STAT 8670 - Computational Methods in Statistics

Chi-Kuang Yeh

2025-09-14

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.

Grade Distribution

- Assignments: 40%
- Term Exam 1: 15%
- Term Exam 2: 15%
- Final Project: 30%

Assignment

- ☒ Assignment 1: Due on September 12th, 2025
- ☐ Assignment 2: Due on September 19th, 2025

Midterm

- ☐ Midterm 1 on October 8, 2025
- ☐ Midterm 2 on November 12, 2025

Final Project

- ☐ Report: Due Date TBA
- ☐ Presentation: Due Date 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
Numerical Approaches and 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.

```
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!

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

```
[1] 3.141593
```

```
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   1.5.2
v ggplot2   4.0.0     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
```

```
## 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

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

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

```
[1] 0.37
```

```
## 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

```
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 `%$%`:

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

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

```
# 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 **iteratively**!

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.

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

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

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

```
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

```
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

```
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.

```
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 $\widehat{\beta}$ if we want using `qr.coef()`,

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

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

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

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

Q: Why there is an `NA`?

2.2.5 Trade-off

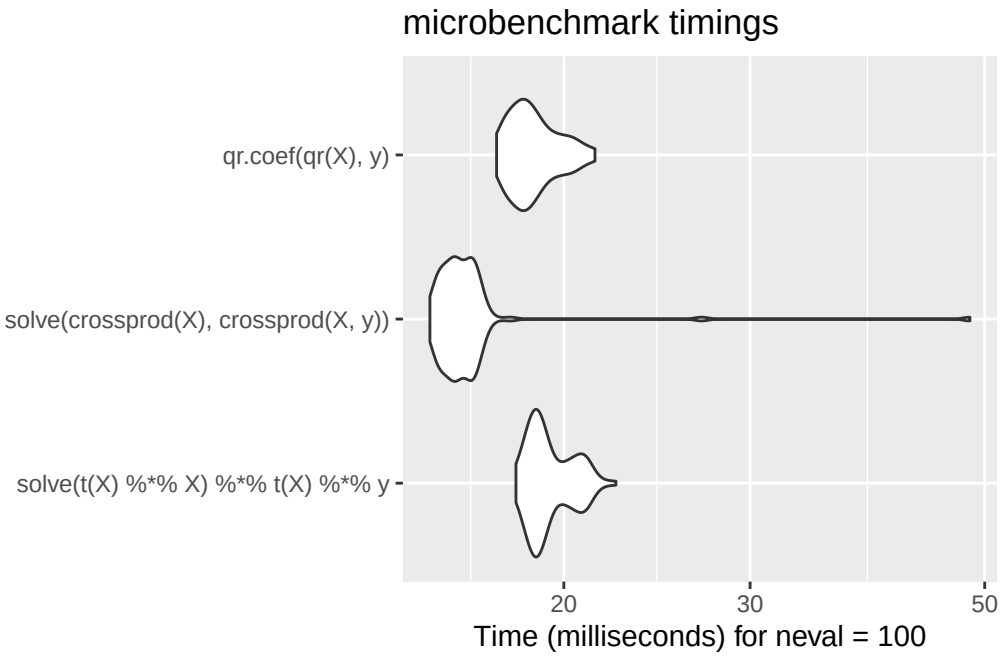
There isn't always elegance and flourish. When we take the robust approach, we accept that it comes at a cost.

```
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
```

```
autoplot(m)
```

```
Warning: `aes_string()` was deprecated in ggplot2 3.0.0.
i Please use tidy evaluation idioms with `aes()`.
i See also `vignette("ggplot2-in-packages")` for more information.
i The deprecated feature was likely used in the microbenchmark package.
  Please report the issue at
  <https://github.com/joshuaulrich/microbenchmark/issues/>.
```



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,

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

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

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

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

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

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

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  70.67   93.61   99.94  100.54  107.31  126.73
```

2.2.6.2 A Better Way: Cholesky Decomposition

Because the covariance matrix is symmetric and positive definite, we can exploit its **Cholesky decomposition**. That is, we write $\Sigma = R^\top R$, where R is a upper triangular matrix. Then,

$$z^\top \Sigma^{-1} z = z^\top (R^\top R)^{-1} z = z^\top R^{-1} R^{-\top} z = (R^{-\top} z)^\top (R^{-\top} z) := v^\top v.$$

Note that $v \in \mathbb{R}^p$ is the solution to the linear system $R^\top v = z$. Because R is upper triangular, we can solve this system efficiently using back substitution. Also, we can solve this without doing the inversion.

Once we have v we can compute its quadratic form $v^\top v$ by the `crossprod()` function.

```r
quad.chol <- function(z, S) {
  R <- chol(S)
  v <- backsolve(R, t(z), transpose = TRUE)
  colSums(v * v)
}

quad.chol(z, S) %>% summary()
```

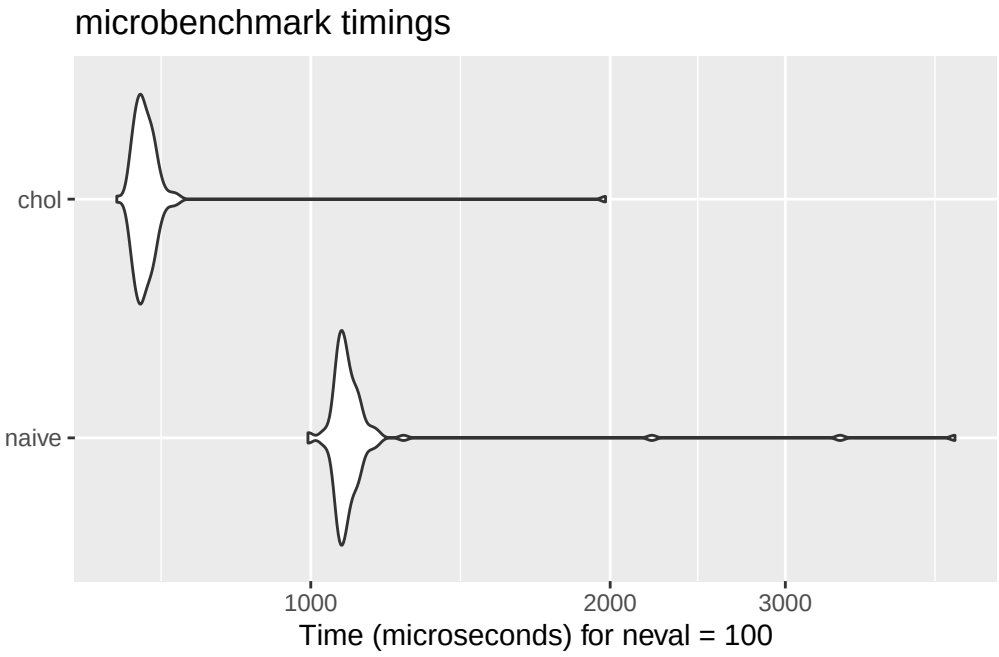
```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  70.67   93.61   99.94  100.54  107.31  126.73
```

2.2.6.3 By product

Another benefit of the Cholesky decomposition is that it gives us a simple way to compute the log-determinant of Σ. The log-determinant of Σ is simply two times the sum of the log of the diagonal elements of R. (Why?)

2.2.6.4 Performance comparison

```
library(microbenchmark)
library(ggplot2)
m2 <- microbenchmark(
  naive = quad.naive(z, S),
  chol  = quad.chol(z, S)
)
autoplot(m2)
```



Q: Why one is faster than the other?

2.2.6.5 Take home message 2

The naive algorithm simply inverts the covariance matrix. The Cholesky-based approach, on the other hand, exploits the fact that covariance matrices are symmetric and positive definite. This results in an implementation that is both faster and numerically more stable—exactly the kind of optimization that makes a difference in real-world statistical computing.

Thus, a knowledge of statistics and numerical analysis can often lead to better algorithms, often invaluable!

2.3 Nonlinear functions

On the top, we have *linear functions*, such as $y = f(x) = ax + b$ or in the linear regression $y = X\beta + \epsilon$. It is a small class of the functions, and may be relatively limited.

E.g., what if we have a quadratic relationship? Then $y = f(x) = ax^2 + bx + c$.

Such nonlinear relationship is very common, , such as $f(x) = a\sin(bx+c)$ or $f(x) = a\exp(bx)+c$, and they may not have a closed-form solution like in the linear regression case.

From now on, we will be talking about the numerical approaches to solve these problems.

2.4 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**: 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.

2.5 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 rely on the **Karush–Kuhn–Tucker** (KKT) conditions.

2.5.1 Root finding

The *root finding* is probably the first numerical approach you learned in the numerical analysis course. Consider a function $f : \mathbb{R} \to \mathbb{R}$. The point $x \in \mathbb{R}$ is called a *root* of f if $f(x) = 0$.

Q: Why do we care about the root finding?

This idea has broad applications. While finding the values of x such that f(x) = 0 is useful in many settings, a more general task is to determine the values of x for which f(x) = y. The same techniques used to find the roots of a function can be applied here by rewriting the problem as

$$\tilde{f}(x) := f(x) - y = 0.$$

In this way, new function $\tilde{f}(x)$ has a root at the solution to, $f(x) = y$, original equation.

For linear function, it is trivial. For quadratic function, we can use the quadratic formula, i.e.,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

However, for more complex functions, we need to use numerical methods to solve it iteratively. Below, we are going to go over some numerical algorithms.

2.5.2 One-dimensional case

We first look at the one-dimensional case. The function we want to optimize is

$$f(x) = x^3 - x + 1$$

2.5.3 Bisection method

Bisection method is just like a *binary search*.

Step 1. Selection two points $a, b \in \chi \subseteq \mathbb{R}$, where χ is the domain of f. Make sure that a and b have opposite signs, i.e., $f(a)f(b) < 0$.

Step 2. Compute the midpoint $c = (a + b)/2$.

Step 3. Evaluate and check the sign of $f(c)$. If $f(c)$ has the same sign as $f(a)$, then set $a = c$. Otherwise, set $b = c$.

Step 4. Iterate Steps 2 and 3 until the interval $[a, b]$ is sufficiently small.

The intuition here is that we are shirking the search space χ by half in each iteration.

Q: Why this algorithm work and what are the assumptions? 1. We require the function to be continuous 2. We require the function to have opposite signs at the two endpoints $a, b \in \chi \subseteq \mathbb{R}$. 3. We do not require the differentiability!

Q: But what's the cost?

