STAT 8670 – Computational Methods in Statistics

2026 Fall Course Notes

Chi-Kuang Yeh

2026-08-29

Table of contents

Preface

Description

Topics 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

9:00 - 12:00 on Wednesday or by appointment

Grade Distribution

- Assignments: 40%
- Exam: 30%
- Project: 30%
- Attendance: *7%

Assignment

☐ Assignment 1: TBA

Midterm

☐ Midterm: TBA

Project

☐ Report: TBA
☐ Presentation: TBA

Topics and Corresponding Lectures

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

Status	Topic	Lecture Covered
	R Programming	1–
	Numerical Approaches and Optimization	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 Introduction to R

Learning objectives

By the end of this chapter, you should be able to:

1. Describe how R reads, evaluates, and prints an expression.
2. Distinguish an object from the name bound to that object.
3. Identify the major data types and data structures in R.
4. Predict how vectorization, coercion, recycling, and missing values affect a computation.
5. Create, inspect, index, and modify vectors, matrices, arrays, lists, and data frames.
6. Write and use simple R functions.
7. Choose appropriately among vectorization, loops, and the `apply()` family.
8. Use the tidyverse pipe and core `dplyr` verbs to express a data-analysis pipeline.
9. Write computations that are reproducible, readable, and numerically responsible.

1.1 Why R?

Modern statistical analyses routinely involve hundreds, thousands, or millions of observations. The calculations are too numerous, and often too complex, to perform reliably by hand. We therefore need software that can manage data, implement statistical methods, automate repeated computations, and document the complete analysis.

R is both a high-level programming language and an environment for statistical computing and graphics. It is especially useful for statistics because it is:

- **free and open source**, so students and researchers can use the same tools;
- **designed for data analysis**, with vectors, missing values, statistical models, and graphics built into the language;
- **extensible**, with a large ecosystem of packages developed by researchers and software developers;
- **reproducible**, because an analysis can be saved as code and rerun from beginning to end;

- **interoperable**, with interfaces to databases and languages such as C++, Python, and Julia.

R is not a menu-driven statistics program. Learning R means learning to express an analysis as a sequence of precise and reproducible computations.

> **i** R, RStudio, and Quarto are different tools
>
> **R** performs the computation. **RStudio** is an integrated development environment that helps us write and run code. **Quarto** combines text, code, results, figures, equations, and references into reproducible HTML, PDF, Word, or presentation output.

1.2 The working environment

There are different integrated development environment to use R, some of them are Rstudio, VS Code and Positron.

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.

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.

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.

1.2.1 RStudio

RStudio is an integrated development environment (IDE) for R, Python, and reproducible publishing. Its default layout contains four panes:

Pane	Main purpose
Source	Write and edit scripts and Quarto documents.
Console	Send expressions directly to R and view immediate results.
Environment/History	Inspect objects and review previous commands.
Files/Plots/Packages/Help/Viewer	Navigate files, inspect graphics, manage packages, read documentation, and preview output.

Use the console for short experiments. Any computation that matters should be placed in an R script or Quarto document so that it can be reproduced.

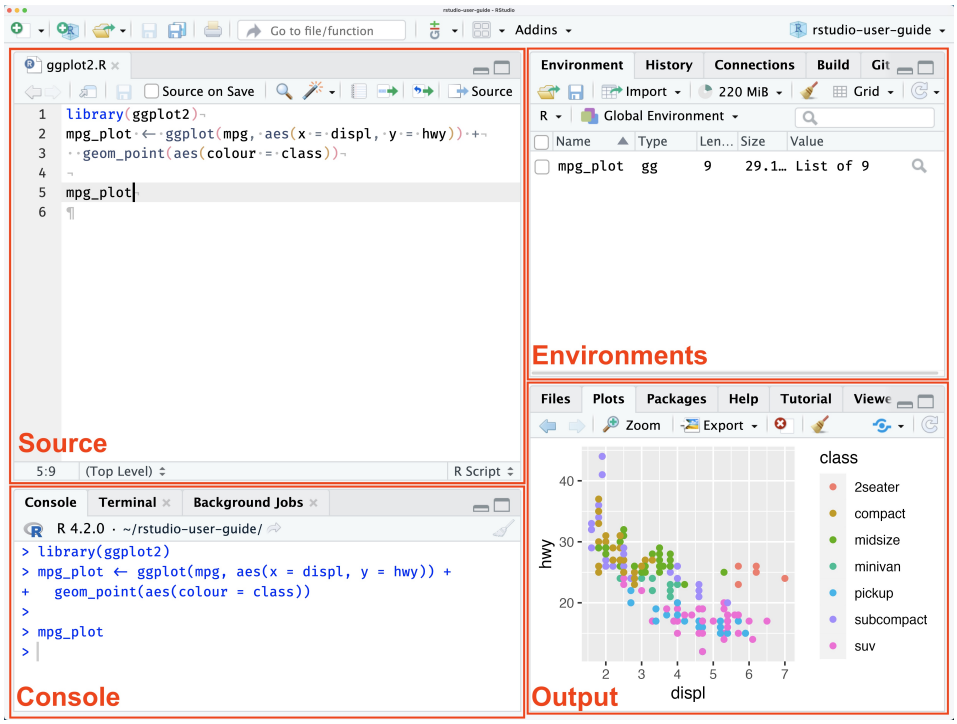


Figure 1.1: RStudio layout borrow from the Posit website.

1.2.2 R scripts and Quarto documents

An R script is a plain-text file with the extension `.R`. It contains R expressions that can be executed one line, one selection, or one file at a time.

A Quarto document has the extension `.qmd`. It combines:

- a YAML header describing the document;
- prose written in Markdown;
- executable code chunks;
- tables, equations, figures, citations, and cross-references.

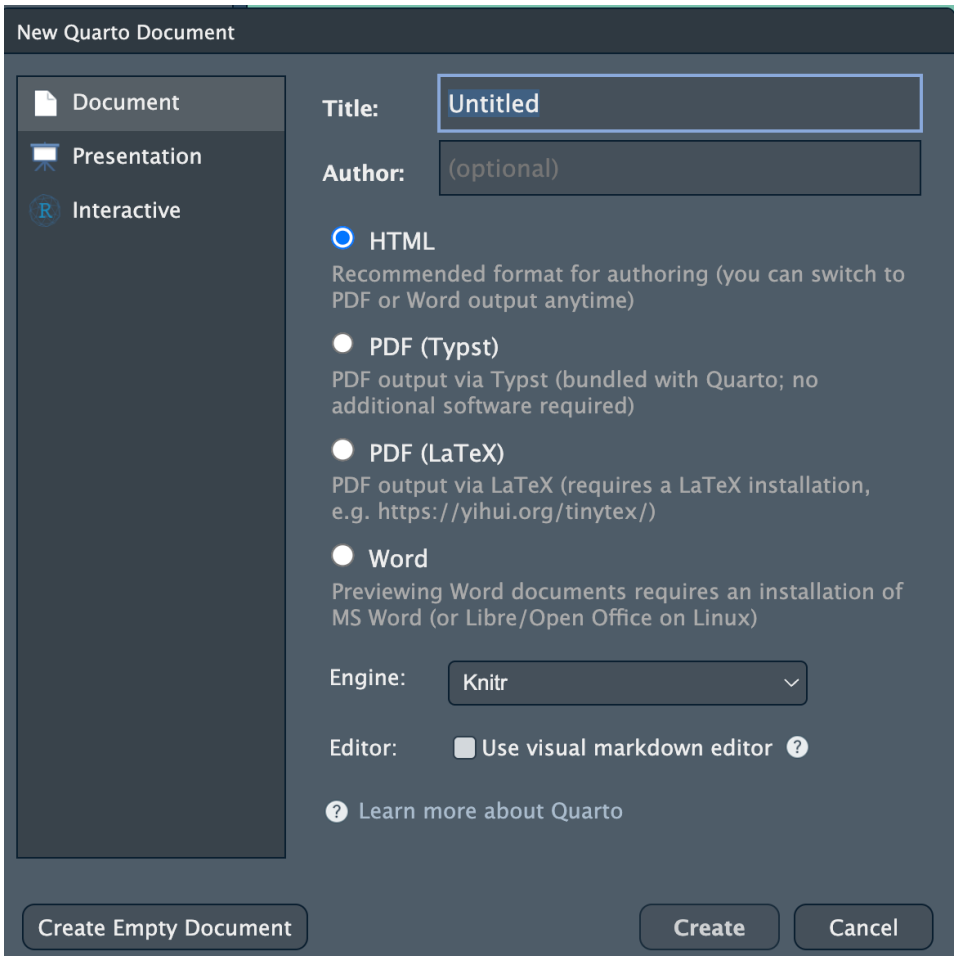


Figure 1.2: Quarto.

