seq(2 , 8, 2). A convenient operator : similar to seq also creates the consecutive sequence with step sizes by or . Try 1:4 and 4:1.

For more information, the commands ?c, ?rep and ?seq access to the online **R** documents for help.

**Exercise 1.** *(5 pt)* Using the c, rep or seq commands, create the following 6 vectors:

x1 = (2, .5, 4, 2);

x2 = (2, .5, 4, 2, 1, 1, 1, 1);

x3 = (1, 0, -1, -2);

x4 = (“Hello”," “,”World“,”!“,”Hello World!");

*Note:* The quotation marks and sometimes the exclamations marks are rendered a little funky in the pdf/html. Just go with it.

**Hint.** For x4, take this opportunity to experiment with the paste function.

x5 = (TRUE, TRUE, NA, FALSE);

**Remark.** Check ?NA and class(NA) to learn more about the missing value object NA. This is not relevant for x5.

x6 = (1, 2, 1, 2, 1, 1, 2, 2).

x1 = c(2, .5, 4, 2)  
x2 = c(x1, seq(from = 1, to = 1, length.out = 4))  
x3 = seq(from = 1, to = -2, length.out = 4)  
x4 = c("Hello", "", "World", "!", "Hello World!")  
x5 = c(T, T, NA, F)  
x6 = c(1,2,1,2,1,1,2,2)

### Create a Matrix

An -by- matrix can be created by the command matrix( , m, n) where the first argument admits a vector with length compatible with the matrix dimensions. For example, x <- matrix(1:4, 2, 2) creates a -by- matrix that arranges the vector (1, 2, 3, 4) by column. To arrange the vector by row, specify the byrow option as follows: x <- matrix(1:4, 2, 2, byrow = TRUE).

Moreover, the command binding vectors/matrices by row rbind and by column cbind are also useful. Check **R** documents for their usages.

**Exercise 2.** *(5 pt)* Using matrix, and rbind, create

More precisely first define a set of four vectors corresponding to the rows of the above matrix and then use rbind to make a corresponding matrix. Note: you will need to play around with the deparse.level option in rbind to get the matrix as above.

a = seq(1, 4, 1)  
b = seq(1, -2, -1)  
c = c(2, .5, 4, 2)  
d = rep(1, 4)  
X <- rbind(a, b, c, d)

### Indexing

There are three ways to extract specific components in a vector.

x <- 1:4  
  
# specify the vector of indices  
x[c(1,4)]  
  
# specify a logical vector of identical length  
x[c(TRUE, FALSE, FALSE, TRUE)]  
  
# specify the indices to discard  
x[-c(2,3)]

The second approach leads to the so called “conditional selection” technique as follows.

x <- 1:4  
x >= 3 # componentwise comparison resulting in a logical vector  
x[x >= 3]  
x[x >=1 & x <=3]

Matrix indexing follows the same manner.

x <- matrix(1:12, 3, 4)  
x[c(1,3),-c(1,4)]  
x[c(TRUE,FALSE,TRUE),]

**Exercise 3.** *(4 pt)*: Consider the matrix X from Exercise 2.

* Make a new vector y1 consisting of all the elements of X which are negative (strictly less than zero).

y1 <- YOUR CODE HERE

* Make a new vector y2 consisting of all the elements of X which are at strictly positive but less than 2.

y2 <- YOUR CODE HERE

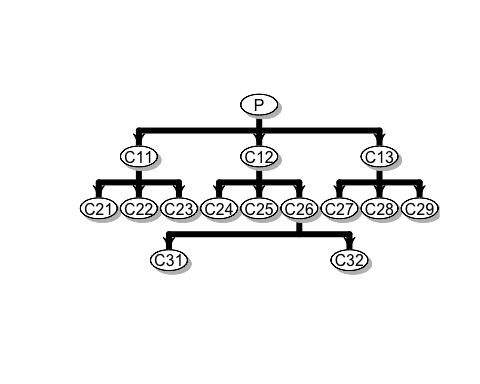
## List

List is a more flexible container of “data” that permits inhomogeneous types. It’s useful if you would like to encapsulate a bunch of components in an object. The list function explicitly specifies a list and the combining function c is still applicable. For example,

x <- list( num = 1:4, # "num =" specifies the name of the first component  
 chac = "hello world!",  
 logic = c(TRUE,FALSE),  
 nu = NULL,  
 mat = matrix(4:1, 2, 2) ) #text to the left of = specifies the component name  
y <- list( 1234,  
 "world" )  
c(x, y)

To extract the components in a list, one should use double bracket [[ ]] instead of a single bracket. If you’ve already specified the component names in a list, then the component names can be placed into the bracket directly. For example, x[["logic"]] accesses the third component of x. A more convenient alternative is x$logic.

