STAT 8670 - Computational Methods in Statistics

Chi-Kuang Yeh

2025-09-03

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

Office Hour

14:00–15:00 on Tuesday and Wednesday.

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
Introduction to R Programming	1–2
Optimization	3–
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.

```
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.

```
# 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 
```

```
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.

```r
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
```

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

```
x1 x2 
 3  3 
```

```r
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.

```
## 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 becor
```

```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.

A comprehensive reference for all the *tidyverse* tools is R for Data Science.

A comprehensive reference for *ggplot2* is ggplot2: Elegant Graphics for Data Analysis.

2 Numerical Approaches and 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.

There is a general principle that will be repeated in this chapter that Kenneth Lange calls *optimization transfer* in his 1999 paper. The basic idea applies to the problem of maximizing a function f.

1. Direct optimize the function f.
    - It can be difficult
2. Optimize a surrogate function g that is easier to optimize than f.
3. So here, instead of optimize f, we optimize g.

Note 1: steps 2&3 are repeated until convergence.

Note 2: maximizing f is equivalent to minimizing $-f$.

Note 3: the surrogate function g should be chosen such that it is easier to optimize than f.

For instance, for a linear regression

$$y = X\beta + \varepsilon. \tag{2.1}$$

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 Theory versus Computation

One confusing aspect of statistical computing is that often there is a disconnect between what is printed in a statistical computing textbook and what should be implemented on the computer.

- In textbooks, simpler to **present solutions as convenient mathematical formulas whenever possible**, in order to communicate basic ideas and to provide some insight.

  - However, directly translating these formulas into computer code is usually not advisable because there are many problematic aspects of computers that are simply not relevant when writing things down on paper.

Some potential issues includ:

1. Memory overflow: The computer has a limited amount of memory, and it is possible to run out of memory when working with large datasets or complex models.
2. Numerical Precision: Sometimes, due to the cut precision of floating-point arithmetic, calculations that are mathematically equivalent can yield different results on a computer.

   - Example 1: round $1/3$ to two decimal places, we get 0.33. Then, $3 \cdot (1/3)$ is exactly 1, but $3 \cdot 0.33$ is 0.99.
   - Example 2: $1 - 0.99999999$ is 0.00000001 (=1E-8), but if we round 0.99999999 to two decimal places, we get 1.00, and then $1 - 1.00$ is 0. If we round 0.00000001 to two decimal places, we get 0.00.
   - Example 3: π

3. (Lienar) Dependence: The detection of linear dependence in matrix computations is influenced by machine precision. Since computers operate with finite precision, situations often arise where true linear dependence exists, but the computer cannot distinguish it from independence.

   - Example: Consider the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

   The 3rd column is a linear combination of the first two columns (i.e., col3 = col1 + col2). However, due to machine precision limitations, the computer might not recognize this exact linear dependence, leading to numerical instability in computations involving this matrix. With a small distortion, we have

$$B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 + 10^{-5} \end{pmatrix}$$

```r
A <- matrix(
  c(1, 2, 3,
    4, 5, 6,
    7, 8, 9),
  nrow = 3, ncol = 3, byrow = TRUE)
B <- A
```

```r
B[3, 3] <- B[3, 3] + 1E-5

qr(A)$rank
```

```
[1] 2
```

```r
qr(B)$rank
```

```
[1] 3
```

2.2 Matrix Inversion

In many statistical analyses, such as linear regression and specify the distribution (such as normal distribution), matrix inversion plays a central role.

2.2.1 Example 1: Normal distribution

We know that, a normal density with the parameters mean μ and standard deviation σ is

$$f\left(x \mid \mu, \sigma^2\right) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\}$$

or we may work on the multivariate normal distribution case which is a bit more involved.

$X = (X1, \dots, X_d)$ is said to be a multivariate normal distribution if and only if it is a linear comibnation of independent and identically distributed standard normals:

$$X = CZ + \mu, \quad Z = (Z_1, \dots, Z_d), \quad Z_i \overset{iid}{\sim} N(0,1).$$

The property of the multivariate normal are:

- mean vector: $E(X) = \mu$
- variance: $Var(X) = CZC^\top = Cvar(Z)C^\top := \Sigma$

Notation: $X \sim N(\mu, \Sigma)$.

PDF:

$$f(x \mid \mu, \Sigma) = (2\pi)^{-d/2} \cdot \exp\left\{-\frac{1}{2}(x-\mu)'\Sigma^{-1}(x-\mu) - \frac{1}{2}\log|\Sigma|\right\}.$$

Some of the potential ways to do this is to take logarithm of the PDF (Think about why).

