# Statistical Computing in the Age of AI

**A graduate textbook in biostatistics**

The rgtlab Curriculum Project

2026-04-29

GRADUATE BIOSTATISTICS SERIES

**Ronald "Ryy" G. Thomas**

# *Statistical Computing in the Age of AI*

**A graduate textbook in biostatistics**

First Edition · 2026

rgtlab

# Welcome

This is the online version of **Statistical Computing in the Age of AI** by The rgtlab Curriculum Project, a graduate textbook in statistical computing for biostatistics students.

The book is a practical, code-first introduction to the computing techniques that biostatisticians use daily: programming in R, version control, linear algebra, simulation, statistical modelling, graphics, reproducible software, and the emerging role of large language models in each of these activities.

It is written in the tradition of the Posit book family (*R for Data Science*, *Advanced R*, *R Packages*) and uses the same Quarto toolchain. The book is the introductory volume in a four-volume graduate sequence:

- *Biostatistics Practicum* covers the workflow infrastructure (Git, Docker, renv, Quarto, CDISC, SAS) that surrounds the methods.
- *Statistical Computing in the Age of AI* (this volume).
- *Advanced Statistical Computing in the Age of AI* covers the advanced material: numerical stability, numerical linear algebra in depth, advanced optimisation, EM, Monte Carlo, MCMC, modern Bayesian computation, high-performance and distributed computing, high-dimensional methods, machine learning, software engineering, interactive visualisation.
- *Applied Generative AI for Health Sciences Research* treats generative AI as the orthogonal axis: capability classes, reasoning models, biomedical RAG, multimodal medical AI, agents, evaluation, regulation, deployment.

See the Preface for motivation and the Conventions page for visual cues used throughout.

## License

This book is licensed to you under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

The code samples in this book are licensed under Creative Commons CC0 1.0 Universal (CC0 1.0), i.e. public domain.

# Statistical Computing in the Age of AI

*A graduate textbook in biostatistics.*

# Copyright

**Statistical Computing in the Age of AI**
A graduate textbook in biostatistics.

Copyright © 2026 Ronald "Ryy" G. Thomas.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit https://creativecommons.org/licenses/by-nc-nd/4.0/.

No ISBN has been assigned; this is an open-access teaching text. Citations should use the form:

> Thomas, R. G. (2026). *Statistical Computing in the Age of AI*. La Jolla, California. Retrieved from https://rgtlab.org/scai.

Typeset in Source Serif 4 and JetBrains Mono using Quarto and `xelatex`. Source available at https://github.com/rgt47/scai.

First draft: Spring 2026.

# Table of contents

# Table of contents

# Preface

## Why this book when an LLM can already 'do it'?

In April 2026, a graduate biostatistics student can open a browser, type 'bootstrap a 95% confidence interval for the median of this data', and receive working R code in under ten seconds. The code will usually run. It will often be correct. And it will sometimes be subtly, dangerously wrong in ways the student is not equipped to detect.

This is the premise of the book.

The premise is *not* that large language models are bad. They are remarkable tools, and ignoring them in a statistics curriculum would be a pedagogical failure. The premise is *not* that statisticians should refuse to use them. The book assumes, and encourages, daily LLM use in statistical computing work. The premise is that an LLM's usefulness to a biostatistician is limited by the statistician's own ability to **verify, critique, and extend** what the LLM produces, and that ability must be built through training that does not treat the LLM as a substitute for learning.

Put differently: the student who learns the material in this book will use LLMs more effectively than the student who does not. Not because this book teaches prompt engineering (it teaches a little of that, at the end of each chapter), but because this book teaches the statistical and computational judgment that LLM output *requires* from a human reviewer.

## What this book is not

This book is not:

- A textbook that pretends LLMs do not exist. A curriculum designed as if 2019 were still the current year trains students for a workplace that has ceased to exist.
- A textbook about LLMs as such. Prompt engineering, fine-tuning, and the engineering internals of language models are other subjects, covered better elsewhere.
- An argument that students should skip the hard parts of statistical computing because the LLM will handle them. The opposite is true: the arrival of LLMs *raised* the bar for what human statisticians must know, because deciding whether the LLM is right is now the statistician's core contribution.

The book is the middle path: teach statistical computing as it has always been taught, but with a consciously integrated treatment of where the human statistician contributes and where the LLM can be delegated to.

