R Workshop for Beginners

Dr. Zhiqiang Zhang

Department of Statistics and Actuarial Science,
The University of Hong Kong

August 10, 2023

Table of Contents

Chapter 1 Introduction to R and RStudio

§1.1 Introduction to R

- **R** is an *integrated suite of software* facilities for data manipulation, calculation and graphical display. Among other things it has
    - an effective data handling and storage facility,
    - a suite of operators for calculations on *arrays*, in particular *matrices*,
    - a large, coherent, integrated collection of intermediate tools for data analysis,
    - graphical facilities for data analysis and display either directly at the computer or on hard copy, and
    - a well developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities. Indeed most of the system supplied functions are themselves written in the *S language*.

Introduction to R

- **S**: an interactive environment for data analysis developed at Bell Laboratories since 1976.
  - 1988 - S2: RA Becker, JM Chambers, AWilks.
  - 1992 - S3: JM Chambers, TJ Hastie.
  - 1998 - S4: JM Chambers.
- Exclusively licensed by AT&T/Lucent to Insightful Corporation, Seattle WA. Product name: "**S-plus**".
- **R**: initially written by **R**oss Ihaka and **R**obert Gentleman at Dept. of Statistics in Univ. of Auckland, New Zealand during 1990s.
  - *We have named our language R — in part to acknowledge the influence of S and in part to celebrate our own efforts.*
- Since 1997: international R-core team 20 people & 1000s of code writers and statisticians happy to share their **libraries**!

Introduction to R

- R is an *interpreted computer language*.
    - Most user-visible functions are written in R itself, calling upon a smaller set of internal primitives.
    - It is possible to interface procedures written in C, C+, or FORTRAN languages for efficiency, and to write additional primitives.
    - System commands can be called from within R.
- Implementation languages: C, Fortran.
    - You need not install C or Fortran in advance!
- R programs can be run on the framework of Python too.
- However, most programs written in R are essentially *ephemeral*, written for pieces of data analysis, available as **R packages**.
    - The same for Python.

Introduction to R

- Most people come to R because of
  - its *rapidly developed tools* for statistical analysis,
  - drawing *beautiful plots*, and
  - *free* of charge.
- Top 20 programming languages by FOSSBYTES[1]:

Programming Language	Mar.2020		Oct.2020		Change in Rating
	Rank	Rating	Rank	Rating	
Java	1	17.78%	2	12.56%	−5.22%
C	2	16.33%	1	16.95%	+0.62%
Python	3	10.11%	3	11.28%	+1.17%
C++	4	6.79%	4	6.94%	+0.15%
⋮	⋮	⋮	⋮	⋮	⋮
R	11	1.26%	9	1.99%	+0.73%
⋮	⋮	⋮	⋮	⋮	⋮

[1] https://fossbytes.com/most-popular-programming-languages/

Why R? Career Opportunity & Drawbacks

- **Python**:
  - **Drawbacks**: Not suitable for mobile application development.[2] Different packages need different versions of Python.
  - **Career Opportunity**: Software Engineer, Software Developer, Web Developer, Quality Assurance Engineer, Data Science.[1]
  - **Average annual salary**: \$120,000[2]
- **R**:
  - **Drawbacks**: Does not have the strict programming guidelines of older and more established languages.[2]
  - **Career Opportunity**: Data Scientist, Business Analyst, Big Data Engineer.[1]
  - **Average annual salary**: \$93,000[2]

[2] https://www.northeastern.edu/graduate/blog/most-popular-programming-language

Introduction to R Studio

§1.2 Introduction to R Studio

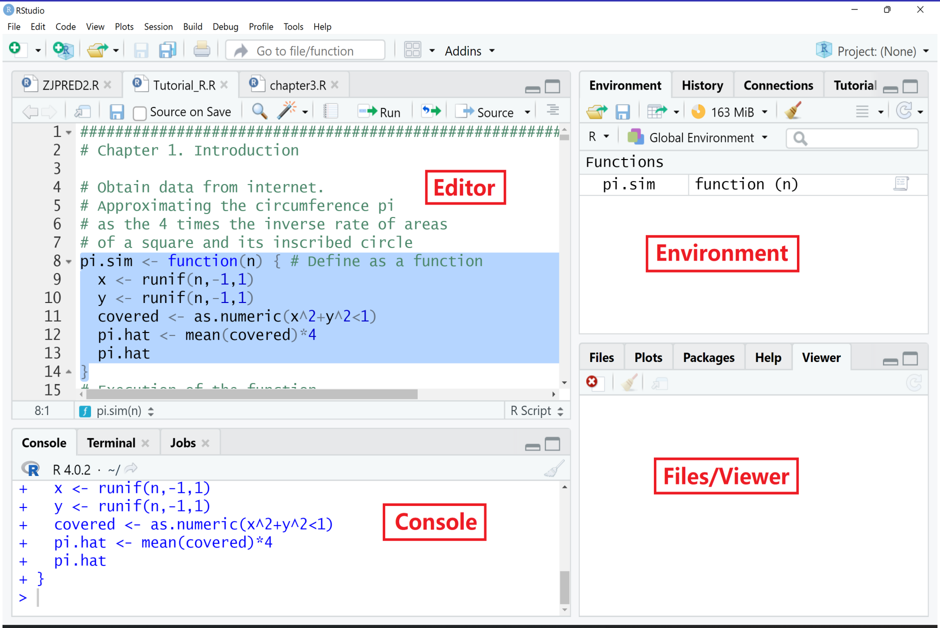
- Nowadays (since around February 2011), R is frequently implemented with another software called "**R Studio**" or "**RStudio**", an active member of the *R community*.
- **RStudio** is a free and open-source (charged for commercial license) Integrated Development Environment (IDE) for R, a programming language for statistical computing and graphics.
- **RStudio** is available in two editions: **RStudio Desktop**, and **RStudio Server**.
- **RStudio** is partly written in the C++ programming language and uses the Qt framework for its graphical user interface. The bigger percentage of the code is written in Java.
- R Version 4.3.1 and RStudio Version Version 2023.06.1+524 are available for online downloading.

Getting start

§1.3 Getting start

- Download **R** from `https://www.r-project.org/` and install.
- Download **RStudio** from `https://www.rstudio.com/` and install.
- Double click, the icon to start RStudio and R.
- Double click (or, open with RStudio) either of the following types of files to start.
  - an R script (R program) file "xxxx.R",
  - an R markdown (R program and report) file "xxxx.Rmd",
  - an R environment (workspace) file "xxxx.RData", or
  - an R history file "xxxx.RHistory".
- See the interface of RStudio (for Windows) in the next page.

Getting start



```
#################################################
# Chapter 1. Introduction

# Obtain data from internet.
# Approximating the circumference pi
# as the 4 times the inverse rate of areas
# of a square and its inscribed circle
pi.sim <- function(n) { # Define as a function
  x <- runif(n,-1,1)
  y <- runif(n,-1,1)
  covered <- as.numeric(x^2+y^2<1)
  pi.hat <- mean(covered)*4
  pi.hat
}
# Execution of the function
```

```
+   x <- runif(n,-1,1)
+   y <- runif(n,-1,1)
+   covered <- as.numeric(x^2+y^2<1)
+   pi.hat <- mean(covered)*4
+   pi.hat
+ }
>
```

Basic manipulations

- Main Menu:
  - File: New File ▶ R Script, Open File, Save as...
  - Edit: Undo, Redo, Cut, Copy, Paste, $\cdots$
  - Tools: Install Packages, $\cdots$
  - Help: R Help, About RStudio, $\cdots$
- Editor:
  - Edit (open, save, modify, $\cdots$) R programs and/or reports (xxxx.R or xxxx.Rmd files).
  - Run R scripts, or parts of them.
  - Display R objects such as data frames, functions, etc.
  - Display running results (for R markdown only).

Basic manipulations

- Console:
  - Console: display R commands ever run, key in (temporary) R commands and run, display running results, etc. Items are displayed in the order of their occurrences.
  - Terminal: the work directory.
- Environment:
  - Environment: display R objects such as the data frames, estimates, calculated values, defined functions, etc; import data sets from Excel, SPSS, etc. (Possibly need installing some packages.)
  - History: history of commands ever run.
- Files/Viewer:
  - Files: files in the work directory.
  - Plots: display and work with plots.
  - Packages: display and/or install R packages.
  - Help: help information.

Useful online resources

- Reference: An Introduction to R

  `https://cran.r-project.org/doc/manuals/R-intro.html`

- R packages:

  `https://cran.r-project.org/web/packages/available_packages_by_name.html`

- Free data sets:

  `https://r-dir.com/reference/datasets.html`

Training exercises

- Open the R script `Tutorial_R.R`.
  - Select the piece of commands in “Chapter 1” to run.
  - Click to view the R functions `pi.sim` and `pi.approx` in the environment.
  - Save the environment as `Tutorial_R.RData`.
  - Save the history as `Tutorial_R.RHistory`.
  - Quit R by keying in `q()` in the console.
  - Restart R through opening (with R Studio) the saved R environment file.
  - Restart R through opening (with R Studio) the saved R history file.

General rules for commands

Chapter 2 R Commands and Objects

§2.1 R Commands

- Commands and comments can be input in either the “Editor” (as part of the program) or the “Console” (as part of the history).
- **Case Sensitive !** ‘X’ and ‘x’ are two different objects.
- Elementary commands consist of either expressions or assignments.
- Commands are separated either by a semi-colon ‘;’, or by a new line.
- Elementary commands can be grouped by braces ‘{’ and ‘}’.
- Comments start with a hash mark ‘#’.

General rules for commands

- A single command can be written in more than one line. R will automatically check the logic of the command.
- In the Console, if a plus mark '+' appears as the command prompt, then the command is not complete.
- Try the following commands in the Console, line by line, excluding the command prompts '>' or '+' in blue.

```
> x=2*(3+5); x
> y=2*(3
+ +5
+ )
> y
```

- The vertical arrow keys ▲ and ▼ on the keyboard can be used (when cursor is) in the Console, to recall and/or correct previous commands in the history.

Use packages and help

- Use installed packages:

  > library(*library_name*), or, require(*library_name*)

- Use help:

  > ?*command_name*, e.g., ?bptest.

  > help(*command_name*), e.g., help(bptest).

  > ??*command_name*, e.g., ??bptest.

  - Some commands automatically popped up when keying help, e.g., help.start().
    - Keying in "help.start" without "()" gives the definition (an R program) of the function help.start in Console. "View(help.start)" gives the same output in Editor.

