STAT 8670 - Computational Methods in Statistics

Chi-Kuang Yeh

2025-08-27

Table of contents

Preface

Description

Topics ins included are optimization, numerical integration, bootstrapping, cross-validation and Jackknife, density estimation, smoothing, and use of the statistical computer package of S-plus/R.

Prerequisites

MATH 4752/6752 – Mathematical Statistics II, and the ability to program in a high-level language.

Instructor

Chi-Kuang Yeh, I am an Assistant Professor in the Department of Mathematics and Statistics, Georgia State University.

- Office: 1407, 25 Park Place.
- Email: cyeh@gsu.edu.

Office Hour

[To be announced. By appointment for now]

Assignment

- ☐ Assignment 1: Date and topics TBA

Midterm

- ☐ Midterm 1: Date and topics TBA
- ☐ Midterm 2: Date and topics TBA

Final Exam

- ☐ Final Project: Date and topics TBA

Topics and Corresponding Lectures

Those chapters are based on the lecture notes. This part will be updated frequently.

Topic	Lecture Covered
Optimization	TBA
Numerical integration	TBA
Jackknife	TBA
Bootstrap	TBA
Cross-validation	TBA
Smoothing	TBA
Density estimation	TBA
Monte Carlo Methods	TBA

Recommended Textbooks

- Givens, G.H. and Hoeting, J.A. (2012). *Computational Statistics*. Wiley, New York.
- Rizzo, M.L. (2007) *Statistical Computing with R*. CRC Press, Roca Baton.
- Hothorn, T. and Everitt, B.S. (2006). *A Handbook of Statistical Analyses Using R*. CRC Press, Boca Raton.

Side Readings

- Wickham, H., Çetinkaya-Rundel, M. and Grolemund, G. (2023). *R for Data Science*. O'Reilly.

1 Data Structure and R Programming

Data types, operators, variables

Two basic types of objects: (1) data & (2) functions

- Data: can be a number, a vector, a matrix, a dataframe, a list or other datatypes
- Function: a function is a set of instructions that takes input, processes it, and returns output. Functions can be built-in or user-defined.

1.1 Data type

- Boolean/Logical: Yes or No, Head or Tail, True or False
- Integers: Whole numbers $\mathbb{Z}$, e.g., 1, 2, 3, -1, -2, -3
- Characters: Text strings, e.g., "Hello", "World."
- Floats: Noninteger fractional numbers, e.g., π, e.
- Missing data: `NA` in R, which stands for "Not Available." It is used to represent missing or undefined values in a dataset.

```r
day <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
weather <- c("Raining", "Sunny", NA, "Windy", "Snowing")
data.frame(rbind(day, weather))
```

```
             X1      X2        X3       X4      X5
day      Monday Tuesday Wednesday Thursday  Friday
weather Raining   Sunny      <NA>    Windy Snowing
```

- Other more complex types

1.1.1 To change data type

You may change the data type using the following functions, but the chance is that some of the information will be missing. Do this with caution!

```r
x <- pi
print(x)
```

```
[1] 3.141593
```

```r
x_int <- as.integer(x)
print(x_int)
```

```
[1] 3
```

Some of the conversion functions:

- `as.integer()`: Convert to integer.
- `as.numeric()`: Convert to numeric (float).
- `as.character()`: Convert to character.
- `as.logical()`: Convert to logical (boolean).
- `as.Date()`: Convert to date.
- `as.factor()`: Convert to factor (categorical variable).
- `as.list()`: Convert to list.
- `as.matrix()`: Convert to matrix.
- `as.data.frame()`: Convert to data frame.
- `as.vector()`: Convert to vector.
- `as.complex()`: Convert to complex number.

1.2 Operators

- Unary: With only **one** argument. E.g., `-x` (negation), `!x` (logical negation).
- Binary: With **two** arguments. E.g., `x + y` (addition), `x - y` (subtraction), `x * y` (multiplication), `x / y` (division).

1.2.1 Comparison Operator

Comparing two objects. E.g., `x == y` (equal), `x != y` (not equal), `x < y` (less than), `x > y` (greater than), `x <= y` (less than or equal to), `x >= y` (greater than or equal to).

1.2.2 Logical Operator

Logical operators are used to combine or manipulate logical values (TRUE or FALSE). E.g., `x & y` (logical AND), `x | y` (logical OR), `!x` (logical NOT).