## How this book earns its subtitle

Three structural features distinguish the book from its predecessors and deliver on the 'in the Age of AI' framing:

1. **Front-loaded consciousness-raising.** Each content chapter opens, after the orientation, with a section titled *The statistician's contribution*. That section names the two to five decisions about the chapter's topic that cannot be delegated to an LLM without human input: is this method appropriate? what is the data structure? which variant of the procedure applies? These sections are short (roughly one printed page each) but are the intellectual scaffolding of the whole book.

2. **Adversarial LLM practice.** Each chapter closes with a section titled *Collaborating with an LLM on [topic]*, containing three prompts deliberately constructed to expose common failure modes. Each prompt is paired with a **Verification** step that tells the reader what to check and how. The prompts are not there to show off what LLMs can do; they are there to train the reader to catch what LLMs cannot.

3. **Verification-first pedagogy throughout.** Every non-trivial code example is paired with a concrete way to check the answer — a

closed-form result, an alternative implementation, a simulation, or a unit test. Code that cannot be verified is called out as such.

In addition, the book inherits the structural conventions of the Posit book family (Wickham, 2019; Wickham et al., 2023; Wickham & Bryan, 2023):

- A three-question *Prerequisites* quiz at the start of each chapter (in the style of *Advanced R*), with answers at the foot of the chapter.
- *Check your understanding* collapsible callouts distributed through the content, serving as paced comprehension prompts.
- A *Further reading* section curating the best next sources.

## Positioning against other books

A reader entering graduate biostatistics has many good statistical computing texts to choose from. This book does not replace any of them; it fills a gap that has opened in the past three years.

- *Advanced R* (Wickham, 2019) and *R for Data Science* (Wickham et al., 2023) teach R itself superbly, but predate widespread LLM use and do not address the human-LLM division of labour. Read them alongside this one.
- *Statistical Computing with R* (Rizzo, 2019) covers the algorithms in more depth than this book, also predates LLM integration, and is longer and more mathematical. Consult it when you want theory.
- The Posit book family on specific tools (*ggplot2*, *Mastering Shiny*, *R Packages*) remains the reference for those tools. This book cites them rather than replacing them.
- *Statistical Rethinking* (McElreath, 2020) is the best current introduction to Bayesian computation; this book's Bayesian chapter (Chapter 14) is a gateway, not a substitute.

What this book adds is a unifying pedagogical framework for statistical computing *in the presence of competent LLMs*. That framework is what the subtitle commits to, and it is what Parts I–VI of the book deliver.

## Prerequisites

Readers should have:

- Completed a one-quarter course in mathematical statistics at the level of Casella and Berger.
- Basic familiarity with linear algebra (vectors, matrices, eigenvalues).
- Access to R 4.4+ and RStudio (or another IDE of their choosing).

No prior experience with R or Git is assumed. Some prior exposure to an LLM (ChatGPT, Claude, Gemini, or similar) is helpful but not required.

## Conventions

See the Conventions page for the visual cues used throughout the book.

## How this book relates to its siblings

This volume is the introductory volume in a four-volume graduate sequence.

- *Biostatistics Practicum* covers the workflow that surrounds the methods: reproducibility, Git, Docker, renv, Quarto reporting, tidyverse data wrangling, CDISC, SAS, AI-assisted coding, and clinical-trial case studies.
- *Statistical Computing in the Age of AI* (this volume) covers the methods: programming in R, numerical algorithms, and the core inferential techniques (linear models, GLMs, mixed models, survival, Bayesian, bootstrap, simulation).
- *Advanced Statistical Computing in the Age of AI* is the second methods volume, covering numerical stability, numerical linear algebra in depth, advanced optimisation, EM, Monte Carlo, MCMC, modern Bayesian computation, high-performance and distributed computing, high-dimensional methods, machine learning for biostatistics, software engineering, and advanced interactive visualisation.

- *Applied Generative AI for Health Sciences Research* treats generative AI as the orthogonal axis: capability classes, reasoning models, biomedical RAG, multimodal medical AI, agents and the Model Context Protocol, evaluation, safety, regulation, and deployment.

Together the four books form a two-year graduate biostatistics curriculum. Each can be read independently.

## Acknowledgements

The structural conventions of this book (chapter-opening diagnostic quiz, end-of-chapter answers, section-level exercises) follow the model used in the Posit open-source textbook series (Wickham, 2019; Wickham et al., 2023; Wickham & Bryan, 2023). The tidyverse R packages maintained by the Posit engineering team are used throughout the code examples, and their consistent interface is reflected in the style of the prose.

The content rests on the foundational work of numerous authors in statistical computing, including Efron and Tibshirani (Efron & Tibshirani, 1993) on the bootstrap; Bates and colleagues (Bates et al., 2015) on `lme4`; Golub and Van Loan (Golub & Van Loan, 2013) on matrix computations; and Nocedal and Wright (Nocedal & Wright, 2006) on optimization. Remaining errors are the author's.