- Help results appeared in "Files":
  - Help pages: lmtest :: bptest means function bptest in package lmtest.
  - Click lmtest :: bptest to see the R Documentation for the function.

R objects

§2.2 Create R objects

- **R objects**: variables (values, vectors, matrices), data frames, functions, fitted results, etc.
- During an R session, objects are created and stored *temporally* in the **Environment**. The R command

  ```
  > objects()
    [1] "pi.sim"
  ```

  (alternatively, ls()) can be used to display the names of the objects which are currently stored within R. The collection of objects currently stored is called the workspace.
- To remove objects the function `rm` is available:

  ```
  > rm(pi.sim)
  > ls()
    character(0)
  ```

Values of variables

- We introduce R variables first.
- Values of variables:
    - numeric: a real or complex number, or a level in a factor;
    - character: a string of letters;
    - logical: TRUE or FALSE;
    - missing: NA for numeric, "" or NA for character;
    - Inf: infinite (numeric); and
    - NaN: undefined (numeric).

Define empty variables

- Define a (an empty) numerical variable y1:

```
> y1 <- numeric(); y1
numeric(0)
```

- Define an empty character vector y2 of dimension 5:

```
> y2 <- character(5); y2
[1] "" "" "" "" ""
```

- Define an empty logical vector y3 of dimension 2:

```
> y3 <- logical(2); y3
[1] FALSE  FALSE
```

- Define an empty matrix A (numerical) of orders 2×3:

```
> A <- matrix(nrow = 2, ncol = 3); A
      [,1]  [,2]  [,3]
[1,]  NA    NA    NA
[2,]  NA    NA    NA
```

Two more examples

- Privileges: character > numeric > logic.
- An example of character variable/vector.

```
> c(1:3, "String", NA, 0/0, 1/0, -1/0, TRUE, FALSE)
[1] "1" "2" "3" "String" NA "NaN" "Inf" "-Inf" "TRUE"
"FALSE"
```

- An example of numeric variable/vector.

```
> c(1:3, NA, 0/0, 1/0, -1/0, TRUE, FALSE)
[1] 1 2 3 NA NaN Inf -Inf 1 0
```

- R command `c(⋯, ⋯, ⋯)` is used to list/assign values (combine several values together) of a variable.

Assign and view values

§2.3 Assign values to variables

- Assign (and define) a value or values to a variable.
- Assign a numeric value to a variable:

```
> a <- 1; a
  [1] 1
> b = 2; b
  [1] 2
> a = b; a; b
  [1] 2
  [1] 2
> 1 -> a; a
  [1] 1
```

Assign and view values

- Assign a character value to a variable:

```
> c <- "MStat"; c
  [1] "MStat"
```

- Assign a logical value to a variable:

```
> d <- a>2; d
  [1] FALSE
```

- Assign a numerical vector (treated as a list of values, or a column vector):

```
> x1 <- c(4, 2, 0); x1
  [1] 4 2 0
> c(1, 0, -1) -> x2; x2
  [1] 1 0 -1
```

Assign values to vectors

- Assign a character vector:

```
> assign("x3", c("A", "B", "C")); x3
  [1] "A" "B" "C"
```

- Assign a logical vector:

```
> x4 <- x1>=2; x4
  [1] TRUE TRUE FALSE
```

- Assign a factor vector:

```
> f <- factor(c("A", "B", "C", "A")); f
  [1] A B C A
  Levels: A B C
```

Assign a complex variable or vector

- Assign a complex variable:

```
> c1 <- 1+2i; c1
  [1]  1+2i
> mode(c1)
  [1] "complex"
```

- Assign a complex vector:

```
> c2 <- c(1-2i, 4+2i); c2
  Error: object 'i' not found
> c2 <- complex(2, c(1,4), c(-1,2)); c2
  [1]  1-1i  4+2i
> mode(c2)
  [1] "complex"
```

Display/View assigned value(s)

- Display/View a variable in Editor:

```
> View(a)
> View(x1)
```

- Display (values of) variables in Console:

```
> x2[2]            # 2nd value of x2
[1] 0
> x3[3]            # 3rd value of x3
[1] "C"
> x1[c(1,3)]            # First and 3rd values of x1
[1] 4 0
```

More on assigning values to vectors

- Concatenate/Connect vectors/values into a new vector:

```
> x5 <- c(x1, 0, x2); x5
[1] 4 2 0 0 1 0 -1
```

- Generate regular sequences:

```
> x6 <- 1:6; x6
[1] 1 2 3 4 5 6

> seq(from=0, to=20, by=5)
[1] 0 5 10 15 20

> seq(from=2, by=-1, length=7)
[1] 2 1 0 -1 -2 -3 -4

> rep(x1, times=3)
[1] 4 2 0 4 2 0 4 2 0

> rep(x1, each=3)
[1] 4 4 4 2 2 2 0 0 0
```

Assign values to matrices

§2.4 Assign values to matrices

- Read x1 and x2 into a matrix as two columns:

```
> m1 <- matrix(c(x1, x2), nrow = 3, ncol = 2); m1
     [,1] [,2]
[1,]    4    1
[2,]    2    0
[3,]    0   -1
```

  - Remark: values of c(x1, x2) are read into the matrix by column.

- Combine x1 and x2 by columns (into a matrix):

```
> M1 <- cbind(x1, x2); M1
     x1   x2
[1,]  4    1
[2,]  2    0
[3,]  0   -1
```

Assign values to matrices

- Define a matrix with x1 and x2 as two rows:

```
> m2 <- matrix(c(x1, x2), nrow = 2, ncol = 3,
+                          byrow = TRUE); m2
       [,1] [,2] [,3]
[1,]     4    2    0
[2,]     1    0   -1
```

- Combine x1 and x2 by rows (into a matrix):

```
> M2 <- rbind(x1, x2); M2
     [,1] [,2] [,3]
x1     4    2    0
x2     1    0   -1
```

Assign values to matrices

- One more example of cbind() and rbind():

```
> cbind(0, rbind(1, 1:3))
     [,1] [,2] [,3] [,4]
[1,]    0    1    1    1
[2,]    0    1    2    3
```

View value(s) of a matrix

- Display row(s), column(s), or element(s) of a matrix:

```
> m1[1,]        # The 1st row of m1
[1] 4 1

> m1[,2]        # The 2nd column of m1
[1] 1 0 -1

> m1[1,2]        # The (1,2)th element of m1
[1] 1

> m1[2:3,1]        # Values in rows 2 & 3, column 1
[1] 2 0
```

Define a data frame

§2.5 Define a data frame

- Define a data frame (a data set) with two variables x1 and x2:

```
> data1 <- data.frame(x1, x2); View(data1)
> data2 <- data.frame(X=x1, Y=x2); View(data2)
```

  - Remark: x1 and x2 must have the same length (dimension).

- Add variable x3 to data frame data1:

```
> data1$x3 <- x3
> View(data1)
```

  - It is required that length(x3)=nrow(data1).

- View/Use variable(s) in a data frame:

```
> data1$x3          # Display variable x3 in data1
  [1] "A" "B" "C"
```

Modes of objects

§2.6 Modes, lengths and orders of objects

- Use model(*object_name*) to see modes (types) of objects.

```
> mode(x1)
  [1] "numeric"
> mode(x3)
  [1] "character"
> mode(x4)
  [1] "logical"
> mode(c)
  [1] "complex"
> mode(f)
  [1] "numeric"
> mode(m1)
  [1] "numeric"
> mode(data1)
  [1] "list"
```

Command length()

- Command Use length(*variable*) displays the length of the variable.

```
> length(x1)
  [1] 3
```

- Command Use length(*data frame*) displays the number of variables in the data frame.

```
> length(data2)
  [1] 2
```

- Command Use length(*matrix*) displays the number of elements in the matrix.

```
> length(m1)
  [1] 6
```

Commands nrow() and ncol()

- Use nrow(*object_ name*) and ncol(*object_ name*) to display the "orders" of an R matrix or data frame.

```
> nrow(m1)        # number of rows of a matrix
  [1] 3
> ncol(m1)        # number of columns of a matrix
  [1] 2
> nrow(data2)     # number of observations in the data frame
  [1] 3
> ncol(data2)     # number of variables in the data frame
  [1] 2
```

Change modes and types

- Change modes

```
> x1; as.character(x1)
  [1] 4 2 0
  [1] "4" "2" "0"
> x4; as.numeric(x4)
  [1] TRUE TRUE FALSE
  [1] 1 1 0
```

- Change types: run the following codes to see the change(s) in outputs.

```
> m2; as.data.frame(m2)
> data2; as.matrix(data2)
```

Define an array

§2.7 Define an array

- Vectors and matrices are 1-dimensional and 2-dimensional arrays of values, respectively. More general arrays can be defined by the `array()` function.
- The following R commands define and display a 3-dimensional array or orders $2 \times 3 \times 2$. Pay attention to the orders of numbers $\{a_{ijk}\}$ in the array.

```
> A3 <- array(1:12, dim = c(2,3,2)); A3
    , , 1
          [,1] [,2] [,3]
    [1,]    1    3    5
    [2,]    2    4    6
    , , 2
          [,1] [,2] [,3]
    [1,]    7    9   11
    [2,]    8   10   12
```

Define an array

- Assign the matrix `m2` as values of `A3[,,1]`

  ```
  A3[ , ,1] <- m2; A3[ , ,1]
  ```

- Assign numeric values to some cells of `A3`

  ```
  A3[ ,1,2] <- 0; A3
  ```

- Assign some character values

  ```
  A3[ ,2,2] <- "B"; A3
  ```

  - A3 becomes a character array.

- Display/Use values

  ```
  A3[2,c(1,3),1]
  ```

Training exercises

1. Define the following R vectors/lists:
    - (a) `TX1`: a list of positive integers no more than 50, with remainder 3 divided by 7, and in ascending order.
    - (b) `TX2`: a list of positive integers no more than 50, with remainder 3 divided by 7, and in descending order.
    - (c) `TX3`: a list of numbers which is the concatenation/connection of `TX1` and `TX2`.
    - (d) `TX4`: a list of numbers with the sequence `TX1` being repeated twice.
    - (e) `TX5`: a list of numbers with each value of `TX2` being repeated twice.
    - (f) `TX6`: a list of character values repeatedly taking values "F", "F", and "M", with the same length as that of `TX5`.
    - (g) `TX7`: a list of factor values, with the same "values" as those in `TX6`.