In Figure 1.3, the red box is the yaml, the yellow box is the executable code chunk and the green box is a figure.

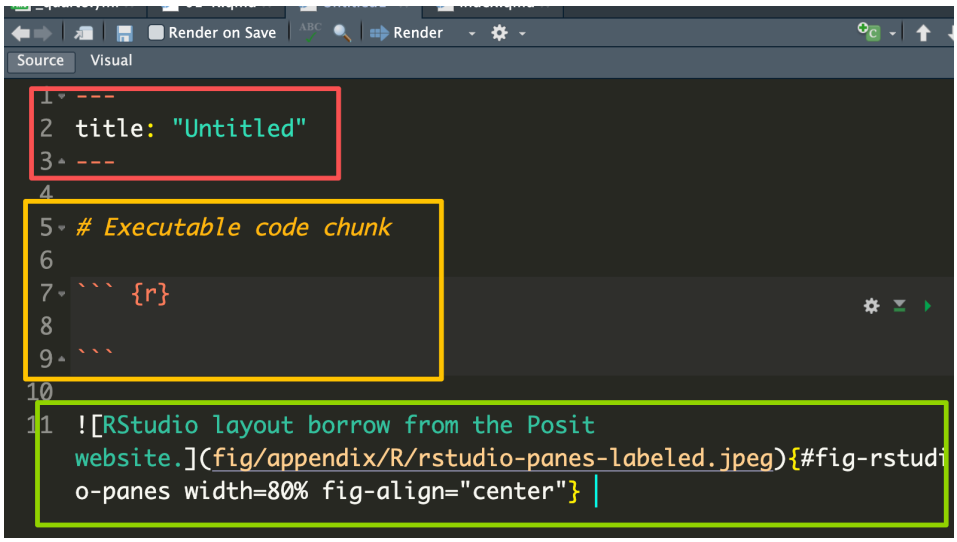


Figure 1.3: Quarto 2.

For example, an executable R chunk is written as

````
```{r}
x <- rnorm(100)
mean(x)
```
````

```
[1] 0.1211347
```

Rendering the document executes the enabled chunks in order and places their results into the final output.

1.2.3 A project-oriented workflow

Create one RStudio Project for the course or for each substantial analysis. Use project-relative paths such as `data/survey.csv` rather than hard-coded paths tied to one computer.

A simple project might contain:

```
stat8670-project/
  data/          # original and processed data
  R/             # reusable functions
  figures/       # saved figures
```

```
  notes/         # Quarto lecture or analysis files
  stat8670-project.Rproj
```

Avoid using `setwd()` inside a reproducible document. When a project is opened, its root provides a stable reference point for relative paths.

1.3 R as a computing environment

At the console, R follows a **read, evaluate, print loop** (REPL):

1. **Read:** parse the text entered by the user.
2. **Evaluate:** compute the value of the expression.
3. **Print:** display the result when appropriate.
4. **Loop:** wait for the next expression.

```
2 + 3 * 4
```

```
[1] 14
```

R respects operator precedence, so multiplication is performed before addition.

```
(2 + 3) * 4
```

```
[1] 20
```

Assignment usually returns its value invisibly:

```
x <- 2 + 3 * 4
x
```

```
[1] 14
```

> A useful mental model
>
> Most R code can be understood as **functions operating on objects**. Even operators such as `+` are functions.

```r
`+`(2, 3)
```

```
[1] 5
```

1.4 Objects, data, and functions

Two fundamental kinds of objects that we will work with are:

1. **Data objects**
2. **Functions**

Data objects include numbers, vectors, matrices, arrays, lists, data frames, and fitted models.

A **function** is a set of instructions that receives input, performs a computation, and returns an output. Functions can be built into R, provided by a package, or written by the user.

```r
mean(c(1, 2, 3, 4))
```

```
[1] 2.5
```

Here, `c(1, 2, 3, 4)` is a data object and `mean()` is a function.

1.5 Expressions, objects, and names

R computes with **objects**. A **name** is a symbol bound to an object.

```r
sample_size <- 100
another_name <- sample_size

sample_size
```

```
[1] 100
```

```r
another_name
```

```
[1] 100
```

The assignment operator `<-` creates or changes a binding.

Names are case-sensitive:

```
n <- 10
N <- 25

c(n = n, N = N)
```

```
 n  N
10 25
```

A useful naming convention is to separate words using `_`:

```
sample_mean <- 10
number_students <- 30
```

Use `ls()` to list names in the current environment and `rm()` to remove a binding.

```
temporary_result <- 42
"temporary_result" %in% ls()
```

```
[1] TRUE
```

```
rm(temporary_result)

"temporary_result" %in% ls()
```

```
[1] FALSE
```

1.6 Data types

The fundamental data structure in R is the **vector**. An atomic vector contains elements of one common type.

Common atomic types include:

Type	Example	Typical use
logical	`TRUE`, `FALSE`	conditions and indicators
integer	`2L`	counts and indices
double	`2`, `3.14`	numerical computation
character	`"R"`	text and labels
complex	`1 + 2i`	complex arithmetic
raw	`charToRaw("R")`	bytes

```
logical_vector <- c(TRUE, FALSE, TRUE)
integer_vector <- c(1L, 2L, 3L)
double_vector <- c(1, 2, 3)
character_vector <- c("one", "two", "three")

typeof(logical_vector)
```

```
[1] "logical"
```

```
typeof(integer_vector)
```

```
[1] "integer"
```

```
typeof(double_vector)
```

```
[1] "double"
```

```
typeof(character_vector)
```

```
[1] "character"
```

1.6.1 Changing data types

Data types can be explicitly converted, but conversion may result in information loss.

```
x <- pi
x
```

```
[1] 3.141593
```

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

```
[1] 3
```

Here, converting π to an integer removes its decimal component.

Common conversion functions include:

- `as.integer()`
- `as.numeric()`
- `as.character()`
- `as.logical()`
- `as.Date()`
- `as.factor()`
- `as.list()`
- `as.matrix()`
- `as.data.frame()`
- `as.vector()`
- `as.complex()`

> ⚠ Type conversion can lose information
>
> Always inspect the result after converting an object from one type to another.

1.7 Operators

A **unary operator** operates on one object:

```r
-x
!x
```

A **binary operator** operates on two objects:

```r
x + y
x - y
x * y
x / y
```

Comparison operators

Comparison operators return logical values:

```
x == y
x != y
x < y
x > y
x <= y
x >= y
```

Logical operators

Logical operators combine logical values:

```
x & y
x | y
!x
```

The operators `&` and `|` operate element by element.

```
x <- c(TRUE, FALSE, FALSE)
y <- c(TRUE, TRUE, FALSE)

x | y
```

```
[1]  TRUE  TRUE FALSE
```

```
x & y
```

```
[1]  TRUE FALSE FALSE
```

By contrast, `&&` and `||` are intended for a single logical condition and are commonly used inside `if` statements.

1.8 Vectorization

Many R functions and operators act on entire vectors.