**Punchline** Only ONE index instead of a vector of indices can be placed into the double bracket! Explore in the following example to see the difference as compared to the single bracket indexing.



C11 <- list( C21 = "C21",  
 C22 = "C22",   
 C23 = "C23")  
C26 <- list( C31 = "C31",  
 C32 = "C32")  
C13 <- list( C27 = "C27",  
 C28 = "C28",  
 C29 = "C29")  
C12 <- list( C24 = "C24",  
 C25 = "C25",  
 C26 = C26)  
P <- list( C11 = C11,  
 C12 = C12,  
 C13 = C13)  
  
# subtree rooted at C12  
P[[2]]  
P$C12  
  
# subtree (leaf) rooted at C24  
P[[c(2,1)]]  
P$C12$C24  
  
# subtree rooted at C26  
P[[c(2,3)]]  
P$C12$C26  
  
# subtree (leaf) rooted at C31  
P[[c(2,3,1)]]  
P$C12$C26$C31

## Date Frame

Inheriting from matrix and list, data.frame is a container general enough for us to study a dataset. It permits inhomogeneous data types across columns (components in a list) but forces the components of the list to be vectors of homogeneous length (so as to be columns in a matrix). For example, the following creates a score table of 3 students.

students <- data.frame( id = c("001", "002", "003"), # ids are characters  
 score\_A = c(95, 97, 90), # scores are numericss  
 score\_B = c(80, 75, 84))   
students

To access the score\_A of student 003, one can follow the manner in a matrix: students[3,2], or that in a list: students[[[2]][3], students[["score\_A"]][3] or students$score\_A[3].

**Exercise 4.** *(5 pt)* Applying the conditional selection technique (see the section “indexing” and do not use *subset*), extract the record of student 003 i.e their id number, and their scores in the two tests.

YOUR CODE HERE

One can also create a matrix or a legitimate list first and then convert it into a data.frame as follows.

scores <- matrix(c(95, 97, 90, 80, 75, 84), 3, 2)  
scores <- data.frame(scores)  
colnames(scores) <- c("score\_A", "score\_B")  
id <- c("001", "002", "003")  
students1 <- cbind(id, scores)  
students2 <- data.frame( list( id = c("001", "002", "003"),  
 score\_A = c(95, 97, 90),  
 score\_B = c(80, 75, 84))   
 )

**Exercise 5.** *(10 pt)* Create a data.frame object to display the calendar for Jan 2018 as follows.

## Sun Mon Tue Wed Thu Fri Sat  
## NY 2 3 4 5 6  
## 7 8 9 10 11 12 13  
## 14 MLK 16 17 18 19 20  
## 21 22 23 24 25 26 27  
## 28 29 30 31   
YOUR CODE HERE

Ignore the ## symbols this was just so the above acts like a comment in R.

1) The character object "" for the spaces; 2) the option row.names = FALSE in print function.

## Factors

Factor is a special data structure in **R** in representing categorical variables and facilitating the data labels and subgroups. It’s basically a character vector that keeps track of its distinct values called levels. Consider the longitudinal layout of the previous score table.

id <- rep(c("001","002","003"), 2)  
subj <- rep(c("A","B"), each = 3)  
score <- c(95, 97, 90, 80, 75, 84)  
students3 <- data.frame(id, subj, score) # try cbind(id, subj, score) to see the difference  
  
# students3$id and students3$subj are automatically formatted as factors  
class(students3$id)  
levels(students3$id)  
  
class(students3$subj)  
levels(students3$subj)  
  
# combind student 003 with 002 via level rename  
students4 <- students3 # work on a copy in case of direct modification of students3  
levels(students4$id)[3] <- "002"  
levels(students4$id)  
students4

The factor function applied to a character vector creates a natural factor. The gl and cut functions are also useful approaches to patterned factors that are generated from numeric variables.

**Exercise 6.** *(5 pt)* Create a factor variable grade in students3, where the score variable is divided into , and corresponding to A, B and C in grade respectively.