Training exercises

2. Define the following matrices in R:
   (a) `TM1`: a matrix with `TX1` and `TX2` as its two column vectors.
   (b) `TM2`: a matrix with `TX4` and `TX5` as its two row vectors.
3. Define the following data frames in R:
   (a) `TD1`: a data frame containing variables `TX4` and `TX5`, named as `V1` and `V2` respectively.
   (b) `TD2`: a data frame containing variables `V1` and `V2` in `TD1`, and variable `V3` taking values as `TX6`.
4. Find/Display the following values in R console:
   (a) The 2nd and 4th values of `TX1`.
   (b) The 3rd and 4th elements in the 2nd column of `TM1`.
   (c) The last two values of the last two variables in `TD2`.

Chapter 3 Simple Computations and Manipulations

§3.1 Computations of numerical lists

- Operators (scalar, arithmetic) : "+", "−", " * ", "/", and "∧".
- Comparisons: "<", ">", ">=", "<=", "==" (the same as), and " !=" (not the same as).
- Basic functions: log(), log10(), exp(), sin(), cos(), sqrt(), sum(), prod(), etc.
- It is noticeable that in the scalar calculation, vectors or matrices will be treated as a list of scalar numbers, and the calculation is conducted for each number, or **pair of numbers**, in the list(s).
    - A matrix is treated as a list of numbers, by default, in the order of column by column.

Scalar computations

- Run the following calculations and look at the results for better understanding.

```
> x1; a; x1−a      # Vector − Scale
  [1] 4 2 0
  [1] 1
  [1] 3 1 -1     (each number in the vector minus 1)
> x1; b; x1*b      # Vector × Scale
  [1] 4 2 0
  [1] 2
  [1] 8 4 0     (each number multiplied by 2)
> x1; x1∧2      # Power of vector
  [1] 4 2 0
  [1] 16 4 0     (squares of numbers)
```

Scalar computations

```
> x1; x2; x1−x2      # Vector ± Vector
[1] 4 2 0
[1] 1 0 -1
[1] 3 2 1      (a list of differences between pairs of numbers)

> x1*x2      # Vector * Vector = Vector of products of corresponding elements
[1] 4 0 0      (a list of products of pairs of numbers)

> x1/x2      # Vector / Vector = Vector of ratios of corresponding elements
[1] 4 Inf 0      (a list of ratios of pairs of numbers)
```

Scalar computations

```
> x5; 1/x5      # Division: 1 over a vector/matrix
  [1] 4 2 0 0 1 0 -1
  [1] 0.25 0.50 Inf Inf 1.00 Inf -1.00      (a list of inverses)
> sum(x1)      # Sum of (values in) x1
  [1] 6
> prod(x1)      # Product of x1
  [1] 0
> log(x1)      # Natural logarithm of x1
  [1] 1.3862944 0.6931472 -Inf
> exp(x1)      # Exponential of x1
  [1] 54.598150 7.389056 1.000000
```

Scalar computations

- Examples of scalar computations in the following 3 pages are seldom used. However, they might be commonly seen mistakes/misusages made by beginners.

```
> x1; m1; x1*m1        # Vector * Matrix
  [1] 4 2 0
        [,1]  [,2]
  [1,]    4     1
  [2,]    2     0
  [3,]    0    -1
        [,1]  [,2]
  [1,]   16     4
  [2,]    4     0
  [3,]    0     0
```

$=(4 \times 4 \quad 1 \times 4)$

$=(2 \times 2 \quad 0 \times 2)$

$=(0 \times 0 \quad -1 \times 0)$

Scalar computations

```
> m2      # Matrix * Vector
     [,1] [,2] [,3]
[1,] 4 2 0
[2,] 1 0 -1
> x2
[1] 1 0 -1
> m2*x2
     [,1] [,2] [,3]
```

[1,] 4$(= 4 \times 1)$ -2$(= 2 \times -1)$ 0$(= 0 \times 0)$

[2,] 0$(= 1 \times 0)$ 0$(= 0 \times 1)$ 1$(= -1 \times -1)$

Scalar computations

```
> x1; x5; x1*x5
[1] 4 2 0
[1] 4 2 0 0 1 0 -1
[1] 16 4 0 0 2 0 -4
```

$(4 \times 4, 2 \times 2, 0 \times 0, 4 \times 0, 2 \times 1, 0 \times 0, 4 \times -1)$

```
Warning message:
In x1 * x5 :
    longer object length is not a multiple of shorter object length
```

- **Summary**: a computation of two lists gives a list of results from the computation between pairs of corresponding scalars.

$$\begin{array}{cccccccccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} & x_{11} & \cdots \\ y_1 & y_2 & y_3 & y_1 & y_2 & y_3 & y_1 & y_2 & y_3 & y_1 & y_2 & \cdots \end{array}$$

Vector/Matrix computations

§3.2 Computations of vectors and matrices

- We use matrix m1 for illustration:

```
> m1
       [,1] [,2]
[1,]    4    1
[2,]    2    0
[3,]    0   -1
```

- Transpose

```
> t(m1)      # transpose
       [,1]  [,2]  [,3]
[1,]    4     2     0
[2,]    1     0    -1
```

- Diagonal elements (note that m1 is not square):

```
> diag(m1)      #
[1] 4 0
```

Vector/Matrix computations

- Create a diagonal matrix given the diagonal elements as a vector

```
> b <- c(1,2,3); diag(b)      # diag(1,2,3) in mathematics
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

  - This is equivalent to diag(c(1,2,3)). However, diag(1,2,3) gives a 2×3 matrix with diagonal elements being 1.

- Identity matrix

```
> diag(3)      # identity matrix of order 3
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Vector/Matrix computations

- Addition or substraction operators of matrices with the same orders: "+" or "−".
- Matrix multiplication operator %*%:

```
> A <- m1%*%t(m1); A
      [,1] [,2] [,3]
[1,]   17    8   -1
[2,]    8    4    0
[3,]   -1    0    1

> t(m1)%*%m1
      [,1] [,2]
[1,]   20    4
[2,]    4    2
```

as.matrix

- Treat a vector/list as a matrix.

```
> X1 <- as.matrix(x1)      # A column vector of dimension 3
> nrow(X1); ncol(X1)       # Show dimensions of X1
3
1
```

- Treat a (numeric) data as a matrix.

```
> data2
  X  Y
1 4  1
2 2  0
3 0 -1
> as.matrix(data2)
     X  Y
[1,] 4  1
[2,] 2  0
[3,] 0 -1
```

Vector/Matrix computations

- Inner product of two vectors

```
> X2 <- t(as.matrix(x2))     # A row vector of dimension 3
> X2%*%X1                    # Inner product
      [,1]
[1,]   4
```

- Outer product of two vectors

```
> X1%*%X2        # Outer product
      [,1] [,2] [,3]
[1,]    4    0   -4
[2,]    2    0   -2
[3,]    0    0    0
```

Vector/Matrix computations

- Matrix inverse:

```
> solve(A)       # A is singular
Error in solve.default(A) :
   system is computationally singular:
   reciprocal condition number = 1.06752e-18

> A[3,3] <- 10; solve(A)       # A becomes non-singular
           [,1]       [,2]       [,3]
[1,]  1.1111111 -2.2222222  0.1111111
[2,] -2.2222222  4.6944444 -0.2222222
[3,]  0.1111111 -0.2222222  0.1111111
```

Vector/Matrix computations

- The matrix function solve() actually solves the matrix function $AX = b$ for X, where A is (must be) a square matrix, and b is either a vector or a matrix (with correct orders).
  - The default value for b is the identity matrix.

```
> solve(A, b)
  [1] -3.000000e+00  6.500000e+00  9.868649e-17(should be 0)
> solve(A, m1)
        [,1]          [,2]
  [1,]  0.0  1.000000e+00
  [2,]  0.5 -2.000000e+00
  [3,]  0.0 -2.467162e-17
```

Vector/Matrix computations

- The determinant of the squared matrix `A` can be calculated by `det(A)`.
- Eigenvalues and eigenvectors of a matrix

```
> A.eigen <- eigen(A)        # Eigenvalues and eigenvectors of A
> A.eigen
  eigen() decomposition
  $values
  [1] 20.8827420  9.9438942  0.1733638

  $vectors
              [,1]        [,2]        [,3]
  [1,]  0.90057801  0.05585920  0.43109047
  [2,]  0.42674490  0.07518195 -0.90124162
  [3,] -0.08275286  0.99560405  0.04386959
```

Vector/Matrix computations

- **Check**: which vectors (row vectors or column vectors) are the eigenvectors?

```
> # Check the eigenvalues and eigenvectors
> A%*%A.eigen$vectors
          [,1]      [,2]        [,3]
[1,] 18.806538 0.5554580  0.07473549
[2,]  8.911604 0.7476014 -0.15624270
[3,] -1.728107 9.9001813  0.00760540
> A.eigen$values[1]*A.eigen$vectors[,1]
[1] 18.806538 8.911604 -1.728107
> A.eigen$values[2]*A.eigen$vectors[,2]
[1] 0.5554580 0.7476014 9.9001813
> A.eigen$values[3]*A.eigen$vectors[,3]
[1] 0.07473549 -0.15624270 0.00760540
```

- **Answer**: column vectors.

Manipulations of logical vectors

§3.3 Manipulations of logical vectors

- Comparisons:

```
> c1 <- x1!=2; x1; c1        # x1≠2
  [1]  4  2  0
  [1]  TRUE  FALSE  TRUE
> c2 <- x2<=0; x2; c2        # x2≤0
  [1]  1  0  -1
  [1]  FALSE  TRUE  TRUE
```

- Difference in "=" and "==":

```
> c3 <- x1=x2        # Incorrect usage
  Error in c3 <- x1 = x2 : could not find function "c3"
> c3 <- x1==x2; c3
  [1]  FALSE  FALSE  FALSE
```

Computations of logical vectors

- Two operators: "&" (AND), "|" (OR).

```
> c1 & c2          # TRUE iff both are TRUE
[1]  FALSE  FALSE  TRUE

> c1 | c2          # TRUE iff either is TRUE
[1]  TRUE  TRUE  TRUE
```

- AND is superior to OR:

```
> c1 | c2 & c3         # AND superior to OR
[1]  TRUE  FALSE  TRUE
> c1 | (c2 & c3)       # Equivalent to the previous
[1]  TRUE  FALSE  TRUE
> (c1 | c2) & c3
[1]  FALSE  FALSE  FALSE
```

Manipulations of (numeric) vectors

§3.4 Manipulations of vectors

- Recall that a (column by default) vector will be treated as a list of values.