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

x^2
```

```
[1]  1  4  9 16
```

```r
sqrt(x)
```

```
[1] 1.000000 1.414214 1.732051 2.000000
```

```r
sum(x)
```

```
[1] 10
```

```r
mean(x)
```

```
[1] 2.5
```

Vectorized code often expresses statistical calculations more directly than element-by-element code.

1.9 Recycling

Arithmetic between vectors is usually element-wise.

```r
x <- c(10, 20, 30)
y <- c(1, 2, 3)

x + y
```

```
[1] 11 22 33
```

If one vector is shorter, R may **recycle** its values.

```r
c(10, 20, 30, 40) + c(1, 2)
```

```
[1] 11 22 31 42
```

When the shorter length does not divide the longer length exactly, R produces a warning.

```
c(10, 20, 30, 40) + c(1, 2, 3)
```

```
Warning in c(10, 20, 30, 40) + c(1, 2, 3): longer object length is not a
multiple of shorter object length
```

```
[1] 11 22 33 41
```

1.10 Coercion

An atomic vector must have one type. Combining different types may cause R to convert values to a common type.

```
mixed_numeric <- c(TRUE, 2L, 3.5)
mixed_character <- c(TRUE, 2L, 3.5, "four")

mixed_numeric
```

```
[1] 1.0 2.0 3.5
```

```
typeof(mixed_numeric)
```

```
[1] "double"
```

```
mixed_character
```

```
[1] "TRUE" "2"    "3.5"  "four"
```

```
typeof(mixed_character)
```

```
[1] "character"
```

A simplified coercion hierarchy is:

```
logical → integer → double → complex → character
```

Inspect unfamiliar objects using

```
typeof()
class()
length()
str()
```

1.11 Missing and special values

R uses `NA` to represent a missing value.

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

weather_data <- data.frame(day, weather)
weather_data
```

```
        day weather
1    Monday Raining
2   Tuesday   Sunny
3 Wednesday    <NA>
4  Thursday   Windy
5    Friday Snowing
```

Missing values usually propagate through calculations:

```
scores <- c(88, NA, 91, 84)

mean(scores)
```

```
[1] NA
```

```
mean(scores, na.rm = TRUE)
```

```
[1] 87.66667
```

```r
is.na(scores)
```

```
[1] FALSE  TRUE FALSE FALSE
```

Do not use

```r
x == NA
```

to detect missing values. Use

```r
is.na(x)
```

instead.

Other special numerical values include

```r
c(1 / 0, -1 / 0, 0 / 0)
```

```
[1]  Inf -Inf  NaN
```

which produce `Inf`, `-Inf`, and `NaN`.

1.12 Indexing and subsetting

Indexing allows us to access or modify parts of an object.

> **!** R indexing begins at 1
>
> Unlike languages such as Python and C++, R uses **1-based indexing**. The first element is indexed by `1`, not `0`.

For a vector:

```r
x <- c(first = 10, second = 20, third = 30, fourth = 40)

x[1]
```

```
first
   10
```

```r
x[c(1, 3)]
```

```
first third
   10    30
```

```r
x[-2]
```

```
 first  third fourth
    10     30     40
```

```r
x[x >= 30]
```

```
 third fourth
    30     40
```

```r
x[c("second", "fourth")]
```

```
second fourth
    20     40
```

Positive integers select positions, negative integers exclude positions, logical values filter positions, and character values select named elements.

> **A common surprise**
>
> Prefer `seq_along(x)` to `1:length(x)` when iterating over the positions of an object.

```r
empty <- numeric(0)

1:length(empty)
```

```
[1] 1 0
```

```r
seq_along(empty)
```

```
integer(0)
```

1.13 Naming elements

Names can be attached to vector elements:

```
temp <- c(20, 30, 27, 31, 45)

names(temp) <- c("Mon", "Tues", "Wed", "Thurs", "Fri")

temp
```

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

```
temp["Wed"]
```

```
Wed
 27
```

A vector does not have rows, so this is inappropriate:

```
rownames(temp) <- "Day1"
```

A matrix can have row and column names:

```
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"

temp_mat
```

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

1.14 Arrays and matrices

An **array** is a multidimensional data structure.

```
array_1d <- array(1:10, dim = 10)
array_1d
```

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

```
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
```

```
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
```

A **matrix** is a two-dimensional array:

```r
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
```

```r
is.matrix(array_2d)
```

```
[1] TRUE
```

```r
is.matrix(my_matrix)
```

```
[1] TRUE
```

```r
is.array(array_2d)
```

```
[1] TRUE
```

```r
is.array(my_matrix)
```

```
[1] TRUE
```

A matrix is an atomic vector with a `dim` attribute, so all entries must share one common type.

```r
A <- matrix(1:6, nrow = 2, ncol = 3)

attributes(A)
```

```
$dim
[1] 2 3
```

```r
A[2, 3]
```

```
[1] 6
```

Matrix multiplication uses `%*%`:

```r
B <- matrix(c(1, 0, 0, 1, 1, 1), nrow = 3)

A %*% B
```

```
     [,1] [,2]
[1,]    1    9
[2,]    2   12
```

while `*` performs element-wise multiplication.

1.15 Lists and key-value pairs

A list can contain objects of different types and lengths.

```r
fit_summary <- list(
  method = "least squares",
  coefficients = c(intercept = 1.2, slope = 0.8),
  converged = TRUE
)

str(fit_summary)
```

```
List of 3
 $ method      : chr "least squares"
 $ coefficients: Named num [1:2] 1.2 0.8
  ..- attr(*, "names")= chr [1:2] "intercept" "slope"
 $ converged   : logi TRUE
```

Named list elements naturally form **key-value pairs**.

```r
key1 <- "Tues"
value1 <- 32

key2 <- "Wed"
value2 <- 28

list_temp <- list()

list_temp[[key1]] <- value1
list_temp[[key2]] <- value2

list_temp
```

```
$Tues
[1] 32

$Wed
[1] 28
```

Values can then be obtained from their keys:

```r
list_temp[["Tues"]]
```

```
[1] 32
```

```r
list_temp$Tues
```

```
[1] 32
```

For lists, note the important distinction:

```r
fit_summary["coefficients"]
```

```
$coefficients
intercept     slope
      1.2       0.8
```

```
fit_summary[["coefficients"]]
```

```
intercept     slope
      1.2       0.8
```

`[` returns a sublist, whereas `[[` extracts the element itself.

1.16 Factors

A factor represents categorical data using integer codes and a set of levels.

```
group <- factor(c("control", "treatment", "control"))

group
```

```
[1] control   treatment control
Levels: control treatment
```

```
unclass(group)
```

```
[1] 1 2 1
attr(,"levels")
[1] "control"   "treatment"
```

```
levels(group)
```

```
[1] "control"   "treatment"
```

The order of factor levels matters in statistical models, so define it deliberately when necessary.

1.17 Data frames

A **data frame** is a two-dimensional rectangular data structure.

Conceptually, a data frame is a named list of equal-length vectors:

- each row represents an observational unit;
- each column represents a variable;
- columns may have different data types.

```
students <- data.frame(
  id = 1:6,
  name = c("Alice", "Bob", "Chen", "Divya", "Elena", "Farah"),
  program = c("MS", "PhD", "MS", "PhD", "MS", "PhD"),
  score = c(90, 85, NA, 94, 88, 91),
  completed = c(TRUE, TRUE, FALSE, TRUE, TRUE, TRUE)
)

students
```

```
  id  name program score completed
1  1 Alice      MS    90      TRUE
2  2   Bob     PhD    85      TRUE
3  3  Chen      MS    NA     FALSE
4  4 Divya     PhD    94      TRUE
5  5 Elena      MS    88      TRUE
6  6 Farah     PhD    91      TRUE
```

Another familiar built-in data frame is:

```
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
```