**Hint.** Functions cut to obtain the grades and transform to obtain the students5 from stuents3.

students5 <- YOUR CODE HERE

## Operations and Functions

Note that scalar operations on vectors usually apply componentwise.

* **Arithmetic operations:** +, -, \*, /, ^, %/% (exact division), %% (modulus), sqrt(), exp(), log()
* **Logical operations**
  + And &, &&; or |, ||; not !
  + Comparisons: <, <=, >, >=, == (different from =), !=
  + Summary functions: all(), any()
* **Summary statistics:** length, max, min, sum, prod, mean, var, sd, median, quantile
* **Matrix operations**
  + Matrix multiplication: %\*% (different from \*)
  + Related functions: t, solve, det, diag, eigen, svd, qr
  + Marginal operations: apply
* **Factors:** tapply (to apply operations/functions grouped by a factor)

**Exercise 7.** *(10 pt)* Without using the var and scale functions, compute the sample mean and sample covariance X.var of the data matrix X as in **Exercise 2.** More precisely, think of the -th row of the matrix as observation of features for -th individual.

**a** Create a 4-dimensional vector called mu where the -th row is the mean of the -th column of .

YOUR CODE HERE

**b** Create a four-dimensional matrix X.var

where is the -th row.

YOUR CODE HERE

**Exercise 8.** *(10 pt)* Imagine that we wanted to make students aware for each of their subjects, the average score of all other students in that subject. Create a variable (or column) called score.mean in students3, where next to each student and subject, the value of the score.mean is the average value of all students taking that subject.

YOUR CODE HERE

### Writing your own functions

It’s convenient to create a user-defined **R** function, where you might encapsulate a standard, complicated or tedious procedure/algorithm in a “black box” like any built-in functions as mentioned above, so that other users might only need to care about the input and output of your function regardless the details. See the following toy example.

my.fun <- function(x, y)  
{  
 # This function takes x and y as input  
 # It returns a list that contains the mean of x, y respectively and their difference  
 mean.x <- mean(x)  
 mean.y <- mean(y)  
 mean.diff <- mean.x - mean.y  
 output <- list(mean\_of\_x = mean.x,  
 mean\_of\_y = mean.y,   
 mean\_diff = mean.diff)  
 return(output)  
}  
x <- runif(50, 0, 1) # simulate 50 numbers from U[0,1]   
y <- runif(50, 0, 3) # simulate 50 numbers from U[0,3]  
my.fun(x, y)

## Flow Control

To help you write down your own **R** programs, the following examples familiarize you with the conditional statements and loops.

### Conditional Statements

mymax.if <- function(x, y)  
{  
 # The mymax function returns the larger one among x and y  
 if (x > y)  
 {  
 message("The first argument is larger!")  
 return(x)  
 }  
 else if (x < y)  
 {  
 message("The second argument is larger!")  
 return(y)  
 }  
 else  
 {  
 message("Equal!")  
 return(x)  
 }  
}  
mymax.if(3, 4)  
mymax.if(4, 3)  
mymax.if(3, 3)

### Loops

factorial <- function(n)  
{  
 # It computes the factorial of the natural number n  
 if(n == 0) return(1) else if(n < 0 | n != as.integer(n)) stop("Please input a natural number!")  
 n.fac <- 1  
   
 # for loops  
 for(i in 1:n)  
 n.fac <- i \* n.fac  
   
 # while loops  
 ## i <- 1  
 ## while(i <= n)  
 ## {  
 ## n.fac <- i \* n.fac  
 ## i <- i + 1  
 ## }  
   
 # repeat loops  
 ## i <- 1  
 ## repeat  
 ## {  
 ## n.fac <- i \* n.fac  
 ## i <- i + 1  
 ## if (i > n) break # attention to the usage of break  
 ## }  
  
 return(n.fac)  
}  
  
double.factorial <- function(n)  
{  
 # It computes the double factorial of the natural number n  
 if(n == 0) return(1) else if(n < 0 | n != as.integer(n)) stop("Please input a natural number!")  
 n.doubfac <- 1  
 for(i in 1:n)  
 if(i %% 2 == n %% 2)  
 n.doubfac <- i \* n.doubfac  
 else next # attention to the usage of next  
 return(n.doubfac)  
}  
  
factorial(6)  
prod(1:6)  
double.factorial(6)  
double.factorial(5)  
factorial(0)  
## factorial(6.5)  
## factorial(-6)

**Exercise 9.** *(15 pt)* The bisection method if a root-finding algorithm from numerical analysis to find a root of a continuous function in an interval once you know that function has different signs at the end points of the interval (i.e. or vice-versa). Read about this in the [Wikipedia link](https://en.wikipedia.org/wiki/Bisection_method).