The graduate students who engaged with early drafts of this material contributed materially through their questions and their willingness to debate the appropriate role of large language models in a statistics curriculum.

Ronald "Ryy" G. Thomas
La Jolla, California
Spring 2026

# Conventions

This book uses a small set of visual conventions.

## In code blocks

- Input lines carry no prefix.
- Output lines are prefixed with `#>`. The whole block stays a valid R script: paste it into a console and it runs without error.
- Numbered discs in the code gutter (1, 2, 3) are annotations. Hover or tap them to read the annotation, or read the numbered list below the block.

A minimal example:

```
library(dplyr)

penguins |>                                                    ①
  filter(!is.na(body_mass_g)) |>                               ②
  group_by(species) |>                                         ③
  summarise(mean_mass = mean(body_mass_g))
```

① Start the pipeline with the `penguins` data frame.
② Drop rows with missing body-mass values.
③ Compute the species-level mean body mass.

## In prose

- **Bold** marks a term on its first use.
- `monospace` marks code, file paths, and package names.

- `package::function()` disambiguates function ownership when the package is not obvious from context.

## Callouts

> **Note**
>
> Clarification that is useful but not essential on first read.

> **Verification**
>
> A concrete check the reader should run. This is the book's signature callout and appears after every non-trivial example.

> **Pitfall**
>
> A failure mode reported by past students of the course.

> **LLM prompt**
>
> An adversarial prompt for use with a language model, paired with an explicit verification step.

# Introduction

## What is statistical computing?

Statistical computing sits at the intersection of three disciplines:

1. **Statistics** provides the *questions*, how to estimate parameters, quantify uncertainty, and test hypotheses.
2. **Computer science** provides the *tools* — algorithms, data structures, numerical stability, and software engineering practices.
3. **Domain science** (here, biomedicine) provides the *data*, messy, irregular, and consequential.

A biostatistician who is weak in any of the three produces work that is incorrect, slow, or irrelevant. This book aims to build competence in all three, and one more: the ability to collaborate effectively with a large language model while retaining full professional responsibility for the analysis.

## The question this chapter answers

Why learn any of this when an LLM will produce working R code on request? The preface gave the short answer; this chapter gives the long one.

The answer has two parts. The first part is *practical*: LLMs fail in specific, learnable, predictable ways, and a statistician who cannot detect those failures will sign her name to analyses that are wrong. The second part is *structural*: certain decisions in statistical computing cannot be delegated to an LLM in principle, because they depend on information the LLM does not have and cannot obtain through prompting.

## A framework for human-LLM collaboration

For every topic in this book, the statistician-LLM division of labour sorts into four categories.

**Category 1: Things the LLM does reliably.**

- Translating a correctly stated mathematical or algorithmic description into syntactically valid R code.
- Refactoring working code for style, readability, or idiom.
- Generating unit tests for a function whose behaviour is well specified.
- Explaining a familiar concept from canonical sources.
- Producing boilerplate for common patterns (a `purrr::map_dfr()` over a list of files; an `lm()` diagnostic quartet).

For these tasks an LLM is a fast, tireless collaborator, and you should use it freely. Verification is usually trivial: the code runs and produces the expected output.

**Category 2: Things the LLM does unreliably.**

- Choosing a method when several are plausible (`lm` versus `glm` versus `lmer` given a particular data structure).
- Handling edge cases it has not encountered many times in training (rare distributions, unusual study designs, combinations of features).
- Producing numerically stable implementations of classical algorithms without explicit guidance.
- Reading recent (post-training-cutoff) package APIs or conventions correctly.

For these tasks the LLM's output is often plausible but wrong. Verification requires you to know the answer yourself, at least at the level of 'what should the output look like?' The book trains you in that knowledge.

**Category 3: Things the LLM cannot do because it lacks information.**

- Knowing whether the data you are analysing were collected by a process that justifies the method the LLM proposes.

- Recognising dependence structures (clustering, time series, pedigrees) from the data itself, when those structures are not explicit in variable names.
- Judging whether an assumption 'holds' in your specific scientific context.
- Deciding what research question the analysis is meant to answer when the question is stated ambiguously.

These tasks require context the LLM does not and cannot have. They require the statistician's judgment by definition.

**Category 4: Things the LLM cannot do because of professional accountability.**

- Taking responsibility for an analysis submitted to a regulatory body or a journal.
- Standing behind the interpretation when challenged by a referee or a sceptical principal investigator.
- Certifying the reproducibility of the compendium.
- Signing the paper.