1.17.1 Inspecting a data frame

Always inspect newly imported data before analyzing them.

```
class(students)
```

```
[1] "data.frame"
```

```
dim(students)
```

```
[1] 6 5
```

```
nrow(students)
```

```
[1] 6
```

```
ncol(students)
```

```
[1] 5
```

```
names(students)
```

```
[1] "id"        "name"      "program"   "score"     "completed"
```

```
str(students)
```

```
'data.frame':   6 obs. of  5 variables:
 $ id       : int  1 2 3 4 5 6
 $ name     : chr  "Alice" "Bob" "Chen" "Divya" ...
 $ program  : chr  "MS" "PhD" "MS" "PhD" ...
 $ score    : num  90 85 NA 94 88 91
 $ completed: logi  TRUE TRUE FALSE TRUE TRUE TRUE
```

```
summary(students)
```

```
       id              name        program      score      completed
 Min.   :1.00   Length   :6   Length   :6   Min.   :85.0   Mode :logical
 1st Qu.:2.25   N.unique :6   N.unique :2   1st Qu.:88.0   FALSE:1
 Median :3.50   N.blank  :0   N.blank  :0   Median :90.0   TRUE :5
 Mean   :3.50   Min.nchar:3   Min.nchar:2   Mean   :89.6
 3rd Qu.:4.75   Max.nchar:5   Max.nchar:3   3rd Qu.:91.0
 Max.   :6.00                               Max.   :94.0
                                            NAs    :1
```

```
head(students, 3)
```

```
  id  name program score completed
1  1 Alice      MS    90      TRUE
2  2   Bob     PhD    85      TRUE
3  3  Chen      MS    NA     FALSE
```

`str()` is especially useful because it gives a compact summary of the object's structure and types.

1.17.2 Subsetting a data frame

Use

```
data[rows, columns]
```

for two-dimensional indexing:

```
students[1, ]
```

```
  id  name program score completed
1  1 Alice      MS    90      TRUE
```

```
students[, c("name", "score")]
```

```
   name score
1 Alice    90
2   Bob    85
3  Chen    NA
4 Divya    94
5 Elena    88
6 Farah    91
```

```
students[students$program == "PhD", ]
```

```
  id  name program score completed
2  2   Bob     PhD    85      TRUE
4  4 Divya     PhD    94      TRUE
6  6 Farah     PhD    91      TRUE
```

```
students[1:3, c("name", "score")]
```

```
   name score
1 Alice    90
2   Bob    85
3  Chen    NA
```

```
students[2, "score"]
```

```
[1] 85
```

Three commonly used ways to extract a column are:

```
students["score"]
```

```
  score
1    90
2    85
3    NA
4    94
5    88
6    91
```

```
students[["score"]]
```

```
[1] 90 85 NA 94 88 91
```

```
students$score
```

```
[1] 90 85 NA 94 88 91
```

The first returns a one-column data frame. The latter two return the underlying vector.

1.17.3 Adding and modifying variables

```
students$passed <- students$score >= 70

students$centered_score <-
  students$score - mean(students$score, na.rm = TRUE)

students
```

```
  id  name program score completed passed centered_score
1  1 Alice      MS    90      TRUE   TRUE            0.4
2  2   Bob     PhD    85      TRUE   TRUE           -4.6
3  3  Chen      MS    NA     FALSE     NA            NA
4  4 Divya     PhD    94      TRUE   TRUE            4.4
5  5 Elena      MS    88      TRUE   TRUE           -1.6
6  6 Farah     PhD    91      TRUE   TRUE            1.4
```

Missingness propagates naturally. If a score is unknown, whether the student passed is also unknown.

1.17.4 Importing rectangular data

```
survey <- read.csv(
  "data/survey.csv",
  na.strings = c("", "NA", ".")
)

measurements <- read.table(
  "data/measurements.txt",
  header = TRUE,
  sep = "\t"
)
```

After importing, check:

```
dim(survey)
names(survey)
str(survey)
head(survey)
colSums(is.na(survey))
```

> **Data import is part of the analysis**
>
> Import settings such as delimiters, missing-value codes, decimal marks, encodings, and inferred variable types can change the data that R sees and therefore affect the final analysis.

> **Do not automatically treat a data frame as a matrix**
>
> `apply()` first converts a data frame to a matrix. A mixed-type data frame can therefore be converted entirely to character values. Use `lapply()`, `vapply()`, or `dplyr::across()` for column-wise operations when types differ.

1.18 Functions

A function packages a computation so that it can be reused.

```
standardize <- function(x, remove_na = FALSE) {
  center <- mean(x, na.rm = remove_na)
  spread <- sd(x, na.rm = remove_na)

  (x - center) / spread
}

standardize(c(2, 4, 6, 8))
```

```
[1] -1.1618950 -0.3872983  0.3872983  1.1618950
```

A function has three main components:

- **formals:** the arguments;
- **body:** the expressions evaluated;
- **environment:** where the function was created.

```
formals(standardize)
```

```
$x


$remove_na
[1] FALSE
```

```
body(standardize)
```

```
{
    center <- mean(x, na.rm = remove_na)
    spread <- sd(x, na.rm = remove_na)
    (x - center)/spread
}
```

```
environment(standardize)
```

```
<environment: R_GlobalEnv>
```

When a function is called, R creates a temporary environment, matches the arguments, evaluates the body, and returns the value of the final expression unless `return()` is called earlier.

1.18.1 Argument matching

Arguments can be matched by name or position.

```
round(x = 3.14159, digits = 3)
```

```
[1] 3.142
```

```
round(3.14159, 3)
```

```
[1] 3.142
```

Named arguments are often clearer in code intended to be read by others.

1.18.2 Lazy evaluation

R evaluates function arguments only when they are needed.

```r
choose_one <- function(x, y) {
  x
}

choose_one(
  10,
  stop("This argument is never evaluated")
)
```

```
[1] 10
```

This behavior is called **lazy evaluation**.

1.19 Environments and lexical scoping

An environment stores name-object bindings and has a parent environment.

R uses **lexical scoping**: functions look for nonlocal names in the environment where they were defined.

```r
make_power <- function(power) {
  function(x) {
    x^power
  }
}

square <- make_power(2)
cube <- make_power(3)

square(5)
```

```
[1] 25
```

```r
cube(5)
```

```
[1] 125
```

A function together with its defining environment is called a **closure**.

Local assignment affects the current function environment:

```
x <- 100

change_locally <- function() {
  x <- 5
  x
}

change_locally()
```

```
[1] 5
```

```
x
```

```
[1] 100
```

1.20 Memory and copy-on-modify

R behaves as if objects are copied when modified.

```
x <- c(10, 20, 30)
y <- x

y[1] <- 999

x
```

```
[1] 10 20 30
```

```
y
```

```
[1] 999  20  30
```

Changing `y` does not change `x`.

Repeatedly expanding an object can require many allocations:

```r
result <- numeric(0)

for (i in seq_len(100000)) {
  result <- c(result, i^2)
}
```

Instead, preallocate the required memory:

```r
n <- 100000

result <- numeric(n)

for (i in seq_len(n)) {
  result[i] <- i^2
}

head(result)
```

```
[1]  1  4  9 16 25 36
```

1.21 Iteration: vectorization, loops, and the apply family

An iterative computation can usually be expressed in several ways.

```r
x <- c(2, 4, 6, 8)

# Vectorized
x^2
```

```
[1]  4 16 36 64
```

```r
# Preallocated loop
squares <- numeric(length(x))

for (i in seq_along(x)) {
  squares[i] <- x[i]^2
}

squares
```

```
[1]  4 16 36 64
```

```r
# Apply a function
vapply(
  x,
  function(value) value^2,
  numeric(1)
)
```

```
[1]  4 16 36 64
```

Use vectorization when an existing vectorized function directly expresses the computation. Use a loop when the calculation has evolving state or complex indexing. Use an apply-family function when the same operation is independently applied to several pieces of an object.

> **!** The apply family is not magic vectorization
>
> `apply()` and `lapply()` still perform iteration internally. They are not automatically faster than a well-written preallocated `for` loop. Their main advantage is often clarity and conciseness.

1.21.1 `apply()`

`apply(X, MARGIN, FUN, ...)` applies a function over dimensions of a matrix or array.