2.2.2 Example 2: Linear regression

Recall the linear regression model . The OLE for β is given by $\hat{\beta} = (X^\top X)^{-1} X^\top y$.

We can solve this using the R command

```r
beta_hat <- solve(t(X) %*% X) %*% t(X) %*% y
```

where `solve()` is the R function for matrix inversion. However, it is not a desired way (think about why).

A better way is to go back to the formula, and look at

$$X^\top X \beta = X^\top y,$$

and solve this using the R command

```r
solve( crossprod(X), crossprod(X, y) )
# this is the same as
# solve(t(X) %*% X, t(X) %*% y)
```

Here, we avoid explicitly calculating the inverse of $X^\top X$. Instead, we use gaussian elimination to solve the system of equations, which is generally more numerically stable and efficient.

2.2.2.1 Speed comparison

```r
set.seed(2025-09-03)
X <- matrix(rnorm(5000 * 100), 5000, 100)
y <- rnorm(5000)
library(microbenchmark)
microbenchmark(solve(t(X) %*% X) %*% t(X) %*% y)
```

```
Unit: milliseconds
                             expr      min       lq
 solve(t(X) %*% X) %*% t(X) %*% y 28.83505 30.16593
     mean   median       uq      max neval
 31.96782 30.79489 32.63315 111.0151   100
Warning message:
In microbenchmark(solve(t(X) %*% X) %*% t(X) %*% y) :
  less accurate nanosecond times to avoid potential integer overflows
```

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

```
Unit: milliseconds
                                 expr      min       lq
     solve(t(X) %*% X) %*% t(X) %*% y 28.90135 30.11608
 solve(crossprod(X), crossprod(X, y)) 25.05859 25.27480
     mean   median       uq      max neval
 31.78686 31.38513 32.66482 53.03354   100
 26.15771 25.81678 26.89188 29.12045   100
```

2.2.3 Take home message:

The take home here is that the issues arise from the finite precision of computer arithmetic and the limited memory available on computers. When implementing statistical methods on a computer, it is crucial to consider these limitations and choose algorithms and implementations that are robust to numerical issues.

2.2.4 Multi-collinearity

The above approach may break down when there is any multi-colinearity in the X matrix. For example, we can tack on a column to X that is very similar (but not identical) to the first column of X.

```
set.seed(7777)
N <- 3000
K <- 100
y <- rnorm(N)
X <- matrix(rnorm(N * K), N, K)
W <- cbind(X, X[, 1] + rnorm(N, sd = 1E-15))
```

```
solve(crossprod(W), crossprod(W, y))

Error in `solve.default()`:
! system is computationally singular: reciprocal condition number = 1.36748e-32
```

The algorithm does not work because the cross product matrix $W^\top W$ is **singular**. In practice, matrices like these can come up a lot in data analysis and it would be useful to have a way to deal with it automatically.

R takes a different approach to solving for the unknown coefficients in a linear model. R uses the QR decomposition, which is not as fast, but has the added benefit of being able to automatically detect and handle colinear columns in the matrix.

Here, we use the fact that X can be decomposed as $X = QR$, where Q is an orthonormal matrix and R is an upper triangular matrix. Given that, we can rewrite $X^\top X\beta = X^\top y$ as

$$\begin{aligned} R^\top Q^\top QR\beta &= R^\top Q^\top y \\ R^\top IR\beta &= R^\top Q^\top y \\ R^\top R\beta &= R^\top Q^\top y, \end{aligned}$$

this leads to $R\beta = Q^\top y$. Now we can perform the Gaussian elimination to do it. Because R is an upper triangular matrix, the computational speed is much faster. Here, we **avoid to compute the cross product** $X^\top X$, which is numerical unstable if it is not *standardized* properly

We can see in R code that even with our singular matrix W above, the QR decomposition continues without error.

```r
Qw <- qr(W)
str(Qw)
```

```
List of 4
 $ qr   : num [1:3000, 1:101] 54.43933 0.00123 -0.02004 -0.00671 -0.00178 ...
 $ rank : int 100
 $ qraux: num [1:101] 1.01 1.01 1.01 1 1 ...
 $ pivot: int [1:101] 1 2 3 4 5 6 7 8 9 10 ...
 - attr(*, "class")= chr "qr"
```

Note that the output of `qr()` computes the rank of W to be 100, not 101 as the last column is collinear to the 1st column. From there, we can get $\hat{\beta}$ if we want using `qr.coef()`,

```r
betahat <- qr.coef(Qw, y)
head(betahat, 3)
```

```
[1]  0.024314718  0.000916951 -0.005980588
```

```r
tail(betahat, 3)
```

```
[1]  0.01545039 -0.01010440           NA
```

Q: Why there is an `NA`?