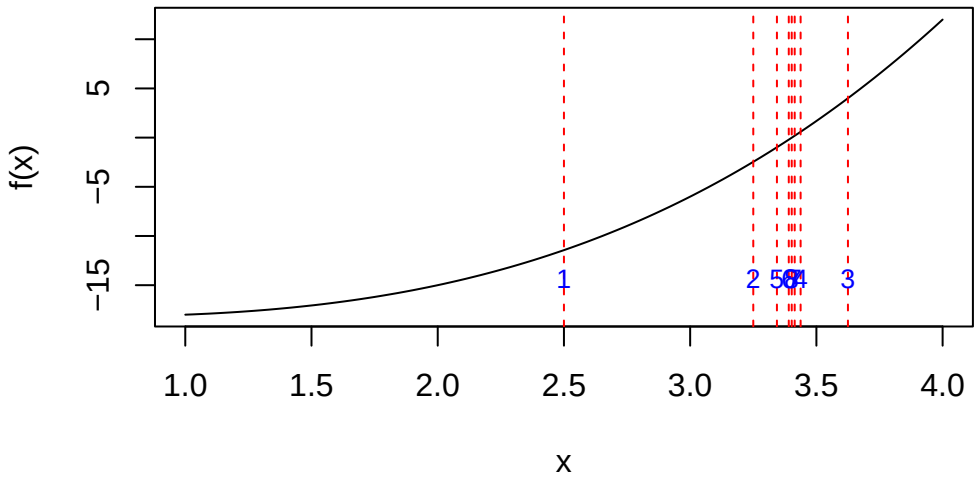
Q: Can this work for every function?

2.5.3.1 Example

Suppose the design region is

```
a <- 1
b <- 4
curve(0.5*x^3 - 0.5*x - 18, from = a, to = b, xlab = "x", ylab = "f(x)")
fun_obj <- function(x) 0.5*x^3 - 0.5*x - 18

my_bisec <- function(fun_obj, a, b, tol = 1E-2, ind_draw = FALSE) {
  if (fun_obj(a) * fun_obj(b) > 0) {
    stop("f(a) and f(b) must have opposite signs!")
  }
  iter <- 0
  while ((b - a) / 2 > tol) {
    c <- (a + b) / 2

    if (ind_draw == TRUE) {
    # Draw vertical line
    abline(v = c, col = "red", lty = 2)
    # Label the iteration above the x-axis
    text(c, par("usr")[3] + 2, labels = iter + 1, col = "blue", pos = 3, cex = 0.8)
    }


    if (fun_obj(c) == 0) {
      return(c)
    } else if (fun_obj(a) * fun_obj(c) < 0) {
      b <- c
    } else {
      a <- c
    }
    iter <- iter + 1
  }
  val_x <- (a + b) / 2
```

```
  val_fx <- fun_obj(val_x)
  return(list(root = val_x, f_root = val_fx, iter = iter))
}

# Run it
res_plot <- my_bisec(fun_obj, a, b, ind_draw = TRUE)
```

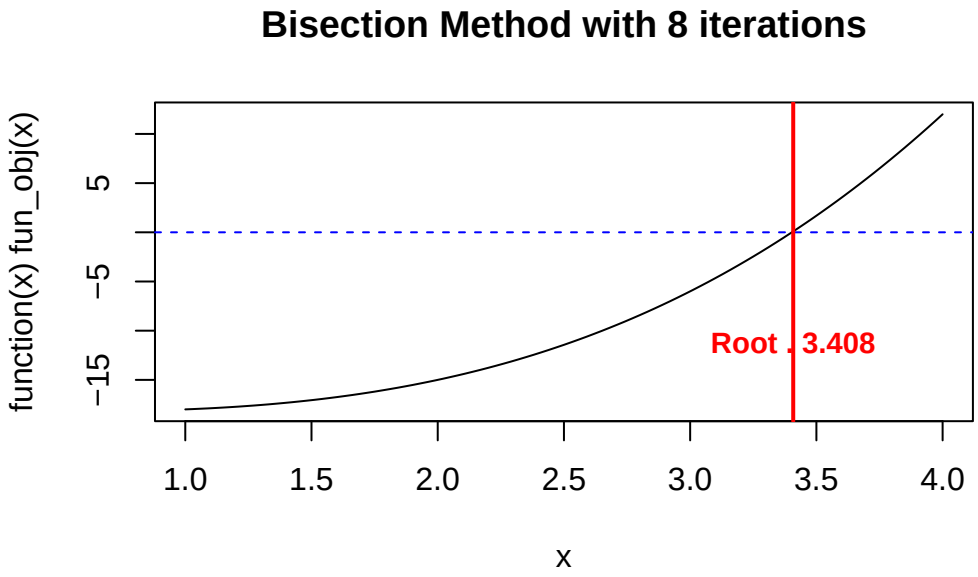



```
res_plot
```

```
$root
[1] 3.408203

$f_root
[1] 0.09048409

$iter
[1] 8
```

```
res <- my_bisec(fun_obj, a, b)
plot(function(x) fun_obj(x), from = a, to = b)
abline(h = 0, col = "blue", lty = 2)
```

```r
title(main = paste0("Bisection Method with ", res$iter, " iterations"))
abline(v = res$root, col = "red", lwd = 2)
text(res$root, par("usr")[3] + 5,
     labels = paste0("Root   ", round(res$root, 3)),
     col = "red", pos = 3, cex = 0.9, font = 2)
```



2.5.4 Newton-Raphson method

The Newton-Raphson method (or simply Newton's method) is an iterative numerical method for finding successively better approximations to the roots (or zeroes) of a real-valued function.

Here, we assume that the function f is differentiable. The idea here is to use the Taylor expansion of the function. Suppose we are search a small neighbour of the solution $x \in \mathbb{R}$, say $x_j \in \mathbb{R}$ is a small number. Then Then we first order Taylor series to approximate $f(x_j + h)$ around x_j is

$$f(x) \approx f(x_j) + f'(x_j)(x - x_j),$$

where $f'(x) := \partial_x f(x)$ is the first derivative of $f(x)$. So the root of this approximation can be improved by updating its place to where $f(x_{j+1}) = 0$.

So if $f(x_j + h)$ is the root, then we have

$$0 = f(x_j) + f'(x_j)h \implies h = -\frac{f(x_j)}{f'(x_j)}.$$

Then, we can come back to $x_{j+1} = x_j + h$, and plug the value of h in from above, we have

$$x_{j+1} = x_j - \frac{f(x_j)}{f'(x_j)}.$$

The algorithm is given as below:

Let $f : \mathbb{R} \to \mathbb{R}$ be differentiable.

Step 0: Choose a function $f(x)$: This is the function for which you want to find a root (i.e., solve $f(x) = 0$).

Step 1: Calculate the derivative $f'(x)$: You will need it to apply the formula.

Step 2: Make an initial guess x_0: Select a starting point x_0 near the expected root.

Step 3: Update the estimate: Use the Newton's method formula to compute the next estimate x_1 using x_0 by

$$x_{j+1} = x_j - \frac{f(x_j)}{f'(x_j)}.$$

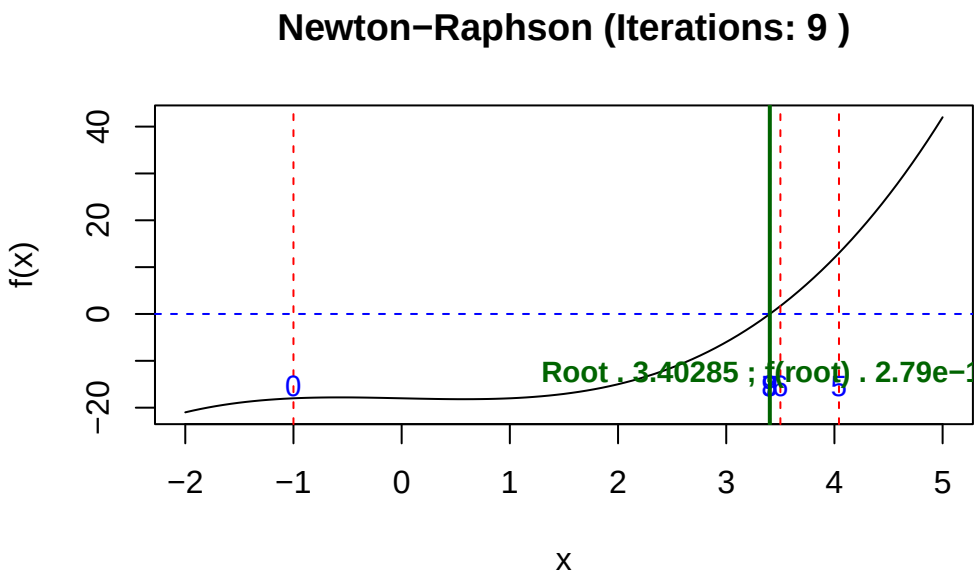
Step 4: Repeat Steps 2 and 3 until the values converge to a root or reach a desired tolerance.

```r
## Function and derivative
f  <- function(x) 0.5*x^3 - 0.5*x - 18
df <- function(x) 1.5*x^2 - 0.5

## Newton-Raphson with iterate tracking
newton_raphson <- function(f, df, x0, tol = 1e-5,
                           maxit = 100, eps = 1e-5) {
  x <- x0
  xs <- x0      # store iterates (x0, x1, x2, ...)
  for (k in 1:maxit) {
    fx  <- f(x)
    dfx <- df(x)
    x_new <- x - fx/dfx
    xs <- c(xs, x_new)
    if (abs(x_new - x) < tol || abs(fx) < tol) {
      return(list(root = x_new, iter = k, path = xs))
    }
```

```
    x <- x_new
  }
  list(root = x, iter = maxit, path = xs)
}

## Starting point
```

If we start at -1 which is far away from

```
x0 <- -1
res <- newton_raphson(f, df, x0)
a <- -2; b <- 5
plot(function(x) f(x), from = a, to = b,
     xlab = "x", ylab = "f(x)",
     main = paste("Newton-Raphson (Iterations:", res$iter, ")"))
abline(h = 0, col = "blue", lty = 2)

## Draw vertical lines for each iterate with labels 0,1,2,...
for (i in seq_along(res$path)) {
  xi <- res$path[i]
  abline(v = xi, col = "red", lty = 2)
  text(xi, par("usr")[3] + 2, labels = i - 1, col = "blue", pos = 3, cex = 0.9)
}

## Final root marker + label
abline(v = res$root, col = "darkgreen", lwd = 2)
text(res$root, par("usr")[3] + 5,
     labels = paste0("Root   ", round(res$root, 5),
                     " ; f(root)   ", signif(f(res$root), 3)),
     col = "darkgreen", pos = 3, cex = 0.95, font = 2)
```