- `MARGIN = 1`: rows
- `MARGIN = 2`: columns

```r
X <- matrix(1:12, nrow = 3, byrow = TRUE)

colnames(X) <- c("x1", "x2", "x3", "x4")

X
```

```
     x1 x2 x3 x4
[1,]  1  2  3  4
[2,]  5  6  7  8
[3,]  9 10 11 12
```

```r
apply(X, MARGIN = 1, FUN = mean)
```

```
[1]  2.5  6.5 10.5
```

```r
apply(X, MARGIN = 2, FUN = sd)
```

```
x1 x2 x3 x4
 4  4  4  4
```

Additional arguments are passed through `...`:

```r
X_missing <- X
X_missing[2, 3] <- NA

apply(X_missing, 2, mean, na.rm = TRUE)
```

```
x1 x2 x3 x4
 5  6  7  8
```

For common operations, specialized functions are preferable:

```r
rowMeans(X)
```

```
[1]  2.5  6.5 10.5
```

```r
colMeans(X)
```

```
x1 x2 x3 x4
 5  6  7  8
```

```r
rowSums(X)
```

```
[1] 10 26 42
```

```r
colSums(X)
```

```
x1 x2 x3 x4
15 18 21 24
```

1.21.2 Why `apply()` can be dangerous for data frames

```
mixed_df <- data.frame(
  id = c("A", "B", "C"),
  x = c(10, 20, 30),
  y = c(2, 4, 6)
)

as.matrix(mixed_df)
```

```
     id  x    y
[1,] "A" "10" "2"
[2,] "B" "20" "4"
[3,] "C" "30" "6"
```

```
typeof(as.matrix(mixed_df))
```

```
[1] "character"
```

Because one column is character, the entire matrix becomes character.

A safer approach is:

```
numeric_columns <- vapply(
  mixed_df,
  is.numeric,
  logical(1)
)

vapply(
  mixed_df[numeric_columns],
  mean,
  numeric(1)
)
```

```
 x  y
20  4
```

1.21.3 `lapply()`

`lapply()` applies a function to each element and always returns a list.

```
lapply(students, class)
```

```
$id
[1] "integer"

$name
[1] "character"

$program
[1] "character"

$score
[1] "numeric"

$completed
[1] "logical"

$passed
[1] "logical"

$centered_score
[1] "numeric"
```

```
lapply(
  students[c("score", "centered_score")],
  mean,
  na.rm = TRUE
)
```

```
$score
[1] 89.6

$centered_score
[1] 5.861978e-15
```

1.21.4 `sapply()` and `vapply()`

`sapply()` attempts to simplify the output.

```
sapply(students, class)
```

```
          id         name      program        score    completed
   "integer"  "character"  "character"    "numeric"    "logical"
      passed centered_score
   "logical"    "numeric"
```

`vapply()` requires the expected output type:

```
vapply(students, typeof, character(1))
```

```
          id         name      program        score    completed
   "integer"  "character"  "character"     "double"    "logical"
      passed centered_score
   "logical"     "double"
```

This makes `vapply()` safer for reusable code.

1.21.5 `tapply()`

`tapply()` splits a vector into groups and applies a function within each group.

```
with(
  students,
  tapply(score, program, mean, na.rm = TRUE)
)
```

```
 MS PhD
 89  90
```

1.21.6 `Map()` and `mapply()`

These functions apply a function to corresponding elements of multiple inputs.

```r
Map(
  function(center, spread) rnorm(3, center, spread),
  center = c(0, 10),
  spread = c(1, 2)
)
```

```
[[1]]
[1]  0.9444969 -0.3233628 -1.3212824

[[2]]
[1]  7.857089 11.570026  8.406937
```

1.21.7 Choosing an iteration tool

Goal	Recommended starting point
Element-wise arithmetic	Vectorized function or operator
Standard matrix row/column summaries	`rowSums()`, `colSums()`, `rowMeans()`, `colMeans()`
General matrix row/column operation	`apply()`
Operation on list or data-frame columns	`lapply()`
Known scalar output type	`vapply()`
Grouped operation on one vector	`tapply()`
Data-frame columns in tidyverse	`dplyr::across()`
Complex state-dependent computation	Preallocated `for` loop

1.22 Packages and help

Packages extend R with functions, data, documentation, and statistical methods. The historical package count can be found https://www.datasciencemeta.com/rpackages.

```r
# CRAN
install.packages("packageName")

# Bioconductor
BiocManager::install("packageName")

# GitHub
remotes::install_github("username/repository")
```

Load an installed package:

```
library(packageName)
```

or call one function explicitly:

```
packageName::function_name()
```

Useful help tools include:

```
?mean
help("mean")
example(mean)
args(mean)
methods(mean)
```

1.23 The tidyverse

The **tidyverse** is a coordinated collection of R packages for data manipulation, visualization, importing, tidying, and programming.

```
install.packages("tidyverse")
```

```
library(tidyverse)
```

Important packages include:

```
library(dplyr)    # data manipulation
library(ggplot2)  # data visualization
library(readr)    # data import
library(tibble)   # tidy data frames
library(tidyr)    # data tidying
library(purrr)    # functional programming
```

Base R and tidyverse tools are complementary rather than competing systems.

1.23.1 Data frames and tibbles

A **tibble** is a modern data frame.

```r
students_tbl <- as_tibble(students)

students_tbl
```

```
# A tibble: 6 x 7
     id name  program score completed passed centered_score
  <int> <chr> <chr>   <dbl> <lgl>     <lgl>           <dbl>
1     1 Alice MS         90 TRUE      TRUE            0.400
2     2 Bob   PhD        85 TRUE      TRUE           -4.60
3     3 Chen  MS         NA FALSE     NA             NA
4     4 Divya PhD        94 TRUE      TRUE            4.40
5     5 Elena MS         88 TRUE      TRUE           -1.60
6     6 Farah PhD        91 TRUE      TRUE            1.40
```

```r
class(students_tbl)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

Some important differences are:

Operation	Base data frame	Tibble
Printing	May print many rows	Compact preview
`x[, "score"]`	May simplify	Remains a tibble
Partial matching	May occur in some contexts	More strict
Row names	Supported	Discouraged

Both are still data frames:

```r
is.data.frame(students_tbl)
```

```
[1] TRUE
```

1.23.2 Pipe operators

The native R pipe is

```
|>
```

and the commonly used magrittr/tidyverse pipe is

```
%>%
```

The pipe passes the result of one operation to the next function.

```
set.seed(777)

x <- rnorm(5)

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

```
[1] 0.37
```

```
# Native pipe
x |>
  mean() |>
  round(2) |>
  print()
```

```
[1] 0.37
```

The pipe makes the sequence of operations explicit.

A useful formatting rule is that `|>` should have spaces around it and normally appear at the end of a line.

1.23.3 Core `dplyr` verbs

Verb	Purpose
`select()`	Choose or reorder columns
`filter()`	Retain rows satisfying conditions
`arrange()`	Sort observations
`mutate()`	Create or modify variables
`summarise()`	Compute summaries

Verb	Purpose
`group_by()`	Define groups

Example:

```
students_tbl |>
  filter(completed, !is.na(score)) |>
  select(name, program, score) |>
  arrange(desc(score))
```

```
# A tibble: 5 x 3
  name  program score
  <chr> <chr>   <dbl>
1 Divya PhD        94
2 Farah PhD        91
3 Alice MS         90
4 Elena MS         88
5 Bob   PhD        85
```

Create variables and summarize by group:

```
students_tbl |>
  mutate(
    passed = score >= 70,
    score_z =
      (score - mean(score, na.rm = TRUE)) /
      sd(score, na.rm = TRUE)
  ) |>
  group_by(program) |>
  summarise(
    n = n(),
    n_observed = sum(!is.na(score)),
    mean_score = mean(score, na.rm = TRUE),
    sd_score = sd(score, na.rm = TRUE),
    .groups = "drop"
  )
```

```
# A tibble: 2 x 5
  program     n n_observed mean_score sd_score
  <chr>   <int>      <int>      <dbl>    <dbl>
1 MS          3          2         89     1.41
2 PhD         3          3         90     4.58
```

1.23.4 Applying functions across several columns

Inside `mutate()` or `summarise()`, `across()` applies functions to selected columns.

```
students_tbl |>
  summarise(
    across(
      where(is.numeric),
      list(
        mean = ~ mean(.x, na.rm = TRUE),
        sd = ~ sd(.x, na.rm = TRUE)
      ),
      .names = "{.col}_{.fn}"
    )
  )
```

```
# A tibble: 1 x 6
  id_mean id_sd score_mean score_sd centered_score_mean centered_score_sd
    <dbl> <dbl>      <dbl>    <dbl>               <dbl>             <dbl>
1     3.5  1.87       89.6     3.36            5.86e-15              3.36
```

This provides a tidyverse alternative to column-wise `lapply()` or `vapply()`.