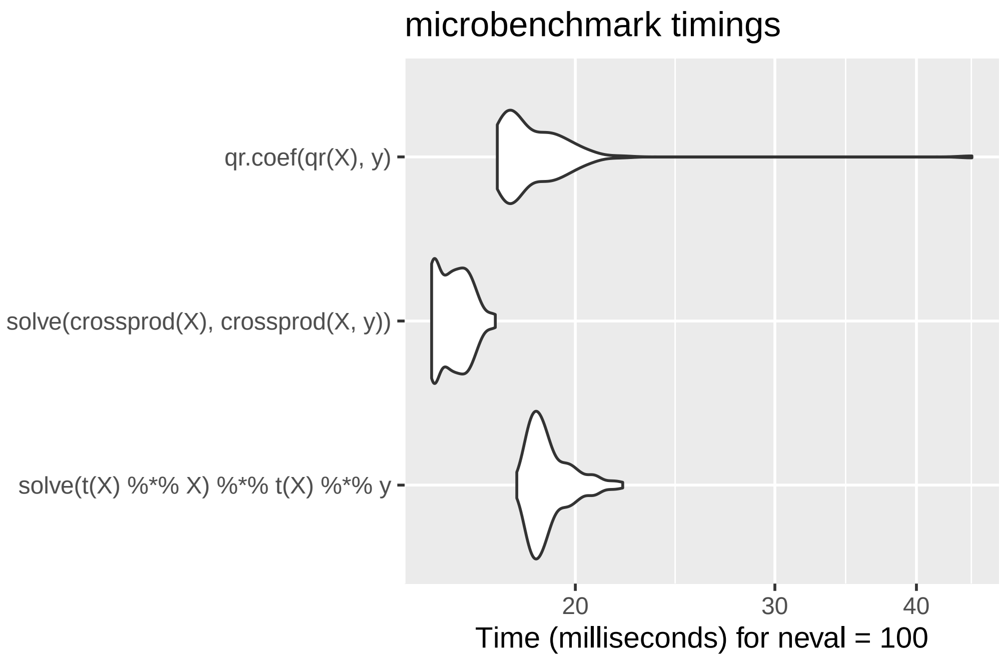
2.2.5 Trade-off

There isn't always elegance and flourish. When we take the robust approach, we accept that it comes at a cost.

```r
library(ggplot2)
library(microbenchmark)
m <- microbenchmark(solve(t(X) %*% X) %*% t(X) %*% y,
                    solve(crossprod(X), crossprod(X, y)),
                    qr.coef(qr(X), y))
```

```
Warning in microbenchmark(solve(t(X) %*% X) %*% t(X) %*% y, solve(crossprod(X),
: less accurate nanosecond times to avoid potential integer overflows
```

```r
autoplot(m)
```



Compared with the approaches discussed above, this method performs similarly to the naive approach but is much more stable and reliable.

In practice, we rarely call functions such as `qr()` or `qr.coef()` directly, since higher-level functions like lm() handle these computations automatically. However, in certain specialized and performance-critical settings, it can be advantageous to use alternative matrix decompositions to compute regression coefficients, especially when the computation must be repeated many times in a loop (i.e., *Vectorization*)

2.2.6 Multivariate Normal revisit

Computing the multivariate normal (MVN) density is a common task, for example, when fitting spatial models or Gaussian process models. Because maximum likelihood estimation(MLE) and likelihood ratio tests (LRT) often require evaluating the likelihood many times, efficiency is crucial.

After taking the log of the MVN density, we have

$$\ell(x \mid \mu, \Sigma) := \log \{f(x \mid \mu, \Sigma)\} = -\frac{d}{2}\log(2\pi) - \frac{1}{2}\log|\Sigma| - \frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu).$$

On the right hand side, the first term is a constant, the second term is linear, and the last term is quadratic, which requires much more computational power.

2.2.6.1 A Naive Implementation

We first center the data $z := x - \mu$. Then we have $z^\top \Sigma^{-1} z$. This simiplified the question for a bit.

Here, much like the linear regression example above, the key bottleneck is the inversion of the p-dimensional covariance matrix Σ. If we take z to be a $p \times 1$ column vector, then a literal translation of the mathematics into R code might look something like this,

```r
t(z) %*% solve(Sigma) %*% z
```

To illustrate, let's simulate some data and compute the quadratic form the naive way:

```r
set.seed(2025-09-03)

# Generate data
z <- matrix(rnorm(200 * 100), 200, 100)
S <- cov(z)

# Naive quadratic form
quad.naive <- function(z, S) {
  Sinv <- solve(S)
  rowSums((z %*% Sinv) * z)
}

library(dplyr)
quad.naive(z, S) %>% summary()
```