These responsibilities rest with the statistician, irrespective of how capable the LLM becomes.

## What LLMs cannot do: concrete examples

Abstract categories are less convincing than concrete failures. Here are four, one from each part of this book, all of which an LLM will cheerfully answer incorrectly if you do not guide it.

**Bootstrap a confidence interval for the maximum of a sample (Chapter 9).** The LLM will produce working code. The code will return a CI whose upper endpoint equals the observed sample maximum, because the bootstrap maximum cannot exceed the observed maximum. This is a known pathology of the bootstrap for extrema. An LLM will not warn you.

**Fit a Cox proportional-hazards model to a data frame with recurring patient visits (Chapter 13).** The LLM will produce

`coxph(Surv(time, event) ~ x, data = df)`. The fit will ignore the dependence between visits from the same patient and report standard errors that are too narrow. A robust-sandwich option exists (`cluster()`) but the LLM will not add it unless you know to ask.

**Apply a mixed-effects model to a small dataset with three clusters (Chapter 12).** The LLM will fit `lmer()`. The REML variance-component estimate for the cluster effect will be almost entirely driven by the finite-sample prior implicit in REML; the user will read the coefficient as an estimate of a population quantity it does not meaningfully estimate. The LLM will not distinguish a well-posed LMM from a poorly-posed one.

**Iteratively re-weighted least squares on a non-identifiable GLM (Chapter 11).** The LLM will run IRLS and return coefficients. IRLS on a non-identifiable model produces results that depend on the starting value; the LLM will return whichever answer its seed produced and present it with confidence. Detecting non-identifiability requires examining the design matrix or the likelihood, a step the LLM will not take unprompted.

Each of these failures corresponds to a content chapter of this book. The chapter teaches the concept; the *statistician's contribution* section at its head tells you what the LLM will get wrong on this topic; the *Collaborating with an LLM* section at its foot practises the verification that catches the error.

## The position this book takes

Given the four-category framework:

> **Use LLMs as an amplifier, not a replacement.** Treat every line of generated code as a hypothesis to be tested, not a result to be trusted.

Specifically, the book recommends that students:

- **Write the first version of any non-trivial function without AI assistance.** This ensures you understand the problem. An LLM-generated solution you did not attempt yourself is a solution whose bugs are invisible to you.

- **Use the LLM to critique, refactor, or generate test cases for code you wrote.** The LLM is excellent at these jobs, and you will be faster with its help.
- **Never submit code whose behaviour you cannot explain in plain English.** If you cannot say what each line does and why, you cannot verify it, and you should not present it as your work.

## Why the hard work still pays

Students occasionally ask whether this training is obsolete, whether, in five years, the LLM will be good enough that these warnings no longer apply. The honest answer is: maybe, and maybe not. But three arguments remain even under the optimistic assumption:

1. **Professional accountability does not transfer.** However capable the LLM becomes, when an FDA reviewer questions your submission or a referee challenges your paper, you, not the model, answer the questions.
2. **Verification always requires a second perspective.** Even perfect LLMs cannot audit their own output; you need the skills to do it.
3. **Your scientific contribution is the judgment, not the code.** A biostatistician who cannot contribute judgment is not a biostatistician; she is a prompt engineer. The market pays considerably more for the former.

In short: learning this material is not a hedge against LLM failure. It is the substance of what a biostatistician is for.

## What you will be able to do

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

- Write clean, vectorised, functional R code that is easy for collaborators to read and maintain.
- Manage your work with Git and GitHub, including resolving merge conflicts and collaborating via pull requests.

- Implement the core numerical algorithms of statistics , linear system solvers, decompositions, and optimizers, from first principles, and explain why packaged implementations are usually preferable.
- Design and execute a simulation study that produces defensible answers to a research question.
- Fit, diagnose, and interpret linear, generalised linear, mixed-effects, survival, and Bayesian models on real biomedical data, including recognising when each is and is not appropriate.
- Bootstrap inferential quantities for statistics whose analytic standard errors are unavailable, and recognise the cases where the bootstrap fails.
- Produce publication-quality graphics with `ggplot2` and build an interactive exploratory tool with Shiny.
- Scale a computation across cores with the `future` framework.
- Package a set of analysis functions as an R package, with unit tests and automated documentation.

And, across all of these:

- Collaborate productively with a large language model while retaining full professional responsibility for the analysis.