1.24 Base R and tidyverse are complementary

For example, the same grouped mean can be obtained using either style.

```
# Base R
aggregate(
  score ~ program,
  data = students,
  FUN = mean,
  na.rm = TRUE
)
```

```
  program score
1      MS    89
2     PhD    90
```

```r
# tidyverse
students_tbl |>
  group_by(program) |>
  summarise(
    mean_score = mean(score, na.rm = TRUE),
    .groups = "drop"
  )
```

```
# A tibble: 2 x 2
  program mean_score
  <chr>        <dbl>
1 MS              89
2 PhD             90
```

The important questions are the same regardless of syntax:

- What object enters the function?
- What object does the function return?
- How are missing values handled?
- Are types and groups preserved?

1.25 Numerical computation and finite precision

Most real numbers cannot be represented exactly using binary floating-point arithmetic.

```r
0.1 + 0.2
```

```
[1] 0.3
```

```r
(0.1 + 0.2) == 0.3
```

```
[1] FALSE
```

```r
all.equal(0.1 + 0.2, 0.3)
```

```
[1] TRUE
```

For computed floating-point quantities, tolerance-based comparisons such as `all.equal()` are usually more appropriate than exact equality.

Numerical algorithms must also consider overflow, underflow, cancellation, conditioning, and convergence.

```
log(dbinom(500, size = 1000, prob = 0.01))
```

```
[1] -Inf
```

```
dbinom(
  500,
  size = 1000,
  prob = 0.01,
  log = TRUE
)
```

```
[1] -1618.143
```

The second computation is numerically safer.

1.26 Randomness and reproducibility

Pseudo-random numbers are generated deterministically from an internal state.

```
set.seed(8670)
rnorm(5)
```

```
[1] -0.4916281  1.1467524  1.5963118 -0.6655463  0.5516107
```

```
set.seed(8670)
rnorm(5)
```

```
[1] -0.4916281  1.1467524  1.5963118 -0.6655463  0.5516107
```

The two results are identical.

Set the seed at a meaningful boundary, usually once near the beginning of a simulation, rather than repeatedly inside the simulation loop.

Record the information necessary to reproduce the analysis:

```r
sessionInfo()
```

```
R version 4.6.0 (2026-04-24)
Platform: aarch64-apple-darwin23
Running under: macOS Tahoe 26.6.2

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.6/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.6/Resources/lib/libRlapack.dylib;  LAPACK

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/New_York
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
 [1] lubridate_1.9.5 forcats_1.0.1   stringr_1.6.0   dplyr_1.2.1
 [5] purrr_1.2.2     readr_2.2.0     tidyr_1.3.2     tibble_3.3.1
 [9] ggplot2_4.0.3   tidyverse_2.0.0

loaded via a namespace (and not attached):
 [1] gtable_0.3.6       jsonlite_2.0.0     compiler_4.6.0     tidyselect_1.2.1
 [5] scales_1.4.0       yaml_2.3.12        fastmap_1.2.0      R6_2.6.1
 [9] generics_0.1.4     knitr_1.51         pillar_1.11.1      RColorBrewer_1.1-
3
[13] tzdb_0.5.0         rlang_1.3.0        utf8_1.2.6         stringi_1.8.9
[17] xfun_0.60          S7_0.2.2           otel_0.2.0         timechange_0.4.0
[21] cli_3.6.6          withr_3.0.3        magrittr_2.0.5     digest_0.6.39
[25] grid_4.6.0         rstudioapi_0.19.0  hms_1.1.4          lifecycle_1.0.5
[29] vctrs_0.7.3        evaluate_1.0.5     glue_1.8.1         farver_2.1.2
[33] codetools_0.2-20   rmarkdown_2.31     tools_4.6.0        pkgconfig_2.0.3
[37] htmltools_0.5.9
```

1.27 Debugging and error messages

When an error occurs:

1. Read the entire error message.
2. Reduce the problem to the smallest example that still fails.
3. Inspect objects with `str()`, `typeof()`, `class()`, and `dim()`.
4. Use `traceback()` after an error when necessary.
5. Verify assumptions with functions such as `stopifnot()`.

```
safe_mean <- function(x) {
  stopifnot(
    is.numeric(x),
    length(x) > 0
  )

  mean(x)
}

safe_mean(c(2, 4, 6))
```

```
[1] 4
```

1.28 A computational workflow for this course

A reliable statistical computation separates:

1. **Inputs:** data, constants, configuration, and random seeds.
2. **Transformation:** cleaning and constructing analysis variables.
3. **Computation:** algorithms, models, optimization, or simulation.
4. **Validation:** diagnostics, tests, and sensitivity analysis.
5. **Communication:** figures, tables, interpretation, and limitations.

For example:

```
set.seed(8670)

n <- 200

x <- rnorm(n)
y <- 1 + 2 * x + rnorm(n, sd = 0.75)

simulated_data <- data.frame(
  x = x,
  y = y
```

```
)

fit <- lm(y ~ x, data = simulated_data)

coef(fit)
```

```
(Intercept)           x
   1.020794    1.991843
```

This script records where the data came from, fixes the random seed, creates an explicit data object, and passes that object to the statistical model.

1.29 In-class exercises

Predict the value and type of each expression before running the code.

```
c(TRUE, 2L, 3.5)

c(1, 2, 3, 4) + c(10, 20)

c(1, 2, NA) > 1

TRUE | NA
```

Using

```
x <- c(4, NA, 10, 15, 3, NA, 21)
```

write expressions that return:

1. all nonmissing values;
2. values greater than or equal to 10;
3. the positions of values greater than or equal to 10;
4. the mean of the observed values.

Without running the code, determine the result of `f(5)` and identify where each occurrence of `a` is found.

```r
a <- 10

f <- function(x) {
  a <- 2

  g <- function(y) {
    a * y
  }

  g(x)
}

f(5)
```

Write a function that takes a positive integer `n` and returns a vector whose ith element is

$$\frac{(-1)^{i+1}}{i}.$$

Preallocate the result and use `seq_len(n)`.

Using `students`, complete each task once with base R and once with `dplyr`:

1. retain students with observed scores;
2. select `name`, `program`, and `score`;
3. create a variable equal to `score - 90`;
4. sort from highest to lowest score;
5. compute the number of observed scores and mean score within each program.

Before running the code, predict whether each result is a vector, data frame, tibble, or grouped tibble.

For each task, choose among a vectorized function, `apply()`, `lapply()`, `vapply()`, `tapply()`, `dplyr::across()`, or a loop. Justify your choice and then write the code.

1. Compute the mean of every numeric column in `students`.
2. Compute the row sums of a $10,000 \times 50$ numerical matrix.
3. Record the class of every column in a data frame.
4. Compute the mean score within each program.
5. Fit the same model to each of 20 independently generated data sets and retain the fitted-model objects.