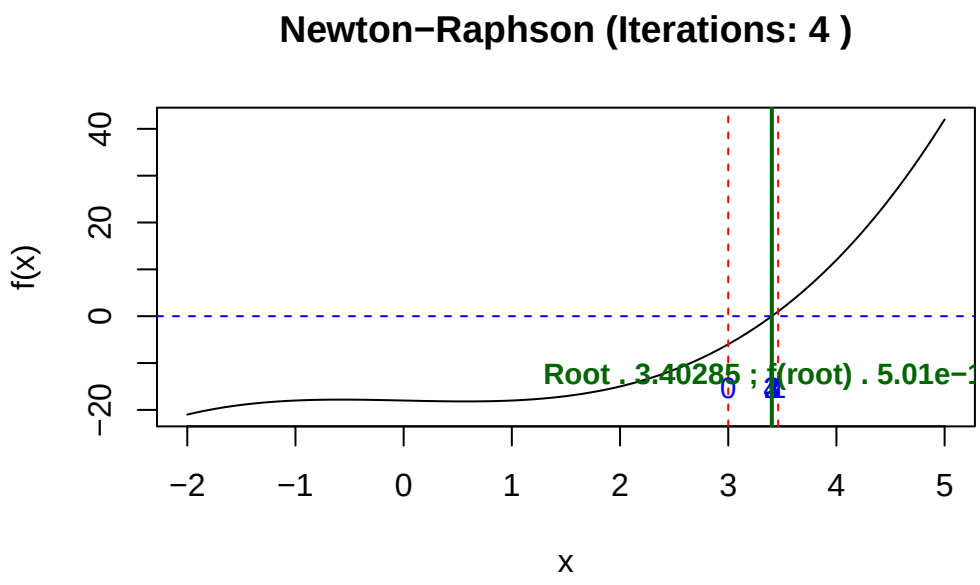

```
res
```

```
$root
[1] 3.402848

$iter
[1] 9

$path
 [1] -1.000000 17.000000 11.387991  7.704327  5.368534  4.042133  3.500619
 [8]  3.405629  3.402850  3.402848
```

If we start at 3 which is near to the point

```
x0 <- 3
res <- newton_raphson(f, df, x0)
## Plot range that shows the iterates and root
a <- -2; b <- 5
plot(function(x) f(x), from = a, to = b,
     xlab = "x", ylab = "f(x)",
     main = paste("Newton-Raphson (Iterations:", res$iter, ")"))
abline(h = 0, col = "blue", lty = 2)
```

```r
## Draw vertical lines for each iterate with labels 0,1,2,...
for (i in seq_along(res$path)) {
  xi <- res$path[i]
  abline(v = xi, col = "red", lty = 2)
  text(xi, par("usr")[3] + 2, labels = i - 1, col = "blue", pos = 3, cex = 0.9)
}

## Final root marker + label
abline(v = res$root, col = "darkgreen", lwd = 2)
text(res$root, par("usr")[3] + 5,
     labels = paste0("Root   ", round(res$root, 5),
                     " ; f(root)   ", signif(f(res$root), 3)),
     col = "darkgreen", pos = 3, cex = 0.95, font = 2)
```




```r
res
```

```
$root
[1] 3.402848

$iter
[1] 4
```

```
$path
[1] 3.000000 3.461538 3.403866 3.402848 3.402848
```

Remarks:

- Assumptions: f is differentiable in a neighborhood of the root r.
- Failure cases: if $f'(x_j) = 0$ (or is very small), the update is ill-defined/unstable; if the initial guess is far, the method can diverge or jump to a different root.
- Practical checks: stop when $|f(x_j)| \leq \delta$ or $|x_{j+1} - x_j| \leq \delta$ is below tolerance δ.

2.5.5 Second Method

The secant method can be thought of as a finite-difference approximation of Newton's method, so it is considered a quasi-Newton method. It is simialr to Newton's method, but it does not require the computation of the derivative of the function. Instead, it approximates the derivative using two previous points.

In the second method, we require the **first two point**s, say $x_0, x_1 \in \mathbb{R}$. Then, we can approximate the derivative of f at x_1 using the finite difference formula. Instead of calculate the derivative $f'(x_1)$, we approximate it as using the secant line. In calculate, we know that, $f'(x_1) \approx \frac{f(x_1)-f(x_0)}{x_1-x_0}$, if x_1 and x_0 are close. Then, we can plug this approximation into the Newton's update formula to get

$$x_j = x_{j-1} - f(x_{j-1})\frac{x_{j-1} - x_{j-2}}{f(x_{j-1}) - f(x_{j-2})} = \frac{x_{j-2}f\left(x_{j-1}\right) - x_{j-1}f\left(x_{j-2}\right)}{f\left(x_{j-1}\right) - f\left(x_{j-2}\right)}.$$

2.6 Hill climbing

In numerical analysis, *hill climbing* is a mathematical optimization technique which belongs to the family of *local search*. The Newton method and secant method can be thought as questions in hill climbing.

The algorithm starts with an arbitrary solution to a problem, then iteratively makes small changes to the solution, each time moving to a neighboring solution that is better than the current one. The process continues until no neighboring solution is better than the current solution, at which point the algorithm terminates.

In the world of optimization, finding the best solution to complex problems can be challenging, especially when the solution space is vast and filled with local optima.

2.6.1 In R

`uniroot()`, `optim()` , `nlm()`, and `mle()` functions. Note that you may *approximate the derivative/gradient.*

2.7 Converegence

In order to compare the efficiency of the set of algorithms, one may compare *their abilities* for finding the optimals. However, what if, say, two algorithms both can find optimals, which one is better? The **convergence rate** comes in. Convergence rate is a measure of how quickly an iterative algorithm approaches its limit or optimal solution, which mean, how fast the algorithm converges to the optimals.

In the previous lecture(s), we saw that we can use R functions such `microbenchmark::microbenchmark()`, to measure the performance. However, it may takes a long time and a lot of computational resources. For such cases, we may use the theoretical convergence rate to compare the efficiency of the algorithms.

The convergence rate is often classified into several categories. It acts like the sequence $\{x_j\}$ we learned in grade schools. Here, $\{x_j\}$ is a sequence of estimates generated by the algorithm at each iterations, and x^* is the true solution or optimal value we are trying to reach. The error at iteration n is defined as $e_j = d(x_j, x^*)$, where the typical metric here is the absolute distance $d(x_j, x^*) = |x_j - x^*|$ (note, in spaces, different metric to define the distance). The convergence rate describes how quickly this error sequence $\{e_j\}$ decreases as j increases. For

$$\lim_{j\to\infty} \frac{\left|x_{j+1} - x^*\right|}{\left|x_j - x^*\right|^q} = \mu.$$

2.7.1 Linear Convergence

If order $q = 1$ and $0 < \mu < 1$, the sequence $\{x_j\} \in \mathbb{R}^d$ converges to x^* linearly. That is, $x_j \to x^*$ as $j \to \infty$ in $\mathbb{R}^d$ if there existence a constant μ such that

$$\frac{\|x_{n+1} - x_\infty\|}{\|x_n - x_\infty\|} \leq \mu, \quad \text{as} \quad n \to \infty.$$

This means that the error decreases proportionally to its current value, leading to a steady but relatively slow convergence.

2.7.2 Superlinear Convergence

Suppose $\{x_n\}$ converges to x^*, if order $q = 1$ and $\mu = 0$, the sequence $\{x_n\}$ converges to x^* superlinearly. That is, x_n is said to be converges to x^* as $n \to \infty$ superlinearly if

$$\lim_{n\to\infty} \frac{\|x_{n+1} - x_\infty\|}{\|x_n - x_\infty\|} = 0.$$

It is clearly that the superlinear is a stronger condition than the linear convergence, such that $\mu = 0$.

2.7.3 Quadratic Convergence

If order $q = 2$ and $\mu > 0$, the sequence $\{x_n\}$ converges to x^* quadratically. That is, x_n is said to be converges to x^* as $n \to \infty$ quadratically if

$$\frac{\|x_{n+1} - x_\infty\|}{\|x_n - x_\infty\|^2} \leq \mu, \quad \text{as} \quad n \to \infty.$$

2.8 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)

2.8.1 Simulating Annealing

Simulated annealing (SA) is a **stochastic** technique for approximating the global optimum of a given function.

Inspired by the physical process of annealing in metallurgy, Simulated Annealing is a probabilistic technique used for solving both combinatorial and continuous optimization problems.

What is Simulated Annealing?

Simulated Annealing is an optimization algorithm designed to search for an optimal or near-optimal solution in a large solution space. The name and concept are derived from the process of annealing in metallurgy, where a material is heated and then slowly cooled to remove defects and achieve a stable crystalline structure. In Simulated Annealing, the "heat" corresponds to the degree of randomness in the search process, which decreases over time (cooling schedule) to refine the solution. The method is widely used in combinatorial optimization, where problems often have numerous local optima that standard techniques like gradient descent might get stuck in. Simulated Annealing excels in escaping these local minima by introducing controlled randomness in its search, allowing for a more thorough exploration of the solution space.

Some terminology:

- Temperature: controls how likely the algorithm is to accept worse solutions as it explores the search space.

Step 1 (Initilization): Begin with an initial solution S_o and an initial temperature T_o.

Step 2 (Neighborhood Search): At step j, a new solution S' is generated by making a small change (or perturbation) to S_j.

Step 3 (Evaluation): evaluate the objective function $f(S')$ Step 3.1: If $f^(S')$ is *better* than $f(S_j)$, we *accept* it and take it as S_{j+1}. Step 3.2: If $f(S')$ is *worse* than $f(S_j)$, we may still accept it with a certain probability $P(S_{j+1} = S') = \exp(-\Delta E/T_j)$, where E is the energy $f(S') - f(S_j)$.

Step 4 Cooling Schedule: Decrease the temperature according to a cooling schedule, e.g., $T_{j+1} = \alpha T_j$ where $\alpha \in (0, 1)$ is a cooling rate.

Step 5 (Evaluation): Repeat Steps 2 and 3 for a certain number of iterations or until convergence criteria are met.

Example:

Figure 1 in my paper

Advantages:

- Global optimization
- Flexibility
- Intuitive
- Derivative?

Limitations:

- Parameter semsitivity
- Computational time

- Slow convergence

2.8.1.1 R implementation

An paper about an implementation in R by Husmann et al. and another package called GenSA.

2.9 Genetic Algorithm

Genetic Algorithm (GA) is a metaheuristic optimization technique inspired by the process of natural evolution/selection.

GA are based on an analogy with the genetic structure and behavior of chromosomes of the population.

STEP 1: Start with an initial generation G_0 of potential solutions (individuals). Each individual is represented by a chromosome, which is typically encoded as a binary string, real-valued vector, or other suitable representation. Evaluate the objective function on those points.

Step 2: Generate the next generation G_{j+1} from the current generation G_j using genetic operators: a). Selection: Retain the individual that is considered as *good* b). Crossover: Create children variables from the parents c). Mutation

Step 3: Repeat Step 2 until the algorithm converges or reaches a stopping criterion.

2.9.1 Particle Swarm Optimization

Particle Swarm Optimization (PSO) was proposed by Kennedy and Eberhart in 1995. It is inspried by the movement of the species in nature, such as fishes or birds.

The algorithm is based on a *population*, not a single current point.

At iteration n of the algorithm, a particle has a velocity $v(n)$ that depends on the follows.

- The location of the best objective function value that it has encountered, $s(n)$.
- The location of the best objective function value among its neighbors, $g(n)$.
- The previous velocity $v(n–1)$.