Write a function bisect(f, lower, upper, tol = 1e-6) to find the root of the univariate function f on the interval [lower, upper] with precision tolerance tol (defaulted to be ) via bisection, which returns a list consisting of root, f.root (f evaluated at root), iter (number of iterations) and estim.prec (estimated precision). Apply it to the function

on with precision tolerance . Compare it with the built-in function uniroot.

YOUR CODE HERE

**Brief discussion of tolerance:** For tolerance: please read the Wikipedia article on the bisection method (the HW has a link or):

<https://en.wikipedia.org/wiki/Bisection_method>

In particular if you read the iteration steps carefully from Wikipedia:

Part(c) says:

" 3. If convergence is satisfactory (that is, c - a is sufficiently small, or |f(c)| is sufficiently small), return c and stop iterating."

This sufficiently small is the tolerance (default value according to HW exercise problem = 10^-6). See the pseudocode in the Wikipedia link. The tolerance shows up as “TOL”. Further when you stop iterating, the final value of = estimated precision.

## Input/Output

scan and read.table are two main functions to read data. The main difference of them lies in that scan reads one component (also called “field”) at a time but read.table reads one line at a time. Hence read.table requires the data to be well-structured as a table so as to create a data.frame in **R** automatically, while scan can be flexible but might require effort in manipulating data after reading. Their usages are quite similar. One should pay attention to the frequently used options file, header, sep, dec, skip, nmax, nlines and nrows in their **R** documents.

write.table is a converse function against read.table, while their usages are almost identical. To get familiar with thier features, explore in the next exercise.

To read inline, one can specify file = stdin() (or omitted in scan function). In that case, it reads from console that the user can input line-by-line, or from the subsequent lines in a program script, until an empty line is read. However, such a trick is NOT compatible in **RMarkdown**.

If you have your data stored in another format, *e.g.* EXCEL or SAS dataset, then you can output it this as a CSV file and read in **R** via read.csv function (almost identical to read.table).

**Exercise 10** *(16 pt)* In the folder for HW 1, you can find data on UNC salaries as a unc\_salary\_data.csv file (all of which are publicly available and scraped by Ryan Thornburg).

**a** Read the data using read.csv into a data frame called salaries

YOUR CODE HERE

Use str(salaries) and head(salaries) to get an idea of the data set.

**b** Make a new data frame called relevant consisting only of the columns: name, dept, age,totalsal. (Hint: consider the subset function).

YOUR CODE HERE

**c** Make a new data frame called top\_200 consisting of the information in relevant of faculty who make more than $200,000.

YOUR CODE HERE

**d** Choose 3 departments that you are interested in. Compute the average salary of faculty in these 3 departments.

YOUR CODE HERE

# Probability and Distributions

This section explores how to create “randomness” in **R** and obtain probabilistic quantities.

## Discrete Random Sampling

Much of the earliest work in probability theory starts with random sampling, *e.g.* from a well-shuffled pack of cards or a well-stirred urn. The sample function applies such procedure to a vector in **R**. Learn more from the **R** documents.

The following exercise means to create a five-fold cross-validating sets, which would be the starting point to assess the performance of a learned machine in, for example, classification errors.

**Exercise 11.** *(10 pt)* iris is a built-in dataset in **R**. Check ?iris for more information. This dataset has data on 50 flowers each from 3 species of Iris (setosa, versicolor, and virginica). Randomly divide iris into five subsets iris1 to iris5 (without replacement), thus each subset has 30 rows of the iris data and further stratified to iris$Species (namely every subset should have 10 rows from each of the 3 species).