Predict the type and contents of `result`. Explain why `apply()` is inappropriate and rewrite the computation using `vapply()` or `dplyr::across()`.

```r
dat <- data.frame(
  subject = c("A", "B", "C"),
  pre = c(10, 12, 9),
  post = c(14, 13, 15)
)

result <- apply(dat, 2, mean)
```

1.30 Takeaways

- R is both a programming language and an environment for statistical computing.
- Objects and functions are the basic building blocks of R programs.
- Atomic vectors are the foundation of R computation.
- R uses **1-based indexing**.
- Vectorization, recycling, coercion, and missing-value propagation are central features of R.
- Matrices and arrays require homogeneous element types, whereas lists can contain heterogeneous objects.
- A data frame is a named list of equal-length columns.
- Tibbles are modern data frames with stricter and more predictable behavior.
- Functions have arguments, a body, and an environment.
- R uses lexical scoping and lazy evaluation.
- The apply family provides convenient abstractions for repeated computation but is not automatically faster than loops.
- `apply()` is designed primarily for matrices and arrays and can cause unwanted coercion when applied to mixed-type data frames.
- Base R and tidyverse tools are complementary.
- The pipe (`%>%` or `|>`) can make a sequence of data transformations easier to read.
- R's copy-on-modify behavior gives objects intuitive semantics, although memory allocation still matters.
- Statistical computing requires attention to floating-point arithmetic, missing values, reproducibility, and numerical stability.
- Understanding R's computational model makes code easier to predict, debug, optimize, and trust.

1.31 Optional challenge

Explain why these two functions may have very different memory and timing behavior even though they return the same values.

```r
grow <- function(n) {
  out <- numeric(0)

  for (i in seq_len(n)) {
    out <- c(out, i^2)
  }

  out
}

preallocate <- function(n) {
  out <- numeric(n)

  for (i in seq_len(n)) {
    out[i] <- i^2
  }

  out
}
```

Design a timing experiment for increasing values of `n`.

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

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

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

Additional introductory material is adapted from Why R?.

2 What is Computational Approaches

> "Statistical computing is more than programming. It is the process of using computation to connect statistical theory with practical data analysis. Programming provides the tools, but the main goal is to understand how a statistical method can be implemented, approximated, validated, and interpreted. A good computational solution should therefore be not only correct in code, but also statistically meaningful, numerically stable, and reproducible."

A computational approach is a systematic way to solve a statistical problem when an exact analytical solution is difficult, unavailable, or inefficient. Examples include using Monte Carlo methods to approximate expectations, numerical optimization to estimate parameters, bootstrap methods to approximate sampling distributions, and cross-validation to select tuning parameters. In this course, the emphasis is on understanding the connection

$$\text{Theory} \iff \text{Computation} \iff \text{Practice}.$$

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

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

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

qr(A)$rank
```

```
[1] 2
```

```
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

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

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

$X = (X_1, \dots, X_d)$ 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: $X \sim N_d(\mu, \Sigma)$.

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

```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
```

```r
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
```

> Take home message 1
>
> 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.

2.2.3 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.

```r
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))
```

```r
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 $\widehat{\beta}$ 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`?

2.2.4 Trade-off

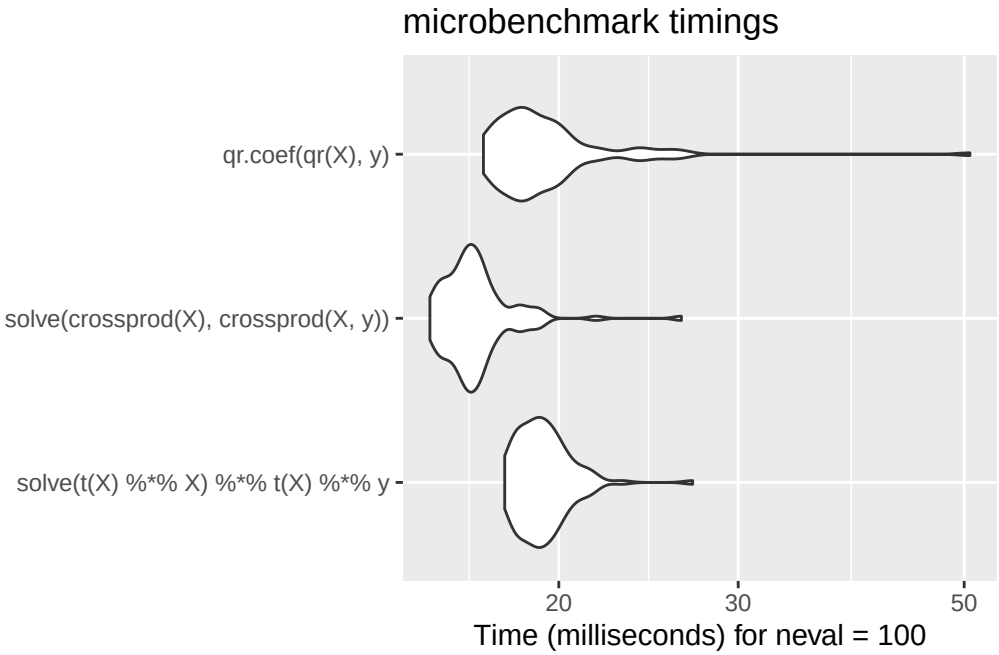
There isn't always elegance and flourish. When we take the robust approach, we accept that it comes at a cost.

```
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*)

2.2.5 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.

2.2.5.1 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)

# 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
```

2.2.5.2 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()
```

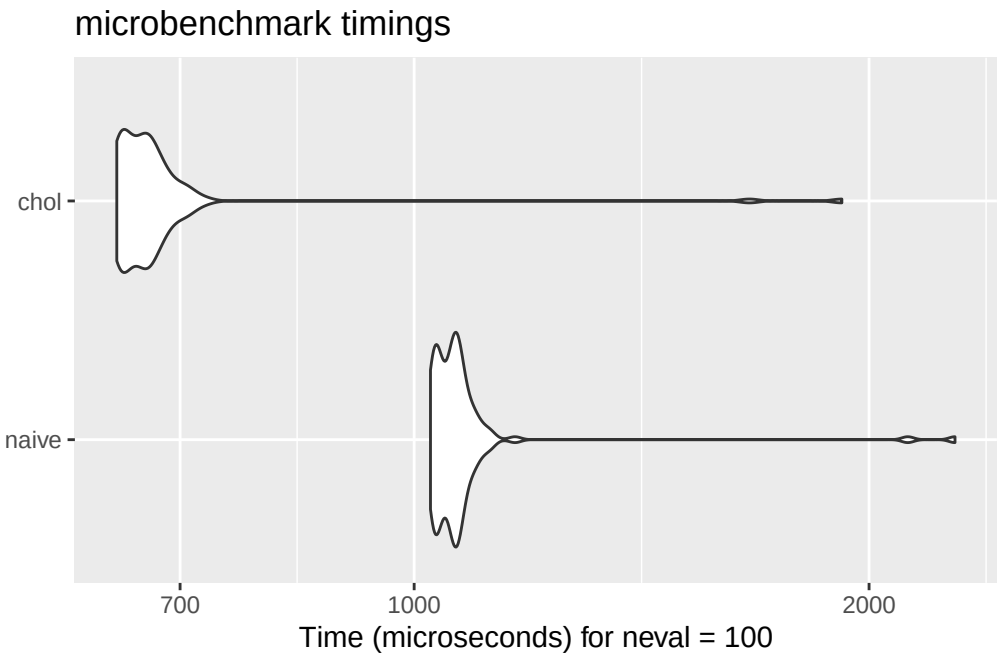
```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  70.67   93.61   99.94  100.54  107.31  126.73
```

2.2.5.3 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?)