The position of a particle x(n) updates according to its velocity:

$$x(n+1) = x(n) + v(n),$$

adjusted to stay within the bounds. The velocity of a particle updates approximately according to this equation:

$$v(n+1) = W(n)v(n) + r(1)(s(n) - x(n)) + r(2)(g(n) - x(n)).$$

Here, $r(1), r(2) \in [0, 1]$ are random scalar values, and $W(n)$ is an *inertia factor* that adjusts during the iterations. The full algorithm uses randomly varying neighborhoods and includes modifications when it encounters an improving point.

Note: There are A LOT of variations of the PSO and other swarm-based algorithms used in the literature.

In R, there is a PSO implementation in the `pso` package. The associated manual may be found here.

2.10 Multivariate Case

We need to calculate the gradient/Jacobian matrix and Hessian matrix.

2.10.1 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).

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:

```
?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:

```
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:

```
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:

```
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:

```
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**.

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

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

You can perform operations on vectors:

```
x * 2
```

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

5.9 Data Sets

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

```
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?.

6 Appendix: Distributions

6.1 Discrete Distributions

6.1.1 Discrete uniform distributions

if the variation can take on finite, countable values with equal probability, it is said to be discrete uniform. The probability mass function of a discrete unfiromt random variable X is given by

$$f(x) = \frac{1}{x_n - x_1 + 1}, \quad \text{for} \quad x = x_1, \dots, x_n.$$

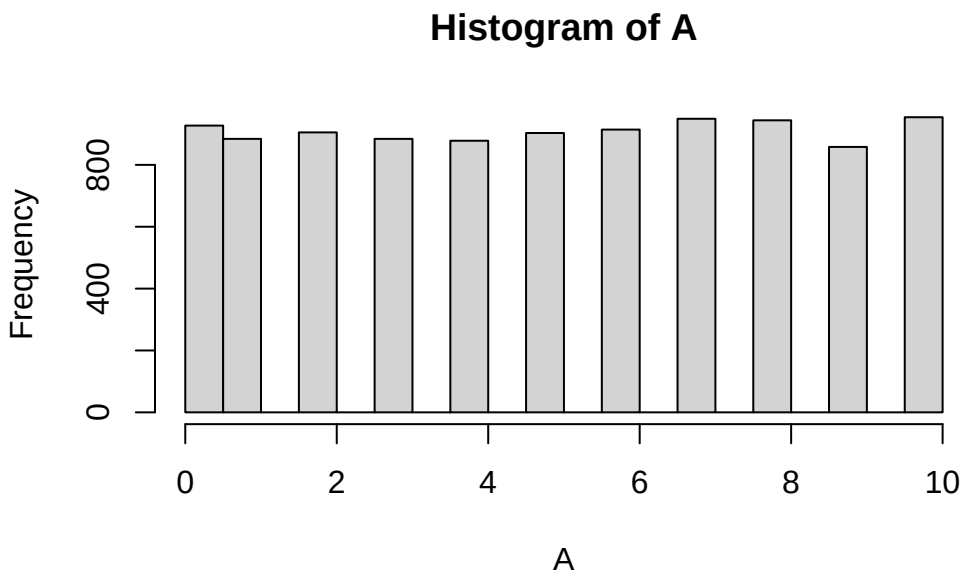
The mean and variance of the discrete uniform distribution are

$$\mathbb{E}X = \frac{x_1 + x_n}{2}, \quad \text{and} \quad \mathbb{V}ar(X) = \frac{(x_n - x_1 + 1)^2 - 1}{12}.$$

6.1.1.1 Implementation in R

There is no buildt-in function t osimlate from the discrete uniform distributions. Hence, we need to download additional packages.

```
pacman::p_load(extraDistr)
library(extraDistr)
# Generate random sample from Uniform(0, 10)
A <- rdunif(10000, 0, 10)

# Histogram of the sample
hist(A)
```




```
# Sample mean and variance
mean(A)
```

```
[1] 5.0314
```

```
var(A)
```

```
[1] 10.06622
```

```
# Theoretical mean of Uniform(a, b) is (a + b)/2
(mean.est <- (min(A) + max(A)) / 2)
```

```
[1] 5
```

```
# Theoretical variance of Uniform(a, b) is (b - a)^2 / 12
(var.est <- ((max(A) - min(A))^2) / 12)
```

```
[1] 8.333333
```

6.1.2 Bernoulli Distribution

There are only 2 possible outcomes in an experiment: *True* and *False*, with probabiltiy θ and $1-\theta$, respectively, the random variable X follows a Bernoulli distribution, denoted by $X \sim \text{Bern}(\theta)$. The probabiltiy distribution has probablity mass function as

$$f(x) = \theta^x (1-\theta)^{1-x}, \quad x = 0, 1.$$

Note that Bernoulli distribution is a special case of the Binomial distribution with $n = 1$.

6.1.2.1 Implementation in R

```
rbinom(10, 1, 0.7) # 0.7 is the probability of success
```

6.1.3 Binomial Distribution

Bernoulli trials (experiment where a Bernoulli Distribution applies) are not very relaise in real-life scenario. In staitsitcs, repeated experiments are important. When the experiments is repeated, θ is the same for each of the trials, and the trials are indpeendnet, and they are only two mutally excluses outcomes (T, and F), we can model this using the *Binomial Distribution*. If X follows the Binomial Distribution, it has a probablitliy mass function as

$$b(x; n, \theta) = \binom{n}{x} \theta^x (1-\theta)^{n-x}, x = 0, 1, 2, 3, \dots, n.$$

The Binomial Distribution is useful to predict the number of heads in n coin tosses, the number of people infected with a disease in a certain population of known size, etc. The parameters n and θ must be given.

The mean and variance of the Binomial Distribution

$$\mathbb{E}X = n\theta, \quad \text{and} \quad \mathbb{V}ar(X) = n\theta(1-\theta).$$

6.1.3.1 Implementation in R

```
X = rbinom(10000, 4, prob = 0.3)
mean(X == 1)
```

```
[1] 0.4116
```

```r
mean(X > 1)
```

```
[1] 0.3571
```

6.1.4 Geometric Distribution

If we are interested in the number of trials until the first success, then this is modeled by the geometric distribution. A random variable X follows a geometric distribution if and only if its probability distribution is given by:

$$g(x;\theta) = \theta(1-\theta)^{x-1}, \quad x = 1, 2, 3, ...$$

```r
# Function to generate n samples from Geometric(p)
geom.gener <- function(n, p) {
  tmp <- NULL
  for (i in 1:n) {
    u <- runif(1)  # Uniform(0,1)
    x <- 1
    p.x <- p
    sum <- p.x

    while (sum < u) {
      x <- x + 1
      p.x <- p.x * (1 - p)
      sum <- sum + p.x
    }

    tmp <- c(tmp, x)
  }
  return(tmp)
}

# Example: generate 100,000 samples with p = 0.75
x <- geom.gener(100000, 0.75)

# Relative frequencies
table(x) / 100000
```

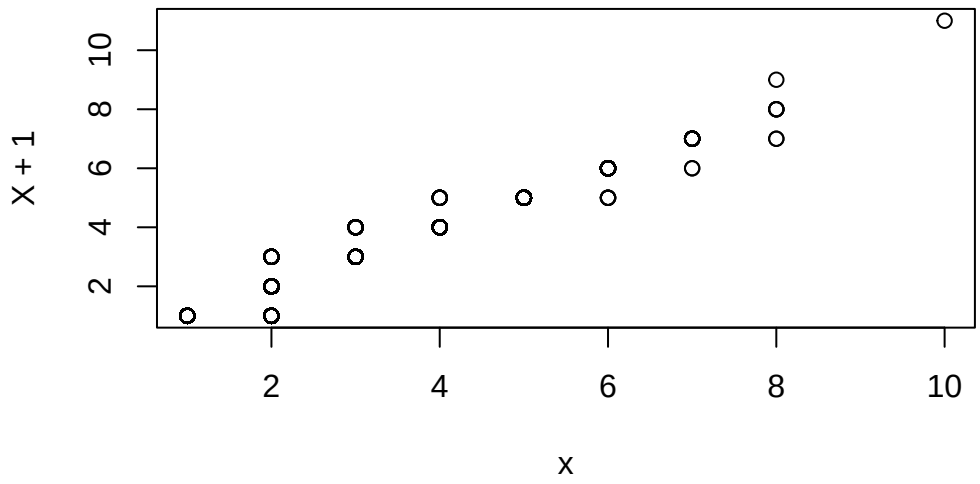
```
x
      1       2       3       4       5       6       7       8      10
0.74867 0.18993 0.04633 0.01120 0.00281 0.00077 0.00021 0.00007 0.00001
```

```r
# OR
X = rgeom(100000, prob= 0.75)
table(X)/100000
```

```
X
      0       1       2       3       4       5       6       7       8      10
0.75057 0.18750 0.04654 0.01141 0.00297 0.00074 0.00021 0.00004 0.00001 0.00001
```

Do you notice the difference between the two functions? R starts the geometric distribution at $x = 0$ and not $x = 1$. We can check that both of these simulate the same distribution by checking a qqplot.

```r
# Correlation between points in QQ plot
cor(qqplot(x, X + 1)$x, qqplot(x, X + 1)$y)
```




```
[1] 0.9966456
```

6.1.5 Hypergeometric Distribution

The motivating question: what happens in a Binomial setting when our trials are NOT independent? In other words, what happens when we sample without replacement?

Consider a set of N elements, of which M are successes. We are interested in obtaining X successes in n trials. This situation is modeled by the Hypergeometric Distribution, which has pdf:

$$h(x, n, M, N) = \frac{\binom{M}{x}\binom{N-M}{n-x}}{\binom{N}{n}}, \quad x = 0, 1, 2, \ldots, n$$

The mean and the variance are

$$\mathbb{E}X = \frac{nM}{N}, \qquad \mathbb{V}ar(X) = \frac{nM(N-M)(N-n)}{N^2(N-1)}.$$