YOUR CODE HERE  
iris.5fold <- list(iris1, iris2, iris3, iris4, iris5)

## Distributions

**R** is endowed with a set of statistical tables. To obtain the density function, cumulative distribution function (CDF), quantile (inverse CDF) and pseudo-random numbers from a specific distribution, one only needs to prefix the distribution name given below by d, p, q and r respectively.

|  |  |  |
| --- | --- | --- |
| **Distributions** | **R Names** | **Key Arguments** |
| Uniform | unif | min, max |
| Normal | norm | mean, sd |
|  | chisq | df, ncp |
| Student’s t | t | df, ncp |
| F | f | df1, df2, ncp |
| Exponential | exp | rate |
| Gamma | gamma | shape, scale |
| Beta | beta | shape1, shape2, ncp |
| Logistic | logis | location, scale |
| Binomial | binom | size, prob |
| Poisson | pois | lambda |
| Geometric | geom | prob |
| Hypergeometric | hyper | m, n, k |
| Negative Binomial | nbinom | size, prob |

Check from their plots.

plot(dnorm, xlim = c(-5, 5)) # bell curve of Normal density  
plot(plogis, xlim = c(-5, 5)) # Logistic/Sigmoid function (CDF of Logistic distribution)

The following two-sample t-test shows the usages of qt, pt and rnorm. Recall that a two-sample homoscedastic t-test statistic is

$$ \hat{\sigma}^2 = {(n\_X - 1)S\_X^2 + (n\_Y - 1)S\_Y^2 \over n\_X + n\_Y -2}, \quad T = {\bar{X} - \bar{Y} \over \hat{\sigma}\sqrt{ {1 \over n\_X} + {1 \over n\_Y}}} \stackrel{d}{\sim} t\_{n\_X + n\_Y - 2} \text{ under } H\_0:\ \mu\_X = \mu\_Y.$$

twosam <- function(x, y, alpha = 0.05)  
{  
 # It conducts a two-sample homoscedastic t-test on x and y  
 n.x <- length(x); n.y <- length(y)  
 mean.x <- mean(x); mean.y <- mean(y)  
 var.x <- var(x); var.y <- var(y)  
 mean.diff <- mean.x - mean.y  
 df <- n.x + n.y - 2  
 sigma <- ((n.x - 1) \* var.x + (n.y - 1) \* var.y) / df  
 var.diff <- (1/n.x + 1/n.y) \* sigma  
 t <- mean.diff / sqrt(var.diff)  
 t.alpha <- qt(1 - alpha/2, df)  
 output <- list(t = t,  
 df = df,  
 p.value = 2 \* pt(-abs(t), df),  
 confint = c(lower = mean.diff - sqrt(var.diff) \* t.alpha,  
 upper = mean.diff + sqrt(var.diff) \* t.alpha),  
 mu = c(mu.x = mean.x, mu.y = mean.y),  
 sigma = sigma)  
 return(output)  
}  
x1 <- rnorm(40, 0, 1)  
x2 <- rnorm(50, 0, 1)  
x3 <- rnorm(50, 1, 1)  
twosam(x1, x2)  
t.test(x1, x2, var.equal = TRUE)  
twosam(x1, x3)  
t.test(x1, x3, var.equal = TRUE)

# Data Exploration and Manipulation

Data analysis in **R** starts with reading data in a data.frame object via scan and read.table as discussed before. Then one would explore the profiles of data via various descriptive statistics whose usages are also introduced in the previous sections. Calling the summary function with a data.frame input also provides appropriate summaries, *e.g.* means and quantiles for numeric variables (columns) and frequencies for factor variables.

This section explores more features that can be achieved through **R**.

## Tables

Statistician often works with catergorical variables via tables. Even for continuous variables, segregating them into catergorical ones in a meaningful way might provide more insights. table function generates frequency tables for factor variables. Multi-way tables, marginal and proportional displays and independence test are explored in the following example. Recall that the Pearson’s independence test statistic on an -by- contingency table

$$\chi^2 = \sum\_{i=1}^r\sum\_{j=1}^c{\left({n\_{ij} - {n\_{i\cdot}n\_{\cdot j} \over n\_{\cdot\cdot}} }\right)^2 \over {n\_{i\cdot}n\_{\cdot j}/ n\_{\cdot\cdot}} } \stackrel{d}{\approx} \chi^2\_{(r-1)(c-1)} \text{ under } H\_0: p\_{ij} = p\_i p\_j.$$

# Discretize the numeric variables  
iris0 <- transform(iris,  
 Sepal.Length = cut(Sepal.Length, 4:8),  
 Sepal.Width = cut(Sepal.Width, 1:5),  
 Petal.Length = cut(Petal.Length, 0:7),  
 Petal.Width = cut(Petal.Width, 0:3))  
  
attach(iris0) # attach function makes columns assessible as usual variables  
table(Sepal.Width)  
iris.table0 <- table(Petal.Width)  
iris.table0  
iris.table1 <- table(Sepal.Width, Petal.Width)  
iris.table1  
iris.table2 <- table(Sepal.Width, Petal.Width, Species)  
iris.table2  
  