- Original vector:

```
> v <- c(1:3); v
    [1]  1  2  3
```

- Change value(s):

```
> v[2:3] <- c(4, 5); v
    [1]  1  4  5
```

- Change length of a vector by adding value(s):

```
> v[5] <- 4; v
    [1]  1  4  5  NA  4
```

Manipulations of (numeric) vectors

- Change length of a vector by removing some values, or, equivalently, by selecting a subset of values.
- Subset, the 2nd to the 4th value(s) of vector:

```
> v1 <- v[2:4]; v1
    [1]  4  5  NA
```

- Subset, the 2nd and the 4th value(s):

```
> v2 <- v[c(2,4)]; v2
    [1]  4  NA
```

- Subset, remove the 2nd and the 4th value(s):

```
> v3 <- v[−c(2,4)]; v3
    [1]  1  5  4
```

- Restriction(s), value(s) less than 4:

```
> v[v<4]
    [1]  1  NA
```

Manipulations of matrices

§3.5 Manipulations of matrices

- Change value(s) through assigning new value(s) for element(s) of the matrix.
- Original example matrix:

```
> m <- matrix(1:9, nrow=3, ncol=3); m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

- Sub-matrix, the 2nd and 3rd rows:

```
> m[2:3, ]
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
```

Manipulations of matrices

- Sub-matrix, the 1st and 3rd columns:

```
> m[ , c(1,3)]
     [,1] [,2]
[1,]    1    7
[2,]    2    8
[3,]    3    9
```

- Restriction(s), row(s) with values in the 1st column >1:

```
> m[m[,1]>1, ]
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
```

Manipulations of matrices

- Restriction(s), column(s) with values in the 1st row <5:

```
> m[,m[1,]<5]
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

- Concatenation: connect column vectors as a new vector.

```
> con <- as.vector(m); con
[1] 1 2 3 4 5 6 7 8 9
```

Manipulations of character vectors

§3.6 Manipulations of character/factor vectors

```
> a <- c('Tutorial', "in", 'R'); a
  [1] "Tutorial" "in" "R"

> paste('Tutorial', "in", 'R')      # Concatenate
  [1] "Tutorial in R"

> cat('Tutorial', "in", 'R')      # Concatenate and print
  [1] Tutorial in R

> paste("Today is", date())      # Length = 1
  [1] "Today is Mon May 20 17:49:28 2019"

> paste(c("X","Y"), 1:5, sep=".")      # Length = maximum length
  [1] "X.1" "Y.2" "X.3" "Y.4" "X.5"
```

Manipulations of factor vectors

- Functions: factor() and levels().
- Suppose we have a character vector as follows:

```
> state <- c("tas", "sa", "qld", "nsw", "nsw", "nt", "wa", "wa",
+            "qld", "vic", "nsw", "vic", "qld", "qld", "sa", "tas",
+            "sa", "nt", "wa", "vic", "qld", "nsw", "nsw", "wa",
+            "sa", "act", "nsw", "vic", "vic", "act")
```

- Convert it into a factor variable: a list (vector) of levels (like observations) which can occur repeatedly:

```
> statef <- factor(state); statef
 [1] tas sa qld nsw nsw nt wa wa qld vic nsw vic qld qld sa
[16] tas sa nt wa vic qld nsw nsw wa sa act nsw vic vic act
Levels: act nsw nt qld sa tas vic wa
```

- The first two lines are values of the factor variable, levels (different values) are given in the last line.

Manipulations of factor vectors

- Levels can be also displayed by function levels():

```
> levels(statef)
  [1] "act" "nsw" "nt" "qld" "sa" "tas" "vic" "wa"
```

- A numerical variable, especially a categorical variable, can also be treated as/converted into a factor variable.

```
> factor(c(1,2,3,3,2,2,1))
  [1] 1 2 3 3 2 2 1
  Levels: 1 2 3

  # Equivalent usages
> as.factor(c(1,2,3,3,2,2,1))
> fac <- as.factor(c(1,2,3,3,2,2,1)); fac
```

Manipulations of data frames

§3.7 Manipulations of data frames

- Consider the data frame data1 as an illustrative example.
- List values of one variable in the data frame:

```
> data1$x1       # values of x1
> data1[,1]      # values of the first variable/column
```

- The above two expressions are equivalent.
  - Note that in the second expression, the data frame is treated as a two-dimensional array (matrix).
  - Recall the difference:
    - all elements in a matrix must be numerical; however,
    - a data frame may contain non-numerical variables.

Manipulations of data frames

- Cite variable(s) in the data frame:

  ```
  > data1[1]      # the first variable (variable name and values)
  > data1[1:2]      # the first 2 variables
    # Treat the data frame as a matrix (variables as columns):
  > data1[ ,1:2]      # the first 2 variables
  > data1[c(1,3)]      # the 1st and 3rd variables
  ```

- Cite observation(s) in the data frame (line(s) in a matrix):

  ```
  > data1[1,]      # the 1st observation
  > data1[1:2,]      # the first 2 observations
  > data1[c(1,3), ]      # the 1st and 3rd observations
  ```

  - Do not forget the comma "," after the specified observation number(s).

- Select observation(s) with restriction(s):

  ```
  > data1[data1$x1<1, ]      # observations with x1<1
  > data1[data1$x2>0, ]      # observations with x2>0
  ```

Manipulations of data frames

- We define a new data frame by the following commands, for the manipulations what follows.

```
v1 <- rep(1:3,3)
v2 <- rep(1:3, each=3)
v3 <- 1:9; v3[8] <- NA
data2 <- data.frame(v1, v2, v3)
View(data2)
```

- Remove observation(s) omitted value(s).

```
data3 <- na.omit(data2); View(data3)
```

- Remove the 3rd and 5th observations.

```
data3 <- data3[c(-3,-5), ]; View(data3)
```

Manipulations of data frames

- Reorder data3 by values of v2:

```
> data3[order(data3$v1), ]          # Ascending order
> data3[order(data3$v1, decreasing = T), ]          # Descending order
> data3[order(−data3$v1), ]          # Descending order
```

- Reorder data3 by more than one variables:

```
> data3[order(data3$v2, −data3$v1), ]
```

- Reorder data2 from the last to the first observation:

```
> data2$Obs <- 1:nrow(data2)          # Add the natural order
> data2[order(−data2$Obs), ]          # Descending order
```

Save and load selected R object(s)

- Save selected R object(s), as an `.Rdata` file, for future use.

```
    # Select R objects (matrix, vector, data frame, function, etc.)
  > SEL <- list(A, data1, A.eigen, pi.sim)
  > View(SEL)
    # Names of these R objects are [[1]], [[2]], etc.
    # Change the name(s)
  > names(SEL) <- c("Matrix", "Data_Frame", "Estimation",
    "Function")
  > View(SEL)
    # Save as an .Rdata file
  > save(SEL, file = "C:/Teaching/R Workshop/R_SEL.Rdata")
```

- Load the saved `.Rdata` file.

```
  > load("C:/Teaching/R Workshop/R_SEL.Rdata")
```

- Apply/Use R object(s) in the list.

```
  > SEL$Matrix; SEL$Data_Frame$x1
```

§3.8 Probabilities and distributions

- R provides four types of functions related to probabilities and distributions:
  - **Probabilities (cdf)** $\mathrm{P}(X \le x)$ by functions p*dist*(), in which "*dist*" represents key words of a specific type of distributions. E.g., function pnorm() provides probabilities related to normal distributions.
  - **Densities (pdf or pmf)** $p(x)$ by functions d*dist*().
  - **Quantiles** q_α by functions q*dist*().
  - **Random numbers** by functions r*dist*().

- **Basic formats**:

p*dist*(x, *parameters*, lower.tail=, log.p=)
d*dist*(x, *parameters*=, log=)
q*dist*(p, *parameters*=, lower.tail=, log.p=)
r*dist*(n, *parameters*=)

Probabilities and distributions

Distribution	**R name**	**additional arguments**
beta	beta	shape1, shape2, ncp
binomial	binom	size, prob
Cauchy	cauchy	location, scale
chi-squared	chisq	df, ncp
exponential	exp	rate
F	f	df1, df2, ncp
gamma	gamma	shape, scale
geometric	geom	prob
hypergeometric	hyper	m, n, k
log-normal	lnorm	meanlog, sdlog
logistic	logis	location, scale
negative binomial	nbinom	size, prob
normal	norm	mean, sd
Poisson	pois	lambda
Student's t	t	df, ncp
uniform	unif	min, max
Weibull	weibull	shape, scale

Examples of probabilities and quantiles

- Consider $X \sim N(2, 0.5^2)$ as an illustrative example.

```
> # Probabilities P(X ≤ 2.3)
> pnorm(2.3, mean = 2, sd = 0.5, lower.tail = TRUE, log.p = FALSE)
  [1] 0.7257469
> pnorm(1:3, 2, 0.5)      # Probabilities of a vector
  [1]  0.02275013  0.50000000  0.97724987
> pnorm(2.3, 2, 0.5, log.p = TRUE)
  [1] -0.320554      # (Log(p), NOT the probability of log(X).)
> dnorm(2.3, 2, 0.5)      # Density p(2.3)
  [1] 0.6664492
> qnorm(0.95, 2, 0.5)      # Left 95% or right 5% quantile
  [1] 2.822427
```

Examples of random numbers

- Run the following R commands twice and see the differences in results.

```
> rnorm(5, 2, 0.5)       # Generate 5 random numbers
```

- Run the following R commands twice and see the differences in results.

```
    # Random numbers with a fixed seed
> set.seed(1234)
> RN <- rnorm(5, 2, 0.5); RN
```

- Run the following R commands twice and and compare the results with those in the previous steps.

```
    # Random numbers with a new seed
> set.seed(4321)
> rnorm(5, 2, 0.5)
```

Examples of likelihoods

- Likelihood is the joint density of a set of observations, see the following examples.

```
> # Joint density/Likelihood of RN
> like <- prod(dnorm(RN,2,0.5)); like
  [1] 0.004857782
> # Log-likelihood of RN
> log(like)
  [1] -5.327173
> loglike <- sum(dnorm(RN,2,0.5,log=T)); loglike
  [1] -5.327173
```

Set working directory

§3.9 Reading or loading data

- Set working directory:

```
> setwd("D:/Teaching/R Workshop")
> getwd()        # Display working directory
  [1] "D:/Teaching/R Workshop"
> list.files()       # Display files
  [1] "Introduction to R.docx"      "Tutorial R.pptx"
  [3] "An Introduction to R.docx"  "An Introduction to R.pdf"
  ...
```

- Run the commands in the next page to read data files from the working directory into R. Notice differences in the resulted objects.