6.1.5.1 Implementation in R

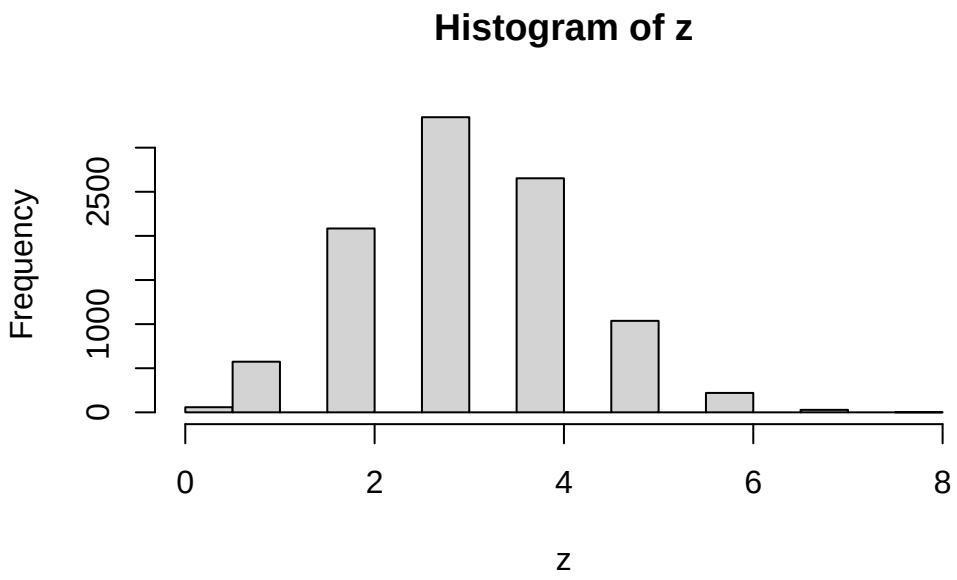
```
# Define population: 12 ones and 13 zeros
x <- c(1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)

# Take one sample of size 8 without replacement
s1 <- sample(x, 8, replace = FALSE)
s1
```

```
[1] 0 1 0 0 0 0 1 1
```

```
# Count how many ones in that sample
sum(s1 == 1)
```

```
[1] 3
```

```
# Repeat the sampling 10,000 times
y <- replicate(10000, sample(x, 8, replace = FALSE))

# Column sums = number of ones in each sample
z <- colSums(y)

# Histogram of simulated distribution
hist(z)
```




```
# Probability of getting exactly 5 ones
mean(z == 5)
```

```
[1] 0.1037
```

```
# Sample mean and variance of distribution
mean(z)
```

```
[1] 3.2098
```

```
var(z)
```

```
[1] 1.371721
```

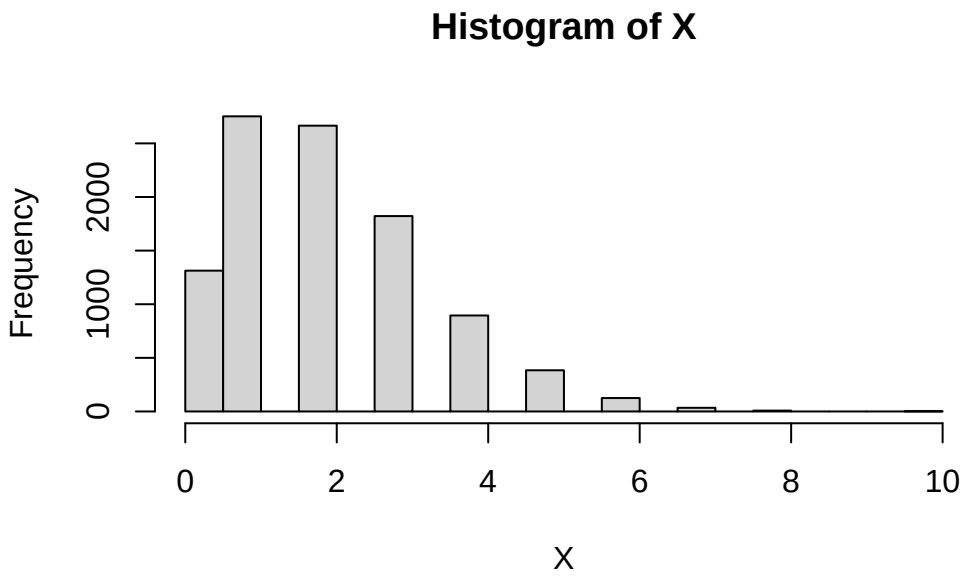
6.1.6 Poisson Distribution

Calculating Binomial probabilities when n is large can be highly tedious and time-consuming. As n approaches infinity and the probability of success approaches 0, where $n\theta$ remains fixed. We can define $n\theta = \lambda$, and obtain the distribution called Poisson.

A random variable X has the Poisson Distribution if and only if its probability distribution is given by:

$$P(x;\lambda) = \frac{\lambda^x e^{-\lambda}}{x!}, \quad x = 0, 1, 2, ...$$

```
# Generate 10,000 samples from Poisson(  = 2)
X <- rpois(10000, 2)

# Histogram of simulated data
hist(X)
```




```
# Probability that X > 3 (estimated by simulation)
mean(X > 3)
```

```
[1] 0.1448
```

```
# Probability that X = 0 (estimated by simulation)
mean(X == 0)
```

```
[1] 0.1313
```

```r
# Theoretical probability that X = 0
exp(-2)
```

```
[1] 0.1353353
```

```r
# Sample mean and variance of simulated distribution
mean(X)
```

```
[1] 2.012
```

```r
var(X)
```

```
[1] 2.012857
```

The mean and variance of the Poisson distribution are

$$\mathbb{E}X = \lambda, \qquad \mathbb{V}ar(X) = \lambda$$

The Poisson distribution is derived as the limiting case of the Binomial (with the above mentioned restrictions) BUT there are many more applications. It models: 1. Number of successes to occur in a given time period 2. Number of telephone calls received in a given time 3. Number of misprints on a page 4. Number of customers entering a bank during various intervals of time

We are now moving onto the continuous distributions that play an important role in Statistical Theory.

6.2 Continuous Distributions

6.2.1 Uniform Distribution

Similar to the discrete uniform distribution except all values within an interval have equal probability. The parameters of the Uniform Density are and (<). The random variable X has the Uniform Distribution if it is continuous and its probability density function is given by

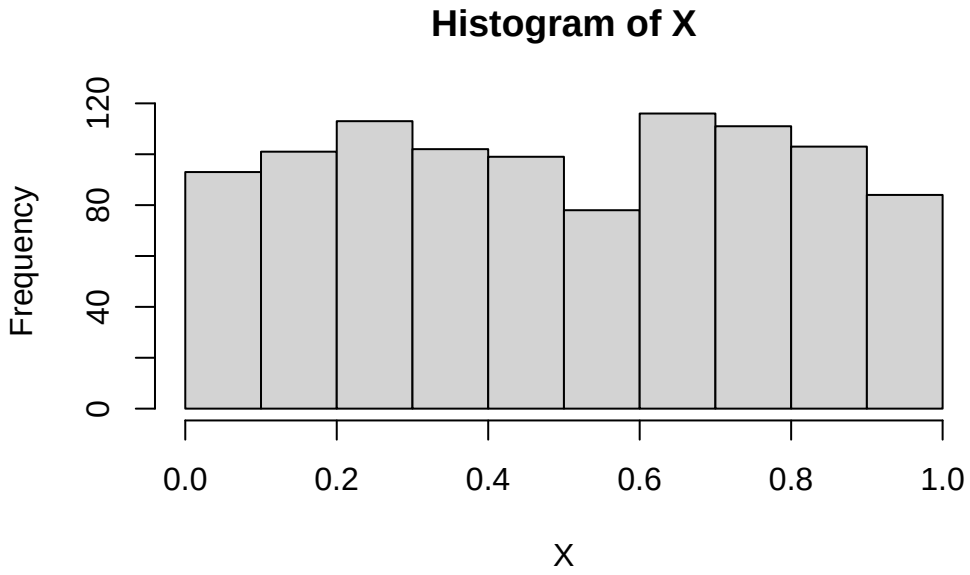
$$f_X(x) = \begin{cases} \dfrac{1}{\beta - \alpha}, & \alpha < x < \beta, \\ 0, & \text{otherwise.} \end{cases}$$

The mean and variance of the Uniform Distribution are

$$\mathbb{E}X = \frac{\alpha + \beta}{2}, \qquad \mathbb{V}ar(X) = \frac{(\beta - \alpha)^2}{12}.$$

6.2.1.1 Implementation in R

```
X <- runif(1000)

# Histogram of simulated data
hist(X)
```




```
mean(X);  var(X)
```

```
[1] 0.497584
```

```
[1] 0.08012191
```

6.2.2 Gamma Distribution

The Gamma function is defined as

$$\Gamma(\alpha) = \int_0^\infty y^{\alpha-1} e^{-y}\, dy.$$

For $\alpha > 0$,

$$\Gamma(\alpha) = (\alpha - 1)\Gamma(\alpha - 1).$$

For any positive integer $\alpha > 0$,

$$\Gamma(\alpha) = (\alpha - 1)!$$

A continuous random variable follows a Gamma Distribution if and only if its probability density function is of the form

$$f_X(x) = \begin{cases} \dfrac{1}{\beta^\alpha \Gamma(\alpha)} x^{\alpha-1} e^{-x/\beta}, & x > 0, \\ 0, & \text{otherwise}, \end{cases}$$

where α, β are positive. The parameters α and β determine the shape of the distribution. α is called the shape parameter and β is called the scale parameter.

Theorem: The mean and variance of the Gamma Distribution are

$$\mathbb{E}X = \alpha\beta, \qquad \mathbb{V}ar(X) = \alpha\beta^2$$

6.2.2.1 Implementation in R

```
rgamma(n = 10, shape = 10, scale = 3)
```

6.2.3 Exponential Distribution

A special case of the Gamma Distribution arises when $\alpha = 1$. To differentiate it from the Gamma distribution, we also let $\beta = \theta$. A continuous random variable follows an Exponential Distribution if and only if its probability density function is of the form:

$$f_X(x) = \begin{cases} \dfrac{1}{\theta} e^{-x/\theta}, & x > 0, \\ 0, & \text{otherwise}. \end{cases}$$

Exponential Distributions have many applications. One of them is a waiting time until the first success of a Poisson process. In this situation, it is often better to model the phenomenon in terms of a rate parameter (i.e. 4 calls per week). Thus, the distribution of waiting times becomes:

$$f_Y(y) = \begin{cases} \lambda e^{-\lambda y}, & y > 0, \\ 0, & \text{otherwise.} \end{cases}$$

6.2.3.1 Memoryless Property

This is also called the *Markov Property.* The distribution of waiting time does not depend on how long you have already waited. In other words:

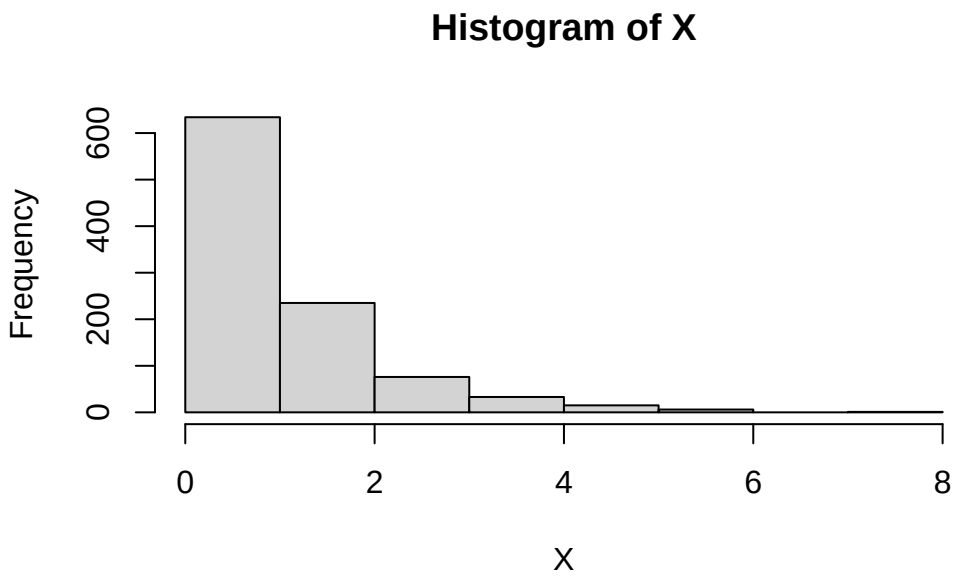
$$P(X > s + t \mid X > t) = P(X > s).$$

The mean and variance of the exponential distribution are

$$\mathbb{E}X = \theta, \qquad \mathbb{V}ar(X) = \theta^2.$$

6.2.3.2 Implementation in R

```
# Generate 1000 samples from Exponential(  = 1)
X <- rexp(1000)
hist(X)
```




```
mean(X); var(X)
```

```
[1] 0.9933691
```

```
[1] 1.009712
```

6.2.4 Chi-Square Distribution

There is another special form of the gamma distribution when $\alpha = \nu/2$ and $\beta = 2.\nu$ is pronounced "nu" and is called the degrees of freedom.