We shall note that the logical operators in R are vectorized, `x | y` and `x || y` are different. The former is vectorized, while the latter is not.

```r
x <- c(TRUE, FALSE, FALSE)
y <- c(TRUE, FALSE, FALSE)
x | y  # [1]  TRUE FALSE FALSE
x || y # This will return an error
```

1.3 Indexing

Indexing is a way to access or modify specific elements in a data structure. In **R**, indexing can be done using square brackets `[]` for vectors and matrices, or the `$` operator for data frames. Note that the index starts from **0** in **R**, which is different from some other programming languages like Python.

1.4 Naming

In **R**, you can assign names to objects using the `names()` function. This is useful for making your code more readable and for accessing specific elements in a data structure.

A good practice is to use _ (underscore) to separate words in variable names, e.g., `my_variable`. This makes the code more readable and easier to understand.

```r
# Assign names to a vector
temp <- c(20, 30, 27, 31, 45)
names(temp) <- c("Mon", "Tues", "Wed", "Thurs", "Fri")
print(temp)
```

```
  Mon  Tues   Wed Thurs   Fri 
   20    30    27    31    45 
```

```r
rownames(temp) <- "Day1" # error
```

```r
temp_mat <- matrix(c(20, 30, 27, 31, 45), nrow = 1, ncol = 5)
colnames(temp_mat) <- c("Mon", "Tues", "Wed", "Thurs", "Fri")
rownames(temp_mat) <- "Day1" # error
print(temp_mat)
```

```
     Mon Tues Wed Thurs Fri
Day1  20   30  27    31  45
```

1.5 Array and Matrix

One may define an array or a matrix in **R** using the `array()` or `matrix()` functions, respectively. An array is a multi-dimensional data structure, while a matrix is a two-dimensional array.

```r
# Create a 1-dimensional array
array_1d <- array(1:10, dim = 10)
array_1d
```

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

```r
# Create a 2-dimensional array
array_2d <- array(1:12, dim = c(4, 3))
array_2d
```

```
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

```r
# Create a 3-dimensional array
array_3d <- array(1:24, dim = c(4, 3, 2))
array_3d
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    5    9
```

```
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

```r
# Create a matrix
my_matrix <- matrix(1:12, nrow = 4, ncol = 3)
my_matrix
```

```
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Note here, the matrix is a special case of an array, where the number of dimensions is exactly 2.

```r
is.matrix(array_2d)   # TRUE
is.matrix(my_matrix)  # TRUE

is.array(array_2d)    # TRUE
is.array(my_matrix)   # TRUE
```

1.6 Key and Value Pair

Key-Value Pair is a data structure that consists of a key and its corresponding value. In **R**, this can be implemented using named vectors, lists, or data frames. Usually, the most commonly used case is in the lists and data frames. The values can be extra by providing the corresonding key

```r
key1 <- "Tues"
value1 <- 32
key2 <- "Wed"
value2 <- 28

list_temp <- list()
list_temp[[ key1 ]] <- value1
list_temp[[ key2 ]] <- value2

print(list_temp)
```

```
$Tues
[1] 32

$Wed
[1] 28
```

```r
## Now providing a key - Tues
### First way
list_temp[["Tues"]]
```

```
[1] 32
```

```r
### Second way
list_temp$Tues
```

```
[1] 32
```

1.7 Data Frame

Dataframe is a two-dimensional, tabular data structure in R that can hold different types of variables (numeric, character, factor, etc.) in each column. It is similar to a spreadsheet or SQL table.

```r
iris <- datasets::iris
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

1.8 Apply function

The `apply()` function is the basic model of the family of apply functions in R, which includes specific functions like `lapply()`, `sapply()`, `tapply()`, `mapply()`, `vapply()`, `rapply()`, `bapply()`, `eapply()`, and others. These functions are used to apply a function to elements of a data structure (like a vector, list, or data frame) in a (sometimes) more efficient and concise way than using loops.

```
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
dimnames(x)[[1]] <- letters[1:8]
print(x)
```

```
  x1 x2
a  3  4
b  3  3
c  3  2
d  3  1
e  3  2
f  3  3
g  3  4
h  3  5
```

```
apply(x, MARGIN = 2, mean) #apply the mean function to their "columns"
```

```
x1 x2
 3  3
```

```
col.sums <- apply(x, MARGIN = 2, sum) #apply the sum function to their "columns"
row.sums <- apply(x, MARGIN = 1, sum) #apply the sum function to their "rows"
rbind(cbind(x, Rtot = row.sums), Ctot = c(col.sums, sum(col.sums)))
```

```
     x1 x2 Rtot
a     3  4    7
b     3  3    6
c     3  2    5
d     3  1    4
e     3  2    5
f     3  3    6
g     3  4    7
h     3  5    8
Ctot 24 24   48
```

Some of the commonly used apply functions:

- **lapply**: Apply a Function over a List or Vector
- **sapply**: a user-friendly version and wrapper of lapply by default returning a vector, matrix
- **vapply**: similar to sapply, but has a pre-specified type of return value, so it can be safer (and sometimes faster) to use.