Read data from files

```
> # Reading data from .txt or .dat files
> # Without variable names
> read.table('house.txt') -> house1        # Or 'house.dat'
> View(house1); names(house1)
> # With variable names
> read.table('house.txt', header=TRUE) -> house2        # Or
  'house.dat'
> View(house2); names(house2)
> # Read data from .csv files
> read.csv('house.csv') -> house3
> # Read data from .xlsx or  .xls files
> require(readxl)       # Load packages
> read_excel('house.xlsx') -> house4
> read_excel('house.xls') -> house5
```

Load and edit data

- Rename variables in a data frame:

```
> names(house5)[1] = "P"
> names(house5)[2] = "T"
> View(house5)
```

- Write an R data frame into an Excel .csv file:

```
> write.csv(house5, file = "house5.csv")
```

- Loading data from other R packages:

```
> data(package="rpart")      # List all data files
> data(solder,package="rpart")      # Read a data file
> solder      # Activate and display the data, or
> View(solder)      # Activate and view the data
```

Training Exercises

1. Find/generate the following objects.
   (a) $O1 = \mathrm{P}(1 < X < 5)$, where X is an exponential r.v. with rate 2.
   [Answer]: 0.1352899.
   (b) $O2 = \mathrm{P}(X = i)$ for $i = 2, 3, 4$, and X is a Poisson r.v. with mean 5.
   [Answer]: 0.08422434, 0.14037390, 0.17546737.
   (c) Quantiles $O3 = q_\alpha$ of Beta(2,3), Beta distribution with shape parameters 2 and 3, respectively, for $\alpha = 0.01, 0.05$, and 0.1.
   [Answer]: 0.04199864, 0.09761146, 0.14255932.
   (d) $O4$: 5 random numbers following a geometric distribution with probability $p = 0.3$, generated using a seed 0.
   [Answer]: 0, 1, 1, 6, 1.
   (e) $O5$: a vector containing the likelihood and log-likelihood of $O4$.
   [Answer]: 9.805927e-05, -2.630673.

Training Exercises

End of L2

2. (a) Create a data frame `Mydata` containing $n = 100$ observations the following variables. Random numbers should be generated using the seed value 1111.

X1:	100 random numbers from $N(4, 1^1)$.
X2:	100 random numbers from Exponential(2).
X3:	Indicators of "X2>0".
X4:	100 random numbers from $N(0, 0.3^1)$.
Y:	=1+2*X1+3*X2+4*X1*X2*X3+X4.

   (b) Reorder the data in ascending order of X3 first, and then in descending order of X1.

   (c) Report the first observation of all variables in the reordered data frame.
   [Answer]: 6.973465, 0.2602815, 0, 0.3412288, 16.069.

   (d) Finally, save the data frame as "Mydata.csv" in your own working directory.

Iris data

Chapter 4 Descriptive Statistics in R[3]

§4.1 Quantitative statistics

- Descriptive statistics: location measures and dispersion measure, presented in either quantitative statistics, tables or graphs (plots, charts).
- Consider the built-in data set `iris` as an illustrative example.

```
> dat <- iris; View(dat)
```

- There are 150 observations of five variables: `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`, and `Species` (character/factor).
- Rename the variables for simplicity.

```
> names(dat) <- c("SL", "SW", "PL", "PW", "Sp")
```

[3] `http://statsandr.com/blog/descriptive-statistics-in-r/`

Summary

- A summary gives a set of quantitative statistics of all variables in the data frame.

  > summary(dat)

- Notice that
  - Six quantitative statistics, `"Min."`, `"1st Qu."`, `"Median"`, `"Mean"`, `"3rd Qu."`, `"Max."`, are summarized for each numerical variables.
  - Frequencies of levels are counted for the factor variable `Sp`.

- The function `summary()` can be applied to some specific variable(s) too. For example,

  > summary(dat$SL)

  > summary(dat$Sp)

  > summary(dat[,-5])

Mean

- Mean of one variable.

```
> mean(dat$SL)
```

- Mean of numeric variables in a data frame.

```
> mean(dat)      # Doesn't work
[1] NA
> mean(dat[,1:4])      # Doesn't work
[1] NA
> apply(dat,2,mean)      # warning: not numeric
SL   SW   PL   PW   Sp
NA   NA   NA   NA   NA
> apply(dat[,1:4],2,mean) # mean of each column
      SL         SW         PL         PW
5.843333  3.057333  3.758000  1.199333
> apply(dat[,1:4],1,mean) # mean of each row
```

Other quantitative statistics

demo

Statistics	**Function in R**	**Remark**
Maximum	max(x)	
Minimum	min(x)	
Range	range(x) <- max(x)-min(x)	$\overset{\text{def}}{=}$c(min(), max()) Self-definition
Median	median(x)	
Quantile	quantile(x,p)	left quantile
Interquartile range	IQR(x)	
Variance	var(x)	
Standard deviation	sd(x)	

Covariance	cov(x,y)
	cov(dat[,1:4])
		
		Covariance matrix

Correlation	cor(x,y)
	cor(dat[,1:4])
		
		Correlation matrix

Skewness	skewness(x)	library(e1071)
Kurtosis	kurtosis(x)	library(e1071)

Function by()

- Observations can be classified into groups based on values/level of some categorical/factor variable(s). Descriptive statistics can be calculated for each group of observations using the `by()` function.

```
> by(dat$SL, dat$Sp, summary)
  dat$Sp: setosa
  Min. 1st Qu. Median Mean 3rd Qu. Max.
  4.300   4.800   5.000   5.006   5.200   5.800
  ------------------------------------------------
  dat$Sp: versicolor
  Min. 1st Qu. Median Mean 3rd Qu. Max.
  4.900   5.600   5.900   5.936   6.300   7.000
  ------------------------------------------------
  dat$Sp: virginica
  Min. 1st Qu. Median Mean 3rd Qu. Max.
  4.900   6.225   6.500   6.588   6.900   7.900
```

One dimensional table

§4.2 Contingency tables

- Tables of frequencies or proportions of observations falling into categories based on levels of categorical variable(s) are called contingency tables.
- Variable `dat$Sp` is a categorical variable. Define one more categorical variable by the following R command.

```
> dat$size <- ifelse(dat$SL < mean(dat$SL), "small", "large")
```

- One-dimensional contingency tables.

```
> table(dat$size)        # Frequenticies
large small
   70    80
> proportions(table(dat$size))        # Proportions
    large     small
0.4666667 0.5333333
```

Two dimensional table

- Two-dimension contingency tables.

```
    # Frequencies
  > table(dat$Sp, dat$size)

               large small
    setosa         0    50
    versicolor    26    24
    virginica     44     6
```

- Note that two variables in the `table()` function define rows (i) and columns (j) of the contingency table, respectively.

Two dimensional table

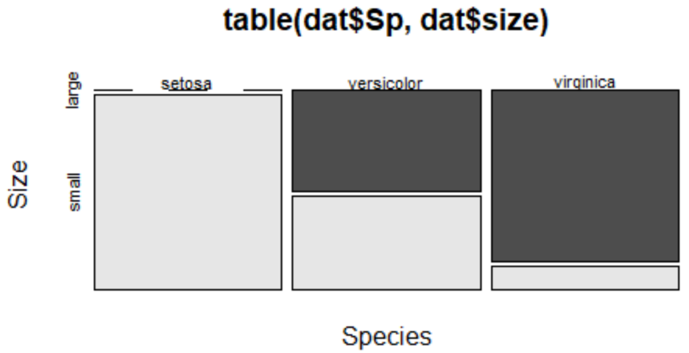
- Contingency tables in proportions.

```
    # Overall proportions
> proportions(table(dat$Sp, dat$size))
                  large     small
    setosa     0.0000000 0.3333333
    versicolor 0.1733333 0.1600000
    virginica  0.2933333 0.0400000
    # Round-off the proportions
> round(proportions(table(dat$Sp, dat$size)),2)
    # Row proportions
> proportions(table(dat$Sp, dat$size),1)
    # Column proportions
> proportions(table(dat$Sp, dat$size),2)
```

- An alternative way.

```
> xtabs(~ Sp + size, data=dat)
```

Visualization of contingency tables



- The above visualization of the contingency table was created by the following R command.

```
> mosaicplot(table(dat$Sp, dat$size), color = TRUE,
+            xlab = "Species", ylab = "Size")
```

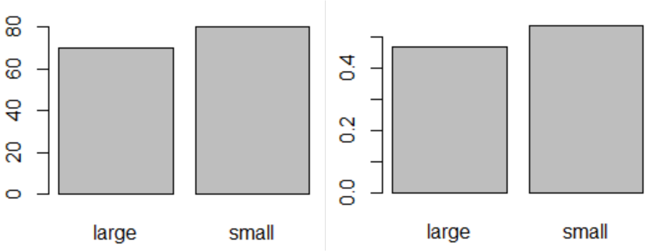
- Note that two variables in `table()` could be understood as X-axis and Y-axis, respectively.

Bar chart

§4.3 Descriptive plots

- Data presentations are frequently done graphically, e.g., histograms, bar charts, pie charts, etc.
- Bar charts can be plotted by the R function `barplot()`.