# flat table in a sligtly different display  
ftable(Sepal.Width, Petal.Width)  
ftable(Species, Sepal.Width, Petal.Width)  
  
# data.frame display  
data.frame(iris.table1)  
data.frame(iris.table2)  
  
# marginal display  
margin.table(iris.table1, 1)  
margin.table(iris.table1, 2)  
  
# proportional display relative to the (margin) total  
prop.table(iris.table1)  
prop.table(iris.table1, 1)  
prop.table(iris.table1, 2)  
prop.table(iris.table2, 1)  
prop.table(iris.table2, c(1,3))  
  
# conduct chi-square independence test  
chisq.test(iris.table1) # warning message due to some frequency entry <= 5  
  
# plots  
pie(iris.table0, main="Petal.Width")  
barplot(t(iris.table1), beside = TRUE,   
 xlab = "Sepal.Width", ylab = "Petal.Width", legend.text = colnames(iris.table1))  
  
detach(iris0) # paired with attach function

## Plots

Compared to other statistical softwares in data analysis, **R** is good at graphic generation and manipulation. The plotting functions in **R** can be classified into the high-level ones and the low-level ones. The high-level functions create complete, new plots on the graphics device while the low-level functions only add extra information to the current plots.

plot is the most generic high-level plotting function in **R**. It will be compatble with most class of objects that the user input and produce appropriate graphics. For example, a numeric vector input results in a scatter plot with respect to its index and a factor vector results in a bar plot of its frequency table. Advanced class of object like lm (fitted result by a linear model) can also be called in plot. Methods will be discussed in specific documents like ?plot.lm.

Other plotting features include

* High-level plotting options: type, main, sub, xlab, ylab, xlim, ylim
* Low-level plotting functions
  + **Symbols:** points, lines, text, abline, segments, arrows, rect, polygon
  + **Decorations:** title, legend, axis
* Environmental graphic options (?par)
  + **Symbols and texts:** pch, cex, col, font
  + **Lines:** lty, lwd
  + **Axes:** tck, tcl, xaxt, yaxt
  + **Windows:** mfcol, mfrow, mar, new
* User interaction: location

I’ll suggest the beginners learn from examples and grab when needed instead of going over such an overwhelming brochure. The following sections illustrate two basic senarios in data analysis. More high-level plotting functions will also be introduced.

**Exercise 12** *(10 pt)*

**a** Recall the UNC salary data set. From the salaries data frame plot the number of CS faculty hired per year vs year.

YOUR CODE HERE

**b** Now add STOR, Math and Physics to the above plot

YOUR CODE HERE

### Access the empirical distribution

**Histogram:** The hist function creates a histogram in **R**, where the breaks and freq options are frequently called. The barplot function could also realize the same result as illustrated in section **Tables**.

**Kernel Density Curve:** The density function estimates an empirical density of the data and gives a density object. One can call plot function with such an object as input and picture a density curve, which is anticipated to closely envelop its historgram. Note that the bw (bandwidth) and kernel options should be carefully considered in methodology.

**Empirical CDF (ECDF):** The ecdf function generates an empirical CDF ("ecdf") object that can be called by the plot function, which results in a step function graphic for the empirical cumulative distribution function. Creating a graph containing multiple CDFs or ECDFs visually displays the good-of-fitness or discrepancy among them. The statistically quantified Kolmogorov-Smirnov test with statistic

realized by the ks.test function is based on it.

**Q-Q Plots:** “Q-Q” stands for the sample quantiles versus that of a given distribution or another sample. qqnorm, qqline and qqplot together is the set of functions in realizing it. **R** documents also illustrate how to create Q-Q plots against distributions other than Normal.

**Box-and-Whisker Plots:** With boxplot function in **R**, we can describe the data with box associated with certain quantiles and the whiskers for extremes. The box in the middle indicates “hinges” (nearly quartiles, see ?boxplot.stats) and median. The lines (“whiskers”) show the largest/smallest observation that falls within a distance of 1.5 times the box size from the nearest hinge. If any observations fall farther away, the additional points are considered “extreme” values and are shown separately.