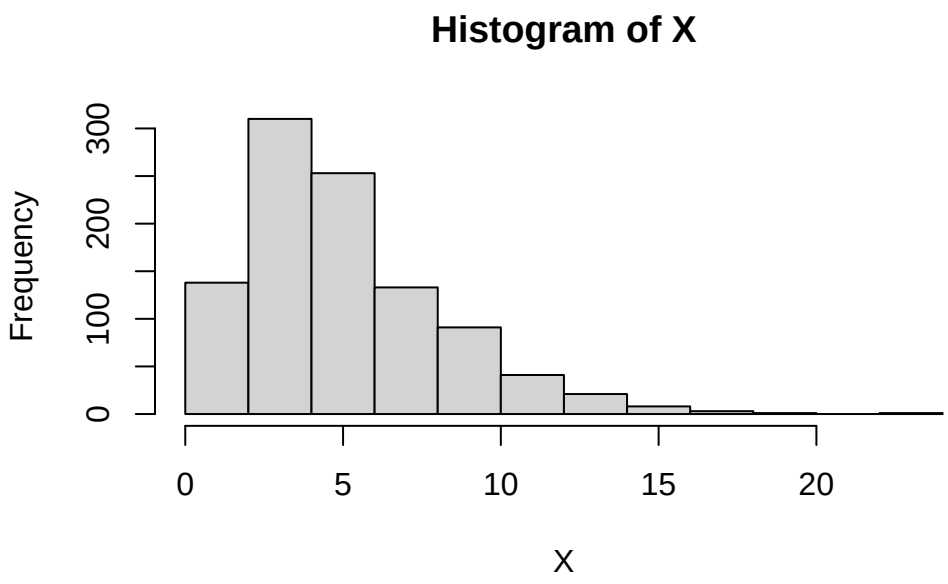
A continuous random variable follows a Chi-Square Distribution if and only if its probability density is given by

$$f_X(x) = \begin{cases} \frac{1}{2^{\nu/2}\Gamma(\frac{\nu}{2})} x^{\frac{\nu}{2}-1} e^{-x/2}, & x > 0, \\ 0, & \text{otherwise.} \end{cases}$$

The mean and variance of the Chi-Square Distribution are

$$\mathbb{E}X = \nu, \qquad \mathbb{V}ar(X) = 2\nu.$$

```r
X <- rchisq(1000, df = 5)

hist(X)
```




```r
mean(X);  var(X)
```

```
[1] 5.031677
```

```
[1] 9.578021
```

6.2.5 Beta Distribution

This is **Not** a special case of the Gamma distribution

The Beta function is defined as

$$B(\alpha, \beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha + \beta)} = \int_0^1 x^{\alpha-1}(1-x)^{\beta-1}\, dx.$$

Like any probability density, the area underneath it must be equal to 1. So, we can rearrange the above definition and write:

$$1 = \int_0^1 \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}\, dx.$$

A continuous random variable follows a Beta Distribution if and only if its probability density function is given by

$$f_X(x) = \begin{cases} \dfrac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}, & 0 < x < 1, \\ 0, & \text{otherwise}, \end{cases}$$

where $\alpha > 0$, $\beta > 0$.

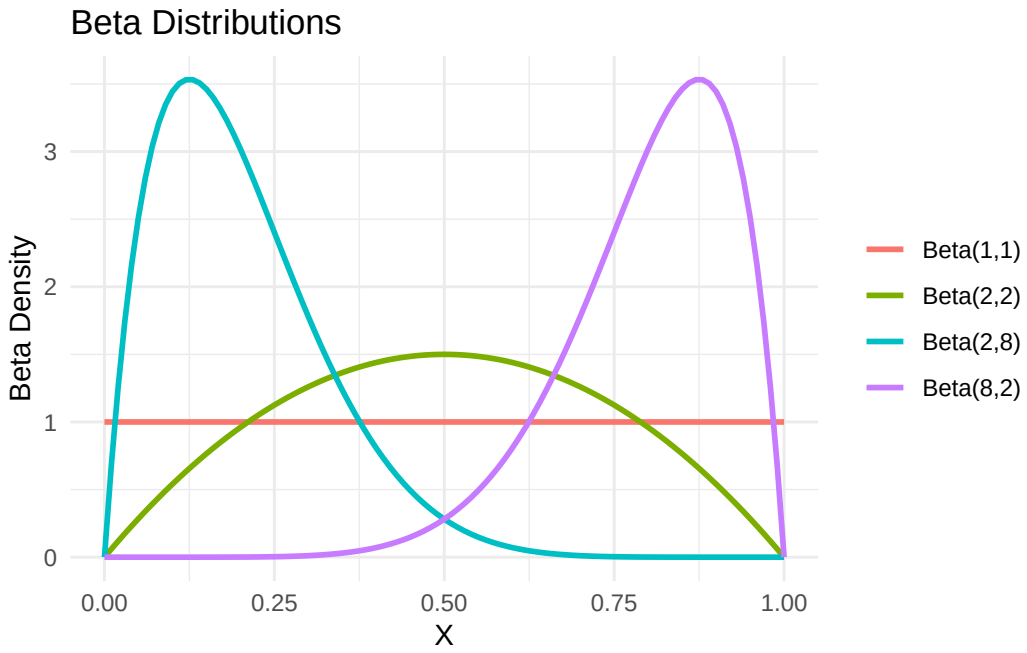
The mean and variance of the Beta Distribution are

$$\mathbb{E}X = \frac{\alpha}{\alpha+\beta}, \qquad \mathbb{V}ar(X) = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}.$$

6.2.5.1 Implementation in R

```r
library(ggplot2)

# Sequence of x values
t <- seq(0, 1, by = 0.01)

# Create data frame with multiple Beta densities
df <- data.frame(
  x = rep(t, 4),
  density = c(dbeta(t, 2, 2),
              dbeta(t, 2, 8),
              dbeta(t, 8, 2),
              dbeta(t, 1, 1)),
  dist = factor(rep(c("Beta(2,2)", "Beta(2,8)", "Beta(8,2)", "Beta(1,1)"),
                    each = length(t)))
)

# Plot with ggplot
ggplot(df, aes(x = x, y = density, color = dist)) +
  geom_line(size = 1) +
  labs(x = "X", y = "Beta Density", title = "Beta Distributions") +
  theme_minimal() +
  theme(legend.title = element_blank())
```

```
Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
i Please use `linewidth` instead.
```



Q: What do you noitce about $B(1,1)$? What distribution is this?

The Beta distribution is related to the Binomial distribution when computing maximum likelihood estimators !(We will make use of this property later when we do Bayesian analysis.)

$$\text{Beta}_{pdf}(p,n,k) = (n+1)\binom{n}{k}p^k(1-p)^{n-k},$$

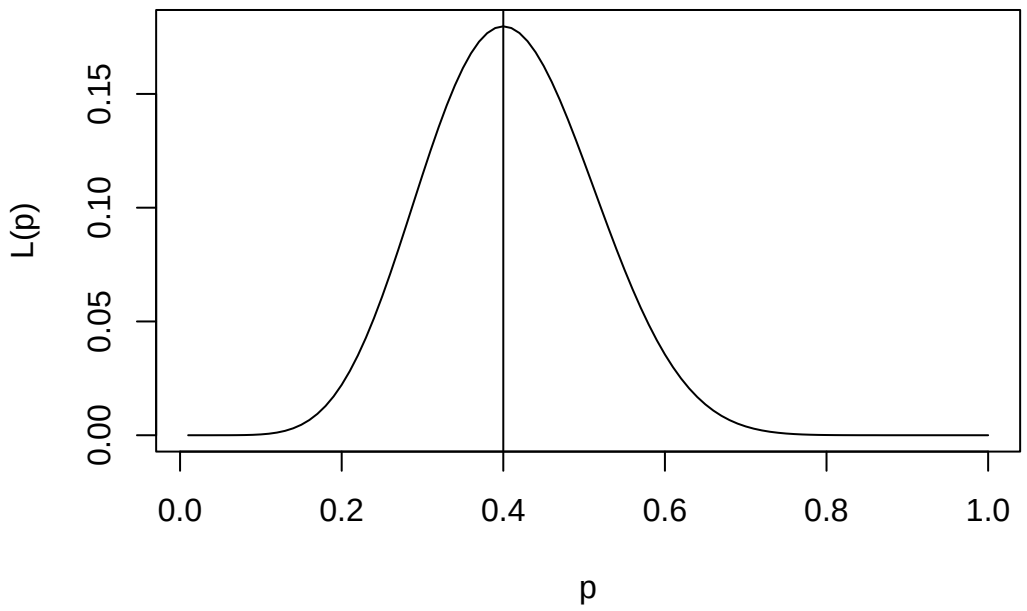
where $p' = \text{Binomial}_{pmf}(k,n,p), \quad k = \text{mode}(\text{Binomial}(n,p))$.