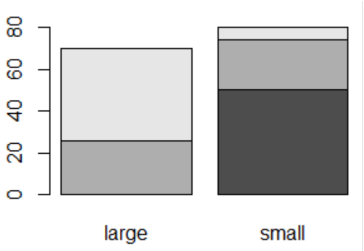
```
> barplot(table(dat$size))        # table() is mandatory
> barplot(proportions(table(dat$size)))
```

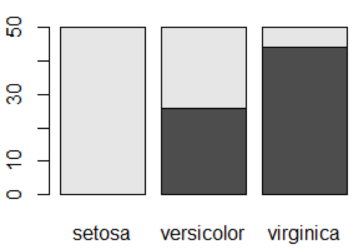


Bar chart

- Bar charts of a two-way contingency table (stacked bar chart).

```
> barplot(table(dat$Sp, dat$size))      # stacked barchart
> barplot(table(dat$size, dat$Sp))
```





- Note the order of two variables in the table() function can be understood as y on x.

drawback: not consistent

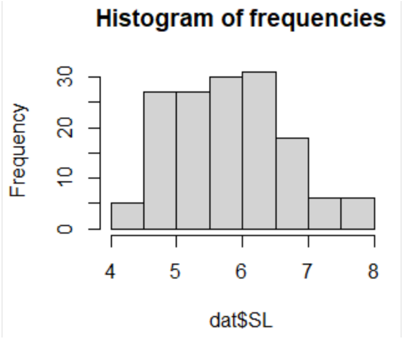
More on bar chart

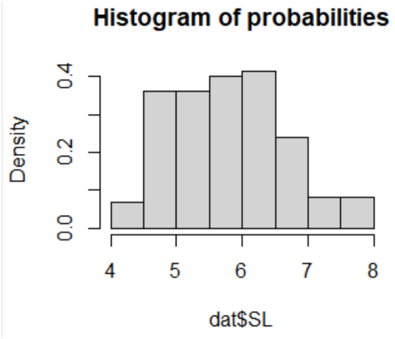
- Run the following R commands to learn more about barplot().

```
# Save tables
table1 <- table(dat$size)
table2 <- table(dat$size,dat$Sp)
# Barplots with color
barplot(table1, col=c("blue", "yellow"))
# Horizontal barplots
barplot(table1, col=c("blue", "yellow"), horiz=T)
# Stacked barplot with legend
barplot(table2, col=c("blue", "yellow"),
        legend.text = c("large", "small"),
        ylim = c(0,80))
# Grouped barplot with legend
barplot(table2, col=c("blue", "yellow"),
        legend.text = c("large", "small"),
        ylim = c(0,80), beside = T)
```

Histogram

- The histogram of ONE variable can be plotted using R function `hist()`.
  - > hist(dat$SL, main="Histogram of frequencies")
  - > hist(dat$SL, probability=T, main="Histogram of probabilities")

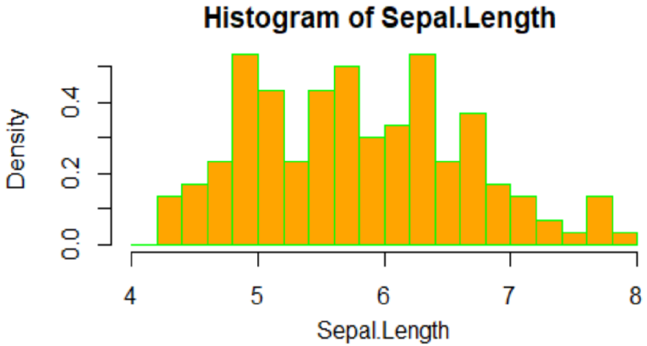




Histogram

- We may plot colorful histograms. See the following example.

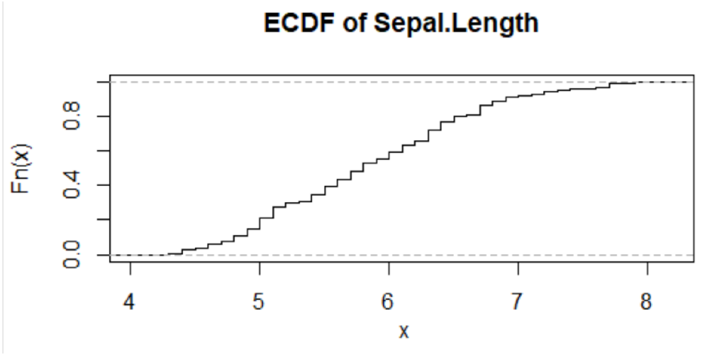
```
> hist(dat$SL, prob=T, seq(4, 8, 0.2),
         main="Histogram of Sepal.Length",
         xlab="Sepal.Length",
         col="orange", border="green")
```



ECDF

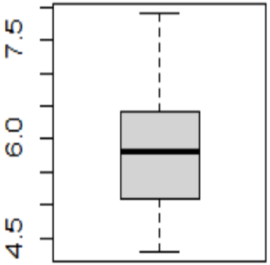
- Empirical cumulative distribution function.

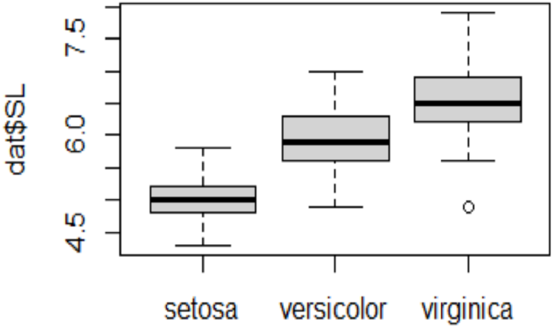
```
> plot(ecdf(dat$SL), do.points=F, verticals=T,
        main="ECDF of Sepal.Length")
```



Box plots

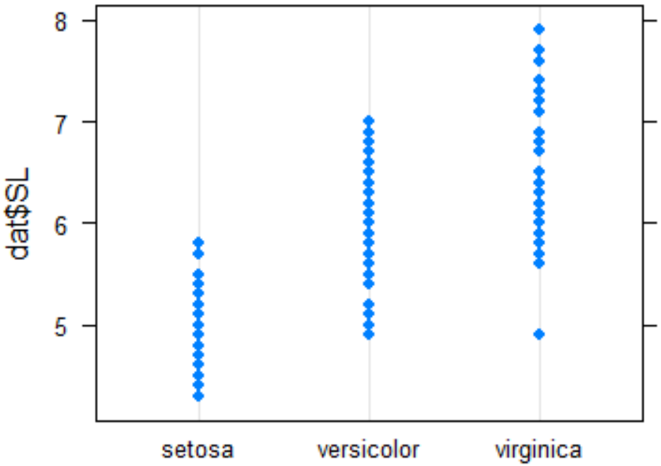
```
> boxplot(dat$SL)          # Box plot of SL
> boxplot(dat$SL~dat$Sp)         # Box plots by species
```





Dot plots

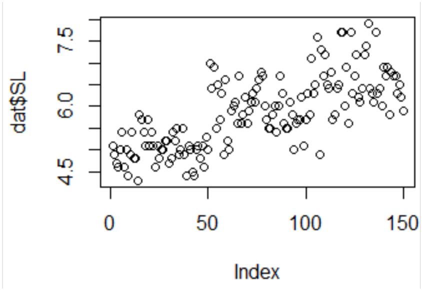
```
> library(lattice)
> dotplot(dat$SL~dat$Sp)
```

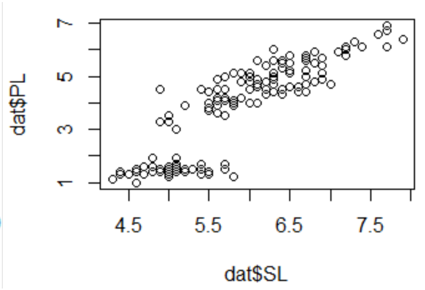


Scatter plot

> plot(dat$SL)   # One-way plot

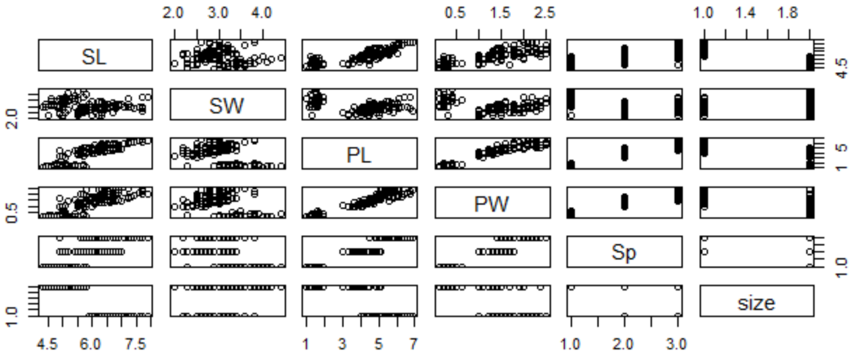
> plot(datSL, datPL)   # Two-way plot





Scatter plots

> plot(dat)   # Scatter plot of a data frame

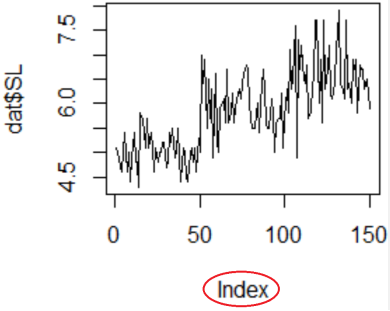


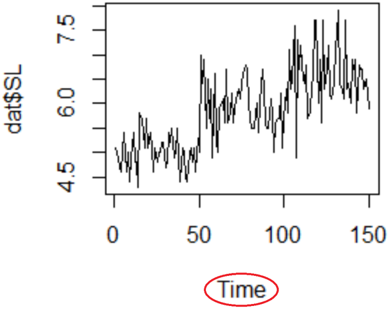
- Note that command `plot(dat$Sp, dat$SL)` produces a box plot since dat$Sp is a categorical/factor variable.

Line plots

- The following two commands plot (almost) the same line plot.