**Exercise 13.** *(15 pt)* The following code generates the ensuing plot about Sepal.Length in iris.

# opar <- par(mfrow = c(1,3))  
# for(l in levels(iris$Species))  
# {  
# Sepal\_Length <- subset(iris, Species == l, select = Sepal.Length)[[1]]  
# h <- hist(Sepal\_Length, sub = paste("Species =", l), ylim = c(0,1.3), freq = FALSE)  
#   
# par(new = TRUE) # add to the current plot  
# # Empirical density curve  
# lines(density(Sepal\_Length),  
# xlim = range(h$breaks), ylim = c(0,1.3), # to match the plotting range  
# col = "blue",   
# main = "", sub = "", xlab = "", ylab = "" # to supress labels  
# )  
# par(new = TRUE) # add to the current plot  
#   
# # Normal density curve  
# curve(dnorm(x, mean = mean(Sepal\_Length), sd = sd(Sepal\_Length)),   
# xlim = range(h$breaks), ylim = c(0,1.3), # to match the plotting range  
# col = "red",  
# main = "", sub = "", xlab = "", ylab = "" # to supress labels  
# )  
#   
# legend("topright",   
# legend = c("Kernel Density", "Normal Density"),   
# col = c("blue", "red"), lty = 1, cex = 0.5)  
# }  
# par(opar)

**Plot generated**:

![](data:application/pdf;base64,)

Either modify the code above or use your own code to obtain similar plots with histograms, kernel density plots and normal density plots for the salary of faculty in CS, Math and Physics from UNC salary data.

YOUR CODE HERE

### Demostrate correlation

plot(x, y) directly creates a scatter plot between the vector x and y. For a data.frame input X, plot(X) would conduct pairwise scatter plots among its columns. A fitted regression line can also be added to an existing scatter plot via the abline function. And the function coplot is a power function in creating conditioning plots given a variable segregated into different levels.

plot(iris[, -5])  
  
attach(iris) # attach function makes columns assessible as usual variables  
lm.mod <- lm(Petal.Length ~ Sepal.Length) # fit a linear model as an lm object  
plot(Sepal.Length, Petal.Length)  
## plot(Petal.Length ~ Sepal.Length)  
abline(lm.mod)  
  
coplot(Petal.Length ~ Sepal.Length | Species)  
detach(iris) # paired with attach function

### \*Create advanced plots

If you are a JAVA programmer, then you might anticipate a plotting toolbox to establish graphs layer-by-layer interactively. The ggplot2 package endows **R** with more advanced and powerful visualization techniques like this. Explore more in [the online package manual](https://cran.r-project.org/web/packages/ggplot2/ggplot2.pdf). The following example solves **Exercise 11** without segregating with respect to Species in iris.

mu <- mean(iris$Sepal.Length)  
sigma <- sd(iris$Sepal.Length)  
ggplot(iris, aes(Sepal.Length)) +  
 geom\_histogram( aes(y = ..density..),  
 bins = 8, color = "black", fill = "white") +  
 geom\_density(aes(color = "blue")) +  
 stat\_function( aes(color = "red"),   
 fun = dnorm, args = list(mean = mu, sd = sigma)) +  
 labs(title = "Histogram of Sepal\_Length") +  
 scale\_color\_identity( name = "Density Estimate",  
 guide = "legend",  
 labels = c("Kernel", "Normal")) +  
 theme\_bw()

## \*Data Manipulation

If you are more familiar with the SQL language in manimulating data, then the dplyr package in **R** is a powerful toolkit in providing functions similar to the SQL operations (*e.g.* select, filter, arrange, mutate, inner\_join, group\_by, summarise and the pipe operator %>%). The following example realizes the conditional selection technique in the previous exercises in a much cleaner way. Explore more in [the online package mannual](https://cran.r-project.org/web/packages/dplyr/dplyr.pdf).

# Exercise 3.  
filter(students, id == '003')  
  
# Exercise 7.  
students6 <- group\_by(students3, subj)

## Warning: The `printer` argument is deprecated as of rlang 0.3.0.  
## This warning is displayed once per session.

mutate(students6, score.mean = mean(score))  
  
# Using the pipe operator makes it more compact  
group\_by(students3, subj) %>%   
 mutate(score.mean = mean(score)) # transform function couldn't apply mean by group

# Bibliography

Dalgaard, Peter. 2008. *Introductory Statistics with R: Statistics and Computing*. Springer.

W. N. Venables, D. M. Smith, and the R Core Team. 2017. *An Introduction to R: Notes on R: A Programming Environment for Data Analysis and Graphics* (version 3.4.3).