2.2.5.4 Performance comparison

```r
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?

> **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!

Examples are borrowed from the following sources:

- Peng, R.D. Advanced Statistical Computing.

3 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.

3.1 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.

3.2 Type of Optimization Algorithms

There are in general two types of the optimization algorithms: (1). **deterministic** and (2). **metaheuristic**. Deterministic and metaheuristic algorithms represent two distinct paradigms in optimization.

*. **Deterministic methods**: such as gradient descent, produce the same solution for a given input and follow a predictable path toward an optimum.

*. In contrast, **metaheuristic approaches**: 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.

3.3 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.

3.3.1 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.

3.3.2 One-dimensional case

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

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

3.3.3 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?

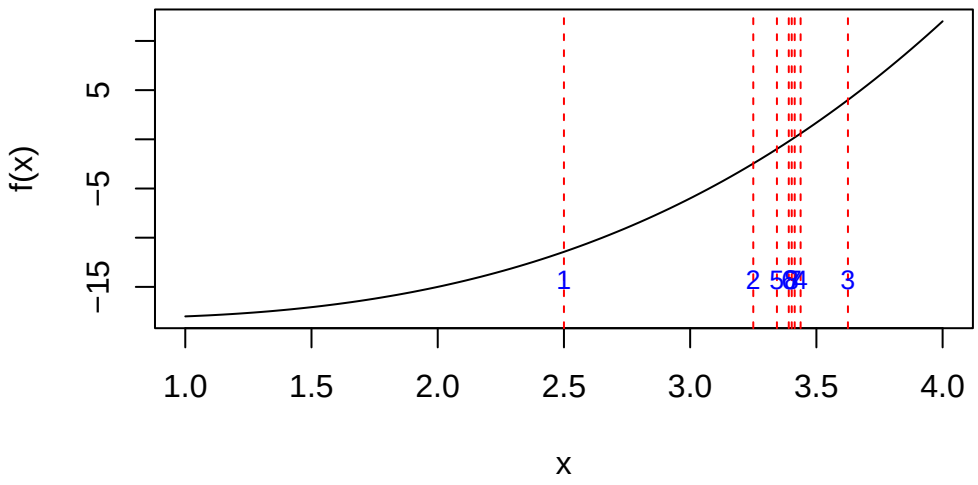
Q: Can this work for every function?

3.3.3.1 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)
```

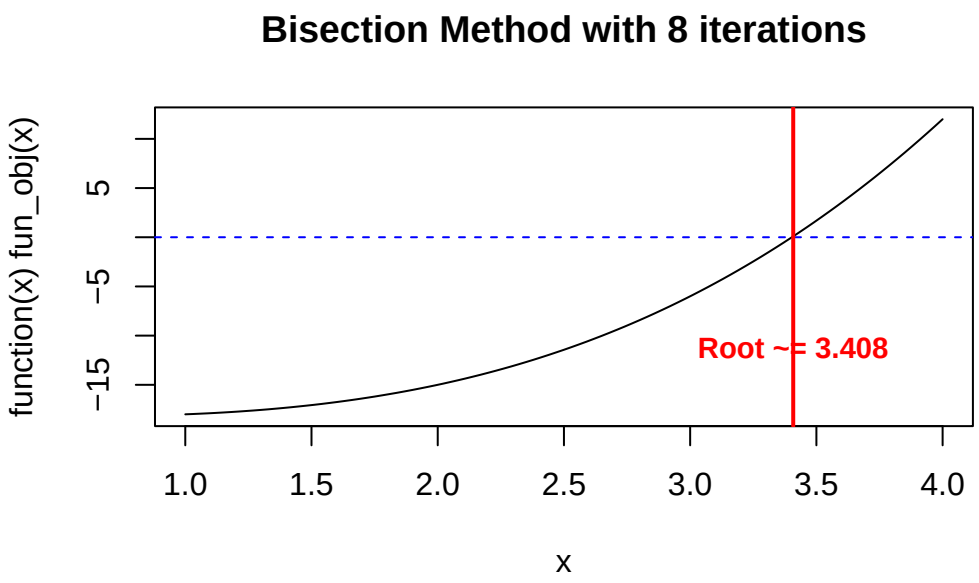
```
    }


    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
```

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



3.3.4 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

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

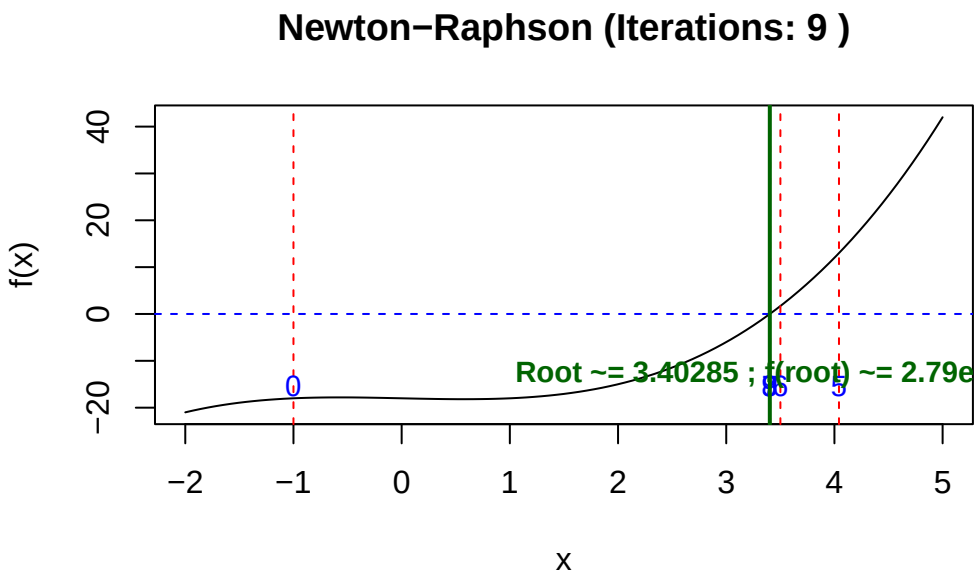
Step 4: Repeat Steps 2 and 3 until the values converge to a root or reach a desired tolerance.

```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)
```



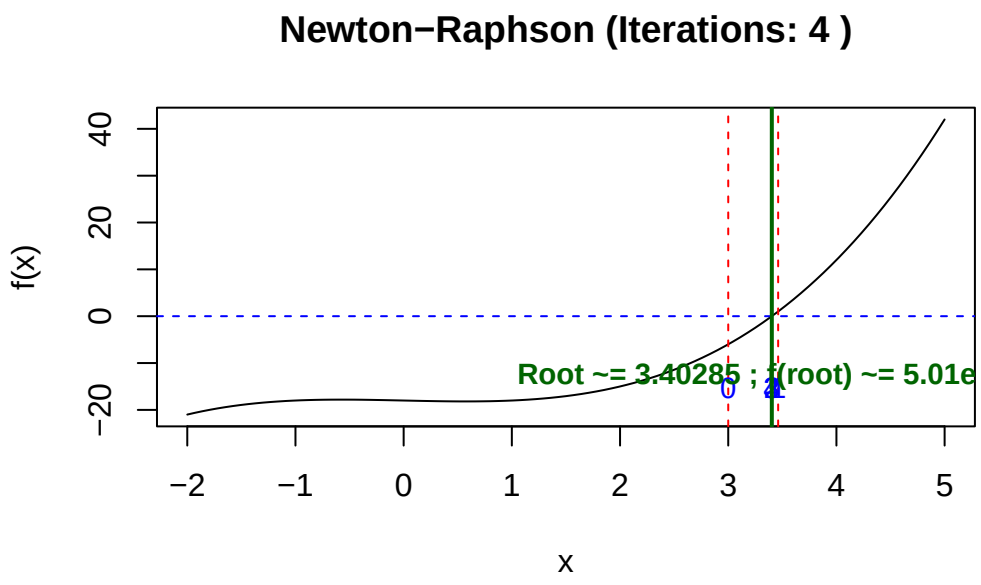

```
res
```

```
$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.

3.3.5 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

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

3.4 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.

3.4.1 In R

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

3.5 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

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

3.5.1 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.

3.5.2 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 $\mu = 0$.

3.5.3 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

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

3.6 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)

3.6.1 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

3.6.1.1 R implementation

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

3.7 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.

3.7.1 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.

Examples are borrowed from the following sources:

- Peng, R.D. Advanced Statistical Computing.