1.9 Tidyverse

The tidyverse is a collection of open source packages for the R programming language introduced by Hadley Wickham and his team that "share an underlying design philosophy, grammar, and data structures" of tidy data. Characteristic features of tidyverse packages include extensive use of non-standard evaluation and encouraging piping.

```r
## Load all tidyverse packages
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.2     v tibble    3.3.0
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.1.0
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

```r
## Or load specific packages in the tidy family
library(dplyr) # Data manipulation
library(ggplot2) # Data visualization
library(readr) # Data import
library(tibble) # Tidy data frames
library(tidyr) # Data tidying
# ...
```

1.10 Pipe

Pipe operator `|>` (native after R version 4.0) or `%>$` (from magrittr package) is a powerful tool in **R** that allows you to chain together multiple operations in a clear and concise way. It takes the output of one function and passes it as the first argument to the next function.

For example, we can write

```r
set.seed(777)
x <- rnorm(5)

## Without using pipe
print(round(mean(x), 2))
```

```
[1] 0.37
```

```r
## Using pipe
x |>
  mean() |> # applying the mean function
  round(2) |> #round to 2nd decimal place
  print()
```

```
[1] 0.37
```

We can see that, without using the pipe, if we are applying multiple functions to the same object, we may have hard time to track. This can make the code less readable and harder to maintain. On the other hand, using pipe, we can clearly see the sequence of operations being applied to the data, making it easier to understand and modify.

1.10.1 Some rules

`|>` should **always have a space before it** and should typically **be the last thing on a line**. This simplifies adding new steps, reorganizing existing ones, and modifying elements within each step.

Note that all of the packages in the tidyverse family support the pipe operator (except `ggplot2`!), so you can use it with any of them.

1.11 Questions in class

1.11.1 Lecture 1, August 25, 2025

Q1. If I know Python already, why learn R?

Reply: My general take are 1). R is more specialized for statistical analysis and data visualization, while Python is a more general-purpose programming language. 2). R has a rich ecosystem of packages and libraries specifically designed for statistical computing, making it a popular choice among statisticians and data scientists. 3). R's syntax and data structures are often more intuitive for statistical tasks, which can lead to faster development and easier collaboration with other statisticians. 4). Also, the tidyverse ecosystem including *ggplot* and others are a big plus when dealing with big dataframes. 5). They are not meant to replace each other, but work as a complement.

Q2. Why my installation of R sometimes failed on a Windows machine?

Reply: There are many reasons. One of the most common reasons is that you may need to manually add the path to the environment variable.

1.11.2 Lecture 2, August 27, 2025

Q1. What's the difference of using `apply` v.s. `looping` in R?

Reply: The apply functions are often faster and more efficient than looping, especially for large datasets, because they have done some vectorization under the hood. Also, it has much higher readibility and better conciseness. However, depends on the task, you may want to do the **benchmarking** to see the performance difference.

Q2. How to use `pipe` with two or more variables?

Reply: There are several ways to do this.

1. Within the tidyverse family: One way is to use the `dplyr` package, which provides a set of functions that work well with the pipe operator. For example, you can use the `mutate()` function to create a new variable based on two existing variables. For example, you can do

```r
library(dplyr)
library(magrittr)  # for %$%
library(purrr)     # for pmap / exec if needed

my_df <- tibble(x = 1:5, y = 6:10)
f  <- function(a, b) a + 2*b

my_df %>%
  mutate(z = f(x, y))
```

```
# A tibble: 5 x 3
      x     y     z
  <int> <int> <dbl>
1     1     6    13
2     2     7    16
3     3     8    19
4     4     9    22
5     5    10    25
```

2. Using base R, you may do something like the following through the `magrittr` package's exposition pipe `%$%`:

```r
library(magrittr)
# method 1
my_df %$% f(x, y)
```

```
[1] 13 16 19 22 25
```

```r
# or use . as a placeholder
# method 2
my_df %>% { f(.$x, .$y) }
```

```
[1] 13 16 19 22 25
```

Some of the materials are adapted from CMU Stat36-350.

2 Optimization

The optimization plays an important role in statistical computing, especially in the context of maximum likelihood estimation (MLE) and other statistical inference methods. This chapter will cover various optimization techniques used in statistical computing.

For instance, for a linear regression

$$y = X\beta + \varepsilon.$$

From regression class, we know that the (ordinary) least-squares estimation (OLE) for β is given by $\widehat{\beta} = (X^\top X)^{-1}X^\top y$. It is convenient as the solution is in the **closed-form**! However, in the most case, the closed-form solutions will not be available.

For GLMs or non-linear regression, we need to do this **iterativelly**!

2.1 Speed comparison

```r
set.seed(2017-07-13)
X <- matrix(rnorm(5000 * 100), 5000, 100)
y <- rnorm(5000)
library(microbenchmark)
microbenchmark(solve(t(X) %*% X) %*% t(X) %*% y)
```

```
Warning in microbenchmark(solve(t(X) %*% X) %*% t(X) %*% y): less accurate
nanosecond times to avoid potential integer overflows
Unit: milliseconds
                             expr      min       lq     mean   median       uq
 solve(t(X) %*% X) %*% t(X) %*% y 28.57097 29.23575 30.04752 29.87713 30.62448
      max neval
 32.21641   100
```

```r
microbenchmark(solve(t(X) %*% X) %*% t(X) %*% y,
               solve(crossprod(X), crossprod(X, y)))
```

```
Unit: milliseconds
                                 expr      min       lq     mean   median
     solve(t(X) %*% X) %*% t(X) %*% y 28.39886 29.03231 29.73657 29.41512
 solve(crossprod(X), crossprod(X, y)) 24.96937 25.01504 25.25663 25.05457
       uq      max neval
 30.23379 32.94231   100
 25.16457 28.50168   100
```

2.2 Type of Optimization Algorithms

There are in general two types of the optimization algorithms: (1). **deterministic** and (2). **metaheuristic**. Deterministic and metaheuristic algorithms represent two distinct paradigms in optimization. Deterministic methods, such as gradient descent, produce the same solution for a given input and follow a predictable path toward an optimum. In contrast, metaheuristic approaches—like genetic algorithms—incorporate randomness and do not guarantee the best possible solution. However, they are often more effective at avoiding local optima and exploring complex search spaces.

2.3 Heuristic Algorithms

Many of the heuristic algorithms are inspired by the nature, such as the genetic algorithm (GA) and particle swarm optimization (PSO). These algorithms are often used for complex optimization problems where traditional methods may struggle to find a solution. Some of the popular heuristic algorithms include:

- Genetic Algorithm (GA)
- Particle Swarm Optimization (PSO)
- Simulated Annealing (SA)
- Ant Colony Optimization (ACO)

2.4 Deterministic Algorithms

Numerical approximation, what you learned in the mathematical optimization course. Some of the algorithms include:

- Gradient Descent
- Newton's Method
- Conjugate Gradient Method
- Quasi-Newton Methods (e.g., BFGS)

- Interior Point Methods

They often reply on the KKT conditions.

2.4.1 In R

`optim()` function, `nlm()` function or `mle()` function.

2.4.2 EM Algorithm

The EM (Expectation–Maximization) algorithm is an optimization method that is often applied to find maximum likelihood estimates when data is incomplete or has missing values. It iteratively refines estimates of parameters by alternating between (1) expectation step (E-step) and (2) maximization step (M-step).

Tip with Title

For example, consider the function $f(x) = x^2$.

Example

$$f(x) = x^2$$

Example Theorem

Examples are borrowed from the following sources:

- Peng, R.D. Advanced Statistical Computing.

3 Resampling, Jackknife and Bootstrap

3.1 Introduction

This chapter covers resampling methods including the jackknife and bootstrap techniques.

3.2 Jackknife

The jackknife is a resampling technique used to estimate the bias and variance of a statistic.

Jackknife is like a **leave-one-out cross-validation**. Let $\mathbf{x} = (x_1, \dots, x_n)$ be an observed random sample, and denote the ith jackknife sample by $\mathbf{x}_{-i} = (x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$, that is, a subset of $\mathbf{x}$.

For the parameter of interest θ, if the statistics is $T(\mathbf{x}) =: \hat{\theta}$ is computed on the full

3.2.1 When does jackknife not work?

Jackknife does not work when the function $T(\cdot)$ is **not a smooth** functional!

3.3 Bootstrap

The bootstrap is a resampling method that allows estimation of the sampling distribution of almost any statistic using random sampling methods.

3.4 Applications

These methods are widely used in statistical inference and have applications in various fields.

---

4 Additional Topics

This chapter covers additional topics that will only be going over if time permits.

4.1 High-dimensional data

4.2 Dimensional Reduction Methods

4.2.1 Principal Component Analysis

References

Part I

Appendix

5 Appendix: Introduction to R?

5.1 R

For conducting analyses with data sets of hundreds to thousands of observations, calculating by hand is not feasible and you will need a statistical software. **R** is one of those. **R** can also be thought of as a high-level programming language. In fact, **R** is one of the top languages to be used by data analysts and data scientists. There are a lot of analysis packages in **R** that are currently developed and maintained by researchers around the world to deal with different data problems. Most importantly, **R** is free! In this section, we will learn how to use **R** to conduct basic statistical analyses.

5.2 IDE

5.2.1 Rstudio

RStudio is an integrated development environment (IDE) designed specifically for working with the **R** programming language. It provides a user-friendly interface that includes a source editor, console, environment pane, and tools for plotting, debugging, version control, and package management. RStudio supports both **R** and Python and is widely used for data analysis, statistical modeling, and reproducible research. It also integrates seamlessly with tools like **R** Markdown, Shiny, and Quarto, making it popular among data scientists, statisticians, and educators.

5.2.2 Visual Studio Code (VS Code)

VS Code is a versatile code editor that supports multiple programming languages, including **R**. With the **R** extension for VS Code, users can write and execute **R** code, access **R**'s console, and utilize features like syntax highlighting, code completion, and debugging. While not as specialized as RStudio for **R** development, VS Code offers a lightweight alternative with extensive customization options and support for various programming tasks.

5.2.3 Positron

Positron IDE is the next-generation integrated development environment developed by Posit, the company behind RStudio. Designed to be a modern, extensible, and language-agnostic IDE, Positron builds on the strengths of RStudio while supporting a broader range of languages and workflows, including **R**, Python, and Quarto.

5.3 RStudio Layout

RStudio consists of several panes: - **Source**: Where you write scripts and markdown documents. - **Console**: Where you type and execute **R** commands. - **Environment/History**: Shows your variables and command history. - **Files/Plots/Packages/Help/Viewer**: For file management, viewing plots, managing packages, accessing help, and viewing web content.

5.4 R Scripts

R scripts are plain text files containing **R** code. You can create a new script in RStudio by clicking `File > New File > R Script`.

5.5 R Help

Use `?function_name` or `help(function_name)` to access help for any **R** function. For example:

```r
?mean
help(mean)
```

5.6 R Packages

Packages extend **R**'s functionality. There are thousands of packages available in **R** ecosystem. You may install them from different sources.

5.6.1 With Comprehensive R Archive Network (CRAN)

CRAN is the primary repository for **R** packages. It contains thousands of packages that can be easily installed and updated.

Install a package with:

```r
install.packages("package_name")
```

5.6.2 With Bioconductor

Bioconductor is a repository for bioinformatics packages in **R**. It provides tools for the analysis and comprehension of high-throughput genomic data.

Install Bioconductor packages using the `BiocManager` package:

```r
BiocManager::install("package_name")
```

5.6.3 From GitHub

Many of the authors of **R** packages host their work on GitHub. You can install these packages using the `devtools` package:

```r
devtools::install_github("username/package_name")
```

5.6.4 Load a package

Once a package is installed, you need to load it into your **R** session to use its functions:

```r
library(package_name)
```

Alternatively, you may use a function in the package with `package_name::function_name()` without loading the entire package.

5.7 R Markdown

R Markdown allows you to combine text, code, and output in a single document. Create a new **R** Markdown file in RStudio via `File > New File > R Markdown...`.

Recently, the posit team has developed a new version of the **R** Markdown called quarto document, with the file extension `.qmd`. It is still under rapid development.

5.8 Vectors

Vectors are the most basic data structure in **R**.

```r
x <- c(1, 2, 3, 4, 5)
x
```

```
[1] 1 2 3 4 5
```

You can perform operations on vectors:

```r
x * 2
```

```
[1]  2  4  6  8 10
```

5.9 Data Sets

Data frames are used for storing data tables. Create a data frame:

```r
df <- data.frame(Name = c("Alice", "Bob"), Score = c(90, 85))
df
```

```
   Name Score
1 Alice    90
2   Bob    85
```

You can import data from files using `read.csv()` or `read.table()`.

This appendix is adapted from Why R?.