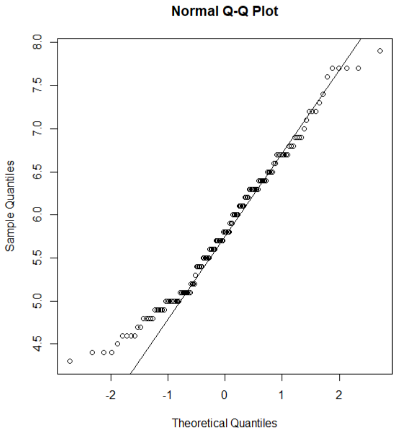
```
> plot(dat$SL, type="l")      # Line plot
> ts.plot(dat$SL)      # Time plot
```





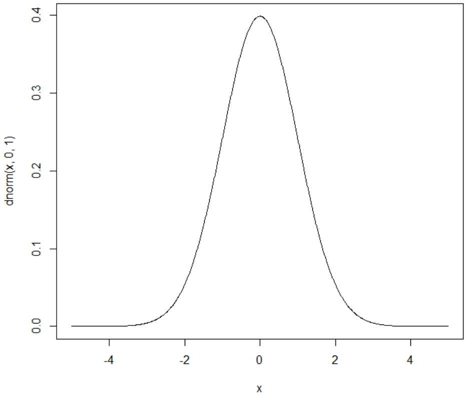
Q-Q plots

```
> par(pty="s")        # Arrange a square region
> qqnorm(dat$SL)       # Dot plot
> qqline(dat$SL)      # Add a regression line
```



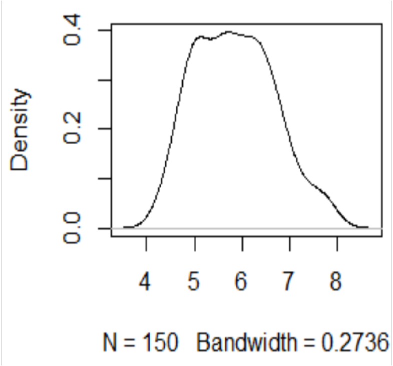
Plot densities

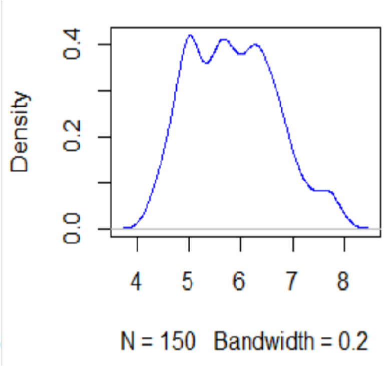
```
    # Plot of a specific density (normal)
>   x <- seq(-5,5,0.01)       # Set x-values
>   plot(x,dnorm(x,0,1),type="l")
```



Plot densities

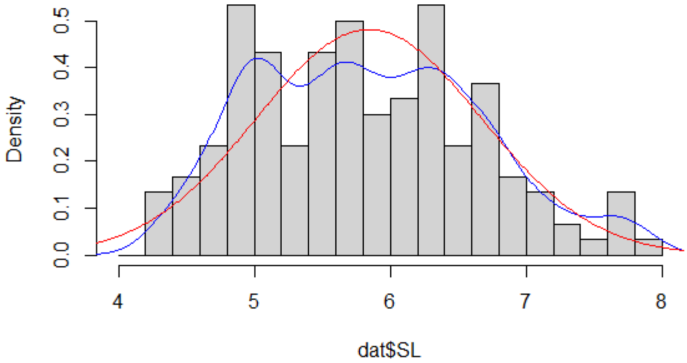
```
  # Plot an estimated density (kde)
> plot(density(dat$SL))        # Default bandwidth
> plot(density(dat$SL, bw=0.2), col="blue")
```





Add lines to histogram

```
> hist(dat$SL, prob=T, seq(4, 8, 0.2))
> lines(density(dat$SL, bw=0.2), col="blue")      # Add kde
  # Add (compared with) a normal density
> x <- seq(3.5,8.5,0.01)
> lines(x, dnorm(x, mean(dat$SL), sd(dat$SL)), col="red")
```



One sample t-tests

§4.4 Statistical tests

- Test $H_0: \mu = 0$ vs $H_1: \mu \neq 0$.

```
> t.test(dat$SL)
```

- Test $H_0: \mu = 6$ vs $H_1: \mu \neq 6$.

```
> t.test(dat$SL, mu=6)
```

- Test $H_0: \mu = 6$ vs $H_1: \mu < 6$.

```
> t.test(dat$SL, mu=6, alternative = "less")
```

Two-sample t-tests

- $H_0 : \mu_1 = \mu_2$ vs $H_1 : \mu_1 \neq \mu_2$.

```
> t.test(dat$SL, dat$SW)
```

- $H_0 : \mu_1 - \mu_2 = 3$ vs $H_1 : \mu_1 - \mu_2 \neq 3$.

```
> t.test(dat$SL, dat$SW, mu=3)
```

- $H_0 : \mu_1 - \mu_2 = 3$ vs $H_1 : \mu_1 - \mu_2 > 3$.

```
> t.test(dat$SL, dat$SW, mu=3, alternative = "greater")
```

- $H_0 : \sigma_1 = \sigma_2$ vs $H_1 : \sigma_1 \neq \sigma_2$.

```
> var.test(dat$SL, dat$SW)
```

- $H_0 : \sigma_1/\sigma_2 = 3$ vs $H_1 : \sigma_1/\sigma_2 \neq 3$.

```
> var.test(dat$SL, dat$SW, ratio=3)
```

- Test difference in means assuming equal variance.

```
> t.test(dat$SL, dat$SW, var.equal = T)
```

Normality tests

- Shapiro-Wilk test.

```
> shapiro.test(dat$SL)
```

- Kolmogorov-Smirnov test.

```
> ks.test(dat$SL, "pnorm", mean=mean(dat$SL), sd=sd(dat$SL))
```

- Jarque-Bera Test.

```
> require(tseries)
> jarque.bera.test(dat$SL)
```

More on Kolmogorov-Smirnov test

- Kolmogorov-Smirnov test for any distribution.

```
> ks.test(dat$SL, "pt", df=4)
```

- Two-sample Kolmogorov-Smirnov test: test for the same distribution.

```
> ks.test(dat$SL, dat$SW)
```

For loops

Chapter 5 Advanced Computations, Functions, and Graphics

- **For and while loops.**
- **Conditional executions**
- **Defining your own functions**
- **More on plots**

For loops

§5.1 For and while loops

- Loops are usually used for batch or recursive computations.
- Define **sequences** using **for loops.**

```
> for (i in 1:5) {x[i] <- 0.1*i; y[i] <- 2*i}
  Error: object 'x' not found
> x; y
  Error: object 'x' not found
> x <- numeric(); y <- numeric()
> for (i in 1:5) {x[i] <- 0.1*i; y[i] <- 2*i}
> x; y
  [1] 0.1 0.2 0.3 0.4 0.5
  [1] 2 4 6 8 10
```

For loops

- **Iteration**: Start with $x = 1$ and $y = 2$, iteratively update x with $\sqrt{xy}$, and then y with $\frac{x+y}{2}$. Find x and y after 100 iterations.

```
> x <- 1; y <- 2        # Initial value
> for (i in 1:100) {
+   x <- sqrt(x*y)      # Update x
+   y <- (x+y)/2        # Update y
+ }                     # End of the loop
> x; y
  [1] 1.604556
  [1] 1.604556
```

- The two values converge to a same constant. But we do not know when.

For loops

- **Iteration**: Output x- and y-sequences or the previous iteration.

```
> x <- numeric(100); y <- numeric(100)
> x[1] <- 1; y[1] <- 2        # Initial value
> for (i in 2:100) {
+    x[i] <- sqrt(x[i-1]*y[i-1])      # Update x
+    y[i] <- (x[i]+y[i-1])/2      # Update y
+ }                    # End of the loop
> x[1:15]
> y[1:15]
```

- We may find that two sequences "converge" after 12 iterations.

For loops

- **Recursion**: find the first 100 values for $z_n = 0.5 + 0.8z_{n-1} - 0.5z_{n-2}$, starting from $z_1 = 1$ and $z_2 = 3.2$.

```
> z <- numeric(100);
> z[1] <- 1; z[2] <- 3.2
> for (i in 3:100) z[i] <- 0.5+0.8*z[i-1]-0.5*z[i-2]
> z
  [1]  1.0000000  3.2000000  2.5600000  0.9480000  -0.0216000
  [6]  0.0087200  0.5177760  0.9098608  0.9690006   0.8202701
 [11]  0.6717158  0.6272376  0.6659322  0.7191270   0.7423355
   ⋮
 [56]  0.7142857  0.7142857  0.7142857  0.7142857   0.7142857
   ⋮
 [96]  0.7142857  0.7142857  0.7142857  0.7142857   0.7142857
```

- The sequence converges up to 7 decimal places after about 55 iterations.

While statement

- To save the computational power, we can use a while loop/statement to stop the recursion/iteration in the previous example when the required precision is achieved.
- Suppose we want to stop the recursion when $|z_n - z_{n-1}| < 10^{-7}$. We can use the following commands:

```
> z <- numeric(); z[1] <- 1; z[2] <- 3.2      # Initialization
> i <- 2      # Count the required number of recursions
> # Distance/Difference between two consecutive values
> dist <- abs(z[2]-z[1])      # Original difference
> while (dist>1E-7) {
>    i <- i+1      # Update steps i
>    z[i] <- 0.5+0.8*z[i-1]-0.5*z[i-2]
>    dist <- abs(z[i]-z[i-1])      # Update difference
> }
> # Output required number of recursions and the final value
> i; z[i]                        # Output: i=48, z[48]=0.7142861
```

IF-ElSE statements

§5.2 Conditional executions

- **IF, IF-ELSE**, and **IFELSE** statements are frequently used for conditional executions.
- Run the following **if statements** and see the results.

```
> # if statement 1
> x <- 8
> if (x<9) a1 <- "x<9"; a1                                      "x<9"
> # if statement 2
> if (x<7) a2 <- "x<7"; a2                  Error: object 'a2' not found
> # if statement 3
> a3 <- character(1)
> if (x<7) a3 <- "x<7"; a3                          "" (Empty variable)
```

If-else statement

- See the following two **if-else statements** and corresponding outputs:

```
> # if-else statement 1
> if (x<7) b1="x<7" else b1="x>=7"
> b1
[1] "x>=7"

> # if-else statement 2, with multiple actions
> if (x<9) {b2=1; b3=2} else {b2=-1; b3=-2}
> b2; b3
[1] 1
[1] 2
```

If-else statement

- **Notice** that “if” and “else” have to be (grouped) in one command. See the following example and outputs.

```
> # if-else statement 3:
> # incorrect use of line-breaks
> if (x>9) { b4 = 1; b5 = 2 }
> else { b4 = -1; b5 = -2 }            Error: unexpected 'else' in "else"
> # correct use of line-breaks
> if (x>9) {
>    b4 = 1; b5 = 2
> } else {
>    b4 = -1; b5 = -2
> }
> b4; b5
```

If-else statement

- Using braces, we can include more than one if-else statement, or a chain of if-else statements, in one command. See the next example.

```
# Chain of if-else statements:
> marks <- 56
> if (marks >= 80) {
>    grade = "A"
>    } else if (marks >= 65) {
>       grade = "B"
>       } else if (marks >= 50) {
>          grade = "C"
>          } else if (marks >= 40) {
>             grade = "D"
>             } else grade = "F"
> grade
  [1] "C"
```

Ifelse statement

- There is a third version of if-else statements, the **ifelse statement**.
- **Format**: $\boxed{\texttt{ifelse(cond, val1, val2)}}$, where
  - `cond` is/are condition(s),
  - `val1` is the value when the condition(s) is/are TRUE, and
  - `val2` is the value for FALSE.
- See the following example of ifelse statement, which is equivalent to if-else statement 1.

```
> # Direct use
> ifelse(x<7, "x<7","x>=7")
  [1] "x>=7"
> # Assignment the value to an R object
> b6 <- ifelse(x<7, "x<7","x>=7")
> b6
  [1] "x>=7"
```