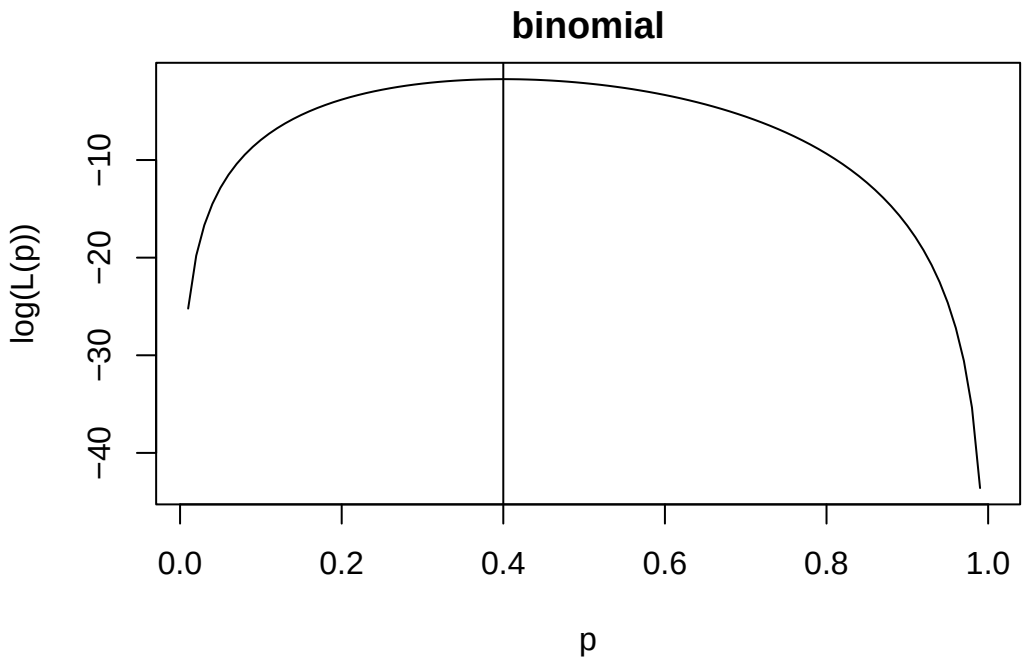
To relate the binomial distribution to the scale and shape parameters of the beta distribution: $\alpha = k+1, \quad \beta = n-\alpha+2$.

6.2.5.2 Implementation in R

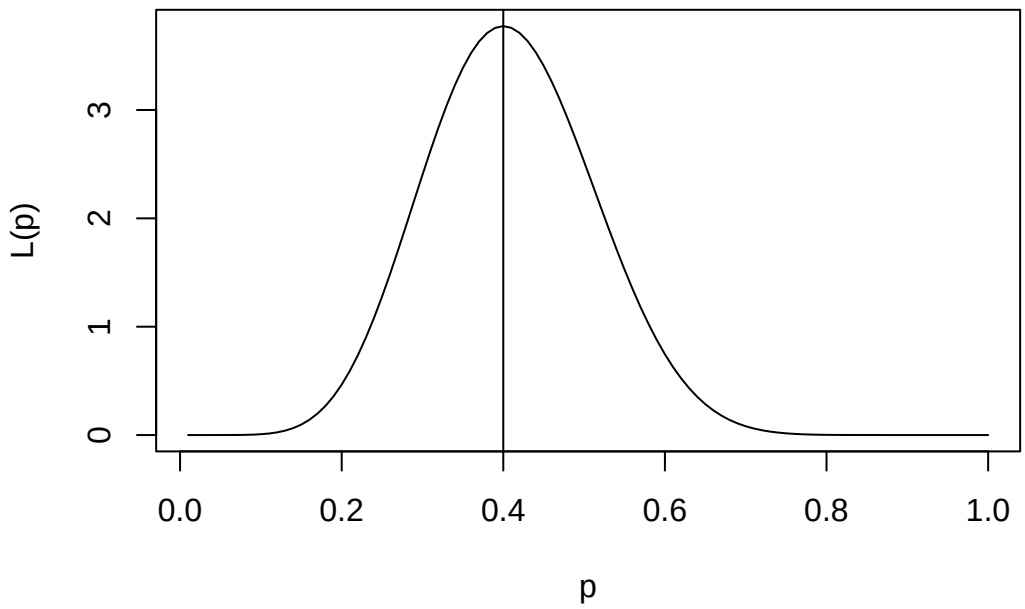
Suppose we flip a coin 20 times and find that we have 8 heads. Thus, our MLE is $8/20 = 0.4$. We can visualize the likelihood function for this scenario:

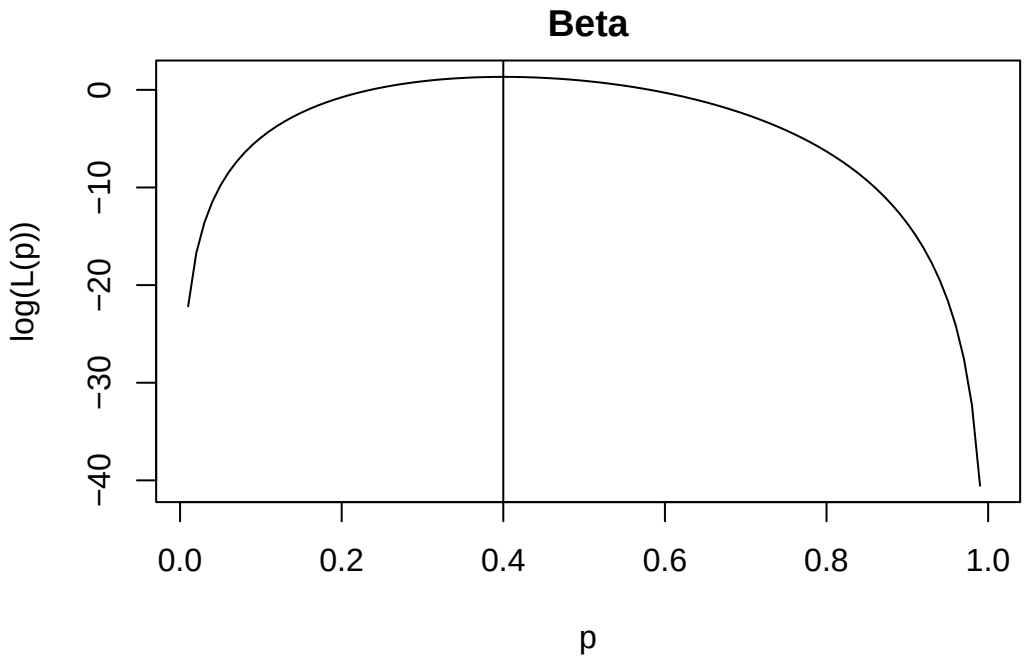
```r
# Likelihood plot using Binomial likelihood
likeli_bino.plot <- function(y, n) {
  L <- function(p) dbinom(y, n, p)
  mle <- optimize(L, interval = c(0, 1), maximum = TRUE)$max

  p <- (1:100) / 100

  # Likelihood
  plot(p, L(p), type = "l")
  abline(v = mle)

  # Log-likelihood
  plot(p, log(L(p)), type = "l", main = "binomial")
  abline(v = mle)
}
par(mfrow = c(1,1), mar = c(4,4,2,1))
likeli_bino.plot(8, 20)
```





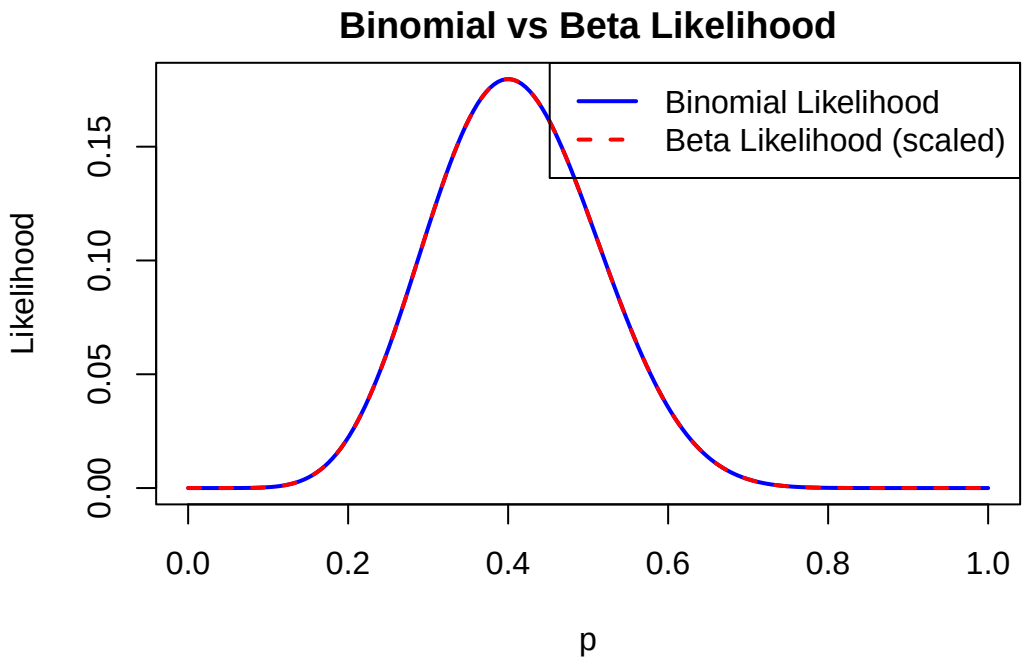

```r
# Likelihood plot using Beta distribution
likeli_beta.plot <- function(y, n) {
  L <- function(p) dbeta(p, y + 1, n - (y + 1) + 2)
  mle <- optimize(L, interval = c(0, 1), maximum = TRUE)$max

  p <- (1:100) / 100

  # Likelihood
  plot(p, L(p), type = "l")
  abline(v = mle)

  # Log-likelihood
  plot(p, log(L(p)), type = "l", main = "Beta")
  abline(v = mle)

  mle
}
likeli_beta.plot(8, 20)
```


L(p)
0
1
2
3
0.0
0.2
0.4
0.6
0.8
1.0
p


Beta
log(L(p))
0
−10
−20
−30
−40
0.0
0.2
0.4
0.6
0.8
1.0
p

```
[1] 0.3999996
```

```r
overlay.likeli <- function(y, n) {
  # Binomial likelihood
  L_binom <- function(p) dbinom(y, n, p)

  # Beta likelihood (with   = y+1,   = n-y+1)
  L_beta <- function(p) dbeta(p, y + 1, n - y + 1)

  # Sequence of p values
  p <- seq(0, 1, length.out = 200)

  # Scale the beta likelihood so it's comparable
  scale_factor <- max(L_binom(p)) / max(L_beta(p))
  par(mfrow=c(1,1))
  # Plot Binomial likelihood
  plot(p, L_binom(p), type = "l", col = "blue", lwd = 2,
       ylab = "Likelihood", xlab = "p",
       main = "Binomial vs Beta Likelihood")

  # Add Beta likelihood (scaled for comparison)
  lines(p, L_beta(p) * scale_factor, col = "red", lwd = 2, lty = 2)

  # Add legend
  legend("topright",
         legend = c("Binomial Likelihood", "Beta Likelihood (scaled)"),
         col = c("blue", "red"),
         lty = c(1, 2), lwd = 2)
}

# Example: 8 successes out of 20 trials

overlay.likeli(8, 20)
```