Function 1: sum of two cubes

§5.3 Defining your own functions

- A list of R built-in objects/functions:

```
> builtins()      # Display at most 1000 objects
> builtin <- builtins(); View(builtin)      # View all of them
```

- Example of self-defined function 1: $f(x,y) = x^3 + y^3$.

```
> sum1.cub <- function(x,y) x^3+y^3
> sum1.cub(1,2)      # Example
  [1] 9
> sum2.cub <- function(x,y) {
+    a <- x^3+y^3
+    cat("The sum of cubes of", x, "and", y, "is", a)      # Output
+ }      # The end of definition
> sum2.cub(1,2)      # Example
  The sum of cubes of 1 and 2 is 9
```

Function 2: transpose of matrix

- Example of self-defined function 2: transpose of matrix.

```
> trans.m <- function(m){
+   vm <- as.vector(m)
+   tm <- matrix(vm, nrow=ncol(m), ncol=nrow(m), byrow=T)
+   return(list(m,tm)) }          # The end of definition
> m <- matrix(1:8, nrow = 2, ncol = 4)
> trans.m(m)
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
```

Function 3: test means for two samples

- Example of self-defined function 3: test means for two samples.
  - Let $X = \{x_i\}_{i=1}^{n_1}$ and $Y = \{y_i\}_{i=1}^{n_2}$ be two independent samples.
  - Denote the sample mean and sample variance of two samples by μ_i and s_i^2, $i = 1, 2$, respectively.
  - The test statistic is

$$T = \frac{\mu_1 - \mu_2}{s\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}},$$

    where

$$s^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}.$$

  - Under the null hypothesis $H_0 : \ \mu_1 = \mu_2$, $T \sim t(n_1 + n_2 - 2)$.
  - The alternative hypothesis can be either one sided ("less" or "greater") or two sided ("two.sided").

Function 3: test means for two samples

- The R definition:

```
twomean.test <- function(x, y, alt) {
  n1 <- length(x); n2 <- length(y)
  mu1 <- mean(x); mu2 <- mean(y)
  s1 <- var(x); s2 <- var(y)
  s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
  st.test <- (mu1 - mu2)/sqrt(s*(1/n1 + 1/n2))
  df.test <- n1+n2-2
  if (alt=="two.sided") pv.test <- 2-2*pt(abs(st.test), df.test)
  if (alt=="less") pv.test <- pt(st.test, df.test)
  if (alt=="greater") pv.test <- 1-pt(st.test, df.test)
  paste("t =", st.test, ",", "df =", df.test, ",",
    "p-value =", pv.test, ",", "alternative =", alt)
    # Output multiple results
}
```

Function 3: test means for two samples

- Run and results:

```
> A <- c( 79.98, 80.04, 80.02, 80.04, 80.03, 80.03, 80.04, 79.97,
+            80.05, 80.03, 80.02, 80.00, 80.02)
> B <- c( 80.02, 79.94, 79.98, 79.97, 79.97, 80.03, 79.95, 79.97)
> twomean.test(A, B, "less")
  [1] "t = 3.47224484709379 , df = 19 , p-value = 0.998724497892945
  , alternative = less"
> twomean.test(A, B, "greater")
  [1] "t = 3.47224484709379 , df = 19 , p-value = 0.0012755021070554
  , alternative = greater"
> twomean.test(A, B, "two.sided")
  [1] "t = 3.47224484709379 , df = 19 , p-value =
  0.00255100421411081 , alternative = two.sided"
```

- **Exercise**: define a function of $\ln(1+x)$ by approximation using Taylor expansion, with accuracy up to the 10th decimal place.

Graphical procedures via plot()

§5.4 More on plots

- Plotting commands are divided into three basic groups:
  - High-level plotting functions create a new plot on the graphics device, possibly with axes, labels, titles and so on.
  - Low-level plotting functions add more information to an existing plot, such as extra points, lines and labels.
  - Interactive graphics functions allow you interactively add information to, or extract information from, an existing plot, using a pointing device such as a mouse.
- In addition, R maintains a list of graphical parameters which can be manipulated to customize your plots.

The plot() function

- Read the help information provided by R command ?plot.
- Format (of plots/variables): plot(x, y, ...).
    - plot(x) creates a *time plot* of a time series variable (data) x.
    - plot(x, y) creates a *scatter plot* of y against x, if x and y are two numerical vectors of the same length.
    - plot(f, y), where f is a factor, creates a *box plot* of y, for each level of f.
    - plot(y $\sim$ a + b + c) creates scatter plots of y against a, b, and c, *one by one*.
    - plot(*df*), where *df* is a data frame, creates scatter plots of *all pairs* of (numerical) variables in the data frame.
    - ... are arguments of graphical parameters such as type, main, sub, etc.

The plot() function

- Examples of basic usages of plot().

```
plot(house2$price)

plot(house2$lot, house2$price)

subf <- as.factor(house2$sub)

plot(subf, house2$price)

plot(house2)

plot(house2$price ~ house2$lot + house2$space + house2$age)
```

Arguments of plot()

- Types of plots: the “type =” argument:
    - "p" for points,
    - "l" for lines,
    - "b" for both,
    - "c" for the lines part alone of "b",
    - "o" for both ’overplotted’,
    - "h" for ’histogram’ like (or ’high-density’) vertical lines,
    - "s" for stair steps,
    - "S" for other steps, see ’Details’ below,
    - "n" for no plotting.

Arguments of plot()

- main =: an overall title for the plot.
- sub =: a sub title for the plot.
- xlab =: a title for the x axis.
- ylab =: a title for the y axis.
- xlim =: range for the x axis.
- ylim =: range for the y axis.
- col =: color of the plot.
- lty =: type of line, active if type = l.
- asp =: the y/x aspect ratio.

Low-level plotting functions

- lines(): add a line (curve).
- points(): add some points.
- abline(h=*value(s)*): add horizontal line(s) at the specified value(s).
- abline(v=*value(s)*): add vertical line(s) at the specified value(s).

Low-level plotting functions

- Example: run the following commands **one by one**:

```
yy <- log(co2[1:30]); xx <- 1:30
plot(y=yy, x=xx, type='l', col=2,lty=2,
      xlim=c(0,35), ylim=c(5.74,5.78),
      xlab='xaxis', ylab='yaxis', main='Demo of Plot()')
y2 <- log(co2[31:60])
lines(y=y2, x=xx, lty=3, col=3)
points(y=y2, x=xx)
abline(h=5.76)
abline(v=c(10,30), col="blue")
```

- R package `ggplot2`: Create Elegant Data Visualisations Using the Grammar of Graphics.

Linear regression model (LIM)

Chapter 6 Linear Regression in R

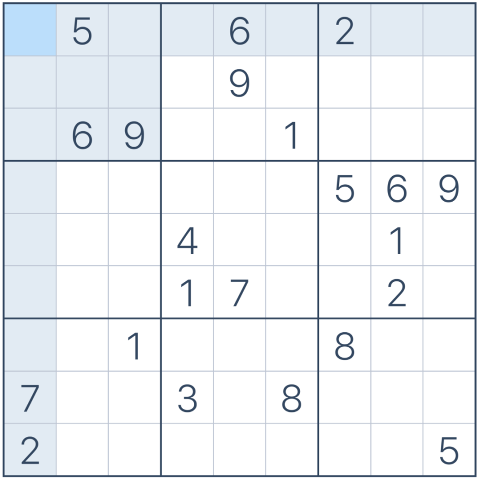
- We use the food data “food.csv” (simple LIM) and house data “house.csv” (multiple LIM) as example data in this chapter.
- Least squares estimation (LSE), analysis of variance (ANOVA), model building strategies, model diagnostics, and predictions, will be introduced.
- Student who are not familiar with linear regressions may learn by themselves some basic knowledge on LIM.
- We shall stick on the provided example R programs for this chapter.

Appendix: Solve Sudoku Puzzles with R

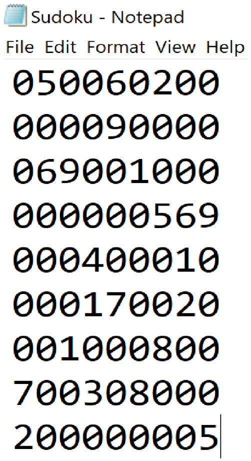
- **Sudoku**: a very popular game of solving "*numerical*" puzzles.
  - A logical game instead of mathematical.
- Large number of sudoku puzzles, either online or off-line, computer-based or mobile Apps, are available.
- R package `sudoku` provides tools to generate and solve sudoku games.
- You can even define/develop your own functions/packages for this.[4]
- Let's solve the sudoku puzzle in the next page with R, as an illustrative example.

[4] https://www.r-bloggers.com/2020/11/how-to-solve-sudoku-with-r/

Appendix



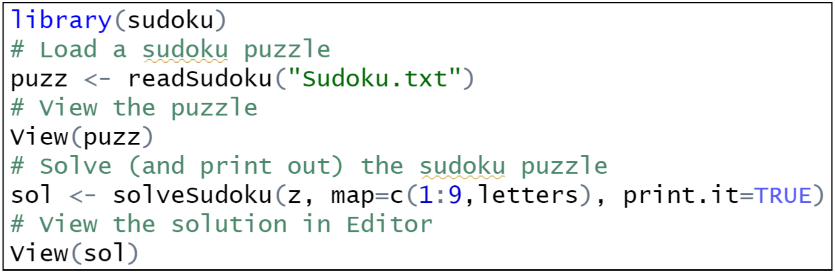
	5			6		2		
				9				
	6	9			1			
						5	6	9
			4				1	
			1	7			2	
		1				8		
7			3		8			
2								5



```
050060200
000090000
069001000
000000569
000400010
000170020
001000800
700308000
200000005
```

Appendix

- Create (`Sudoku.txt`) file for the puzzle, and then run the following R commands to solve it. The txt file can be done by either of the following,
  - directly downloaded (and saved) from some websites, or
  - manually created, or
  - converted from a pdf file (of a sudoku puzzle).



```
library(sudoku)
# Load a sudoku puzzle
puzz <- readSudoku("Sudoku.txt")
# View the puzzle
View(puzz)
# Solve (and print out) the sudoku puzzle
sol <- solveSudoku(z, map=c(1:9,letters), print.it=TRUE)
# View the solution in Editor
View(sol)
```

The End!