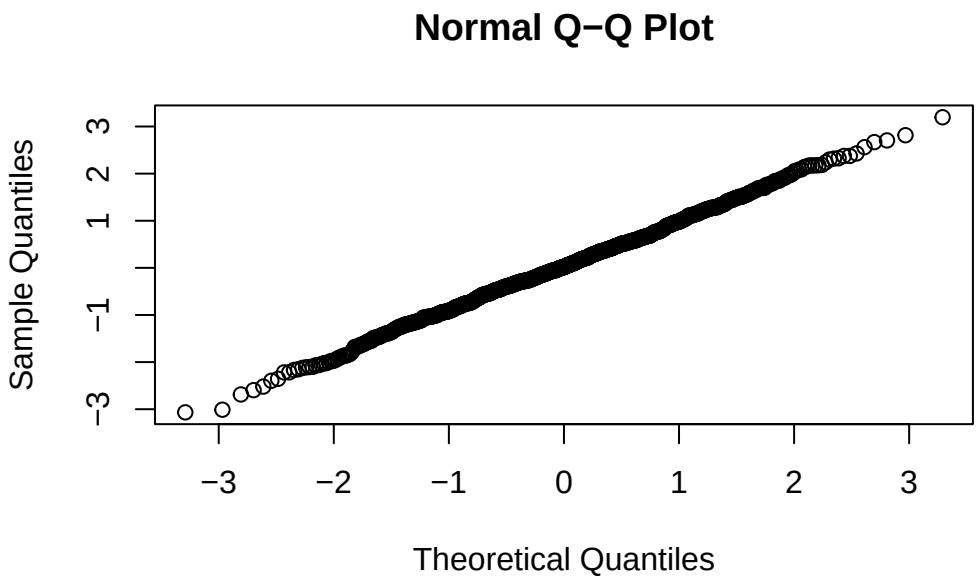
6.2.6 Gaussian Distribution

This is probably the most famous statistical distribution. It is defined by its mean (μ) and variance (σ^2). It is also known as the Normal Distribution. A continuous random variable follows a Normal Distribution if and only if its probability density function is given by

$$f_X(x) = \begin{cases} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), & -\infty < x < \infty, \\ 0, & \text{otherwise.} \end{cases}$$

Note: The Standard Normal Distribution has density with $\mathbb{E}X = 0$ and $\mathbb{V}ar(X) = 1$.

```
X <- rnorm(1000, 0, 1)
qqnorm(X)
```




```
# Approximate probabilities of being within 1, 2, 3 standard deviations
mean(-1 < X & X < 1)  # ~ 68%
```

```
[1] 0.711
```

```
mean(-2 < X & X < 2)  # ~ 95%
```

```
[1] 0.956
```

```
mean(-3 < X & X < 3)  # ~ 99.7%
```

```
[1] 0.997
```

6.2.7 T-Distribution

Another special distribution in statistical inference is the *Student's t-Distribution*.

$$f_X(x) = \begin{cases} \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\,\Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}, & -\infty < x < \infty, \\ 0, & \text{otherwise}, \end{cases}$$

where ν is the degrees of freedom. As $\nu \to \infty$, the pdf converges to the normal distribution.

6.2.8 Implementation in R

```
my_df <- 5
X <- rt(1000, df = my_df)

mean(X); var(X)
```

```
[1] -0.007610344
```

```
[1] 1.711301
```

```
my_df / (my_df - 2)   # Theoretical variance for df > 2
```

```
[1] 1.666667
```

6.2.9 Distribution Function Technique

For continuous random variables, a simple method for finding the probability density of a function of random variables is to find the distribution function and then differentiate to find the pdf.

To find an expression for the distribution function, let $Y = u(x_1, x_2, \ldots, x_n)$, where u is a function. Then,

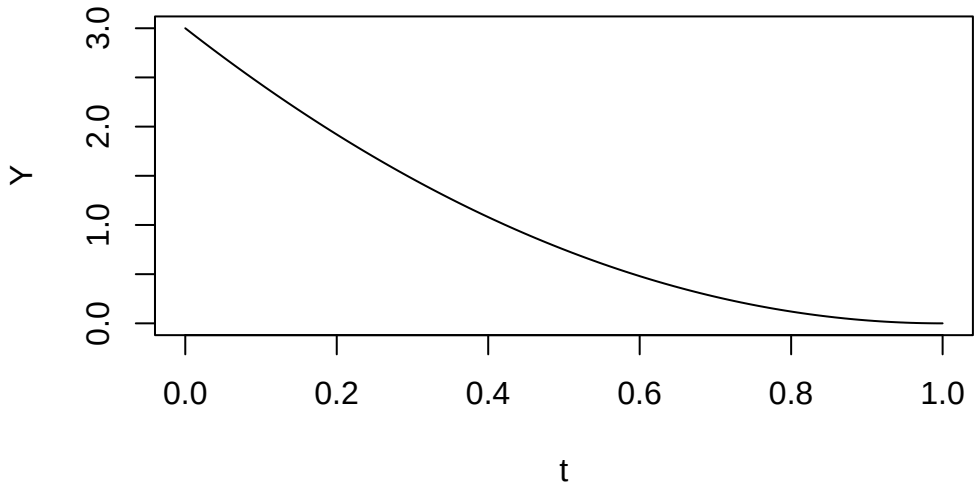
$$F(Y) = P(Y \leq y) = P(u(x_1, x_2, \ldots, x_n) \leq y).$$

Then,

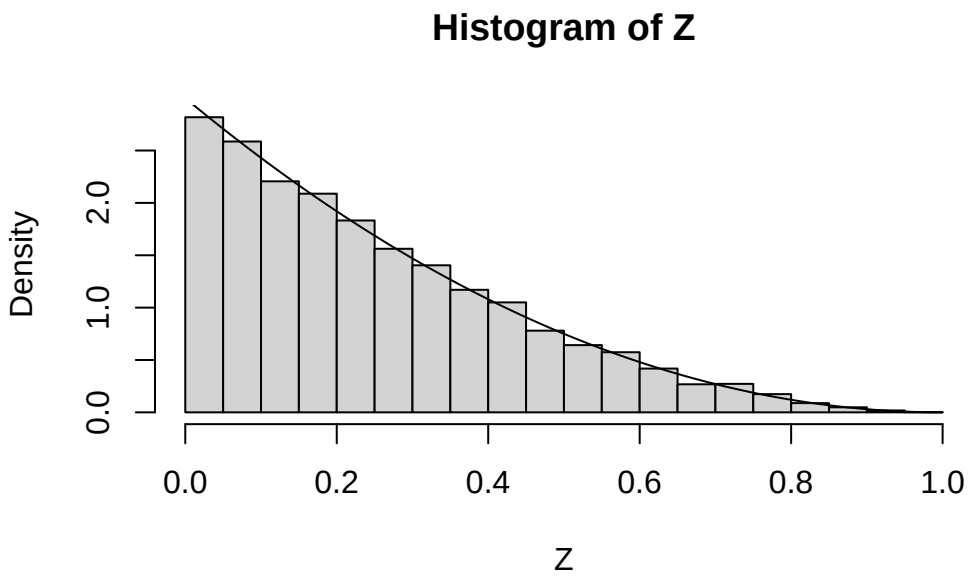
$$f(y) = \frac{dF(Y)}{dy}.$$

Example:

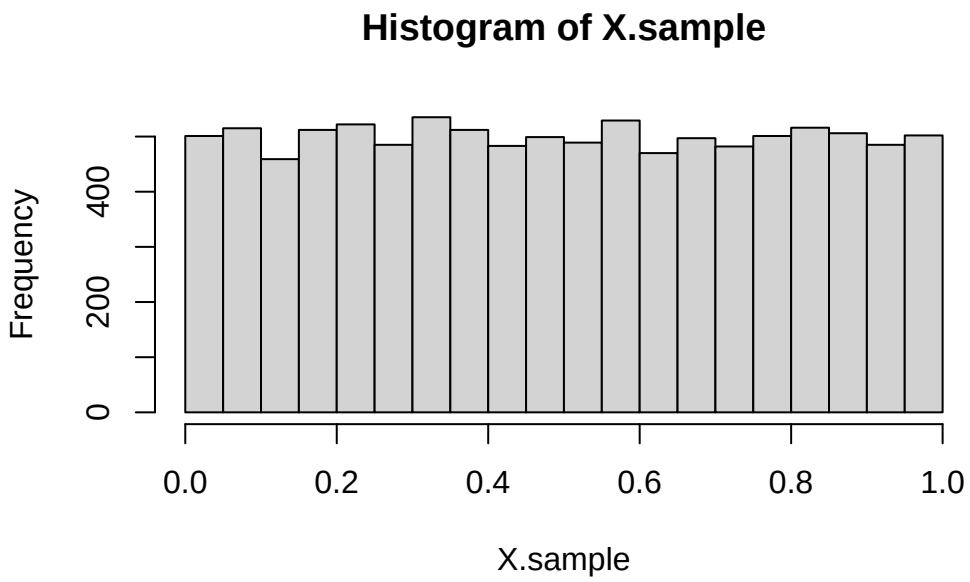
```
# Create sequence from 0 to 1
t <- seq(0, 1, length = 1000)

# Define density function Y = 3 * (1 - t)^2
Y <- 3 * (1 - t)^2

# Plot density
plot(t, Y, type = "l")
```



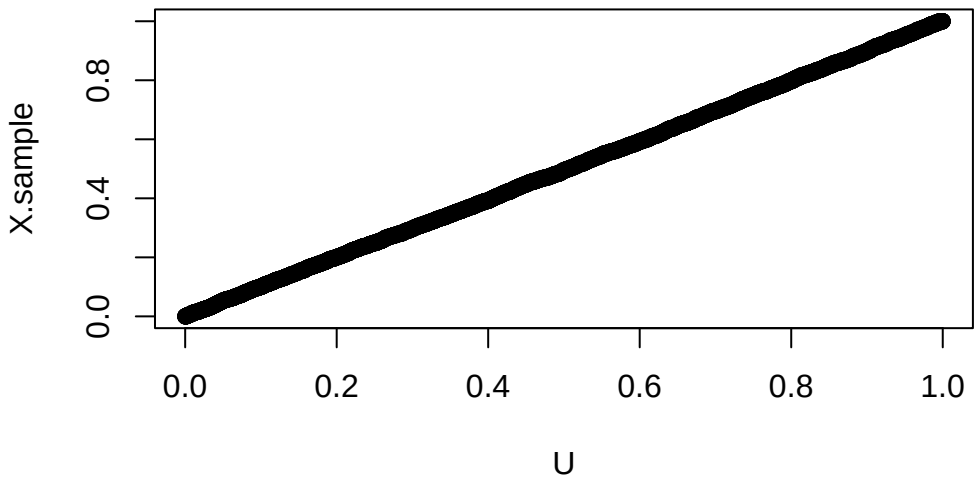

```r
# Sample from t with probability weights Y
Z <- sample(t, 10000, replace = TRUE, prob = Y)

# Histogram of sampled values
hist(Z, freq = FALSE)

# Overlay the density curve
lines(t, Y)
```




```
# Apply transformation: X = (1 - Z)^3
X.sample <- (1 - Z)^3

# Histogram of transformed sample
hist(X.sample)
```




```r
# Compare with uniform distribution
U <- runif(10000)

# QQ-plot to check distributional similarity
qqplot(U, X.sample)
```




```
# Correlation from QQ-plot
cor(qqplot(U, X.sample)$x, qqplot(U, X.sample)$y)
```

```
[1] 0.9999452
```

Special thanks to Dr. Brian Pidgeon who kindly share the notes.