## How each chapter is structured

Every content chapter follows a consistent structure designed to make the human-LLM division of labour explicit:

1. **Prerequisites**, three open-ended diagnostic questions. Answer them honestly; if all three are easy, you can bypass the chapter. Answers appear in the *Quiz answers* section at the foot of the chapter.
2. **Learning objectives**, what you will be able to do by the end.
3. **Orientation**, why this chapter, in this place.
4. **The statistician's contribution**, the two to five decisions about this chapter's topic that the LLM cannot make on your behalf. Read this section first, and revisit it after you finish the chapter.
5. **Content sections**, substantive material, worked examples, and *Check your understanding* collapsible callouts placed at natural pause points.

6. **Collaborating with an LLM on [topic]**, three adversarial prompts paired with Verification steps, exercising the decisions from (4).
7. **Exercises**, three to five, to be attempted without LLM assistance the first time through.
8. **Further reading**, curated pointers for deeper study.
9. **Quiz answers**, responses to the Prerequisites questions.
10. **Practice test** (chapters with matching content in the course's exam bank), multiple-choice questions drawn from the course test bank.

## How to work through this book

For each chapter, the recommended workflow is:

1. **Read** the chapter through once, without running code. Do the Prerequisites quiz honestly.
2. **Replicate** the examples in your own R session. Type, do not copy-paste.
3. **Do** the exercises without consulting an LLM. Start over if you get stuck; check against the quiz answers or ask a peer before falling back on the LLM.
4. **Extend** by pasting the chapter's LLM prompts into a model and critiquing the responses. Confirm that each verification step actually catches the failure mode the prompt was designed to expose.
5. **Reflect** on what, concretely, the LLM contributed and what it could not. That reflection is the skill this book is designed to build.

Students who follow this workflow learn the material. Students who skip steps 2 and 3 do not, and within a year they cannot distinguish their own contributions from the LLM's, which means they cannot professionally account for either.

# Part I.

# Foundations

# 1. Introduction to R

## 1.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 1.14.

1. What is the difference between `c(1, 'a')` and `list(1, 'a')`?
2. In the expression `df[df$x > 0, ]`, what kind of R object is `df$x > 0`, and what rule does R use to combine it with the comma to select rows?
3. Predict the output of `x <- 1:5; x[-1]`, `x[c(TRUE, FALSE)]`, and `x[[1]]`. Which of these produces something that is *not* a numeric vector of length less than 5?

## 1.2. Learning objectives

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

- Install R and RStudio and configure a sensible working environment.
- Read the R documentation system (`?`, `??`, `help()`).
- Distinguish between atomic vectors, lists, and the major data structures built on them.
- Subset and mutate data with base R and with `dplyr`.
- Write vectorised expressions and explain why they outperform equivalent `for` loops.

## 1.3. Orientation

R is the *lingua franca* of statistical computing in academic biostatistics. It is not the fastest language, nor the most elegant, but it has by far the richest library of statistical methods and the largest community of statisticians reading, writing, and reviewing code. This chapter gets you fluent enough in R to read the rest of the book.

## 1.4. The statistician's contribution

Large language models can produce passable R code for almost any introductory task. Three decisions still require the statistician's judgment.

**1. Type discipline.** R coerces silently between types in ways that can hide errors. `c(1, '2')` becomes `c('1', '2')` (character); `TRUE + 1` becomes `2` (integer). An LLM will happily accept whatever types flow through its generated code. You must decide, before any computation runs, which types the data *should* carry, and assert them with `stopifnot(is.numeric(x))` where it matters.

**2. Base R vs tidyverse framing.** The same transformation can be expressed in base R and in `dplyr`. An LLM will lean toward whichever style dominated its training data, usually `dplyr` for interactive work. The performance, dependency, and readability tradeoffs belong to you. Pick the style that fits the project's needs, not the model's default.

**3. Edge cases in statistical data.** R's built-in functions handle `NA`, `NaN`, `Inf`, zero-length vectors, and length-mismatched vectors in different ways. The defaults are usually right for general-purpose data; they are often wrong for biomedical data where missingness is informative. An LLM will not check whether your data has the edge cases that would break its generated code. You will.

## 1.5. Installing R and RStudio

R is a free and open-source implementation of the S programming language. Download the base R installer from the Comprehensive R Archive Network

(CRAN); Posit's RStudio Desktop, a free integrated development environment, is available at https://posit.co/download/rstudio-desktop/. Install R first, then RStudio.

On macOS, the `rig` command-line tool (https://github.com/r-lib/rig) lets you install and switch between R versions cleanly:

```
brew install --cask rig
rig add release        # installs the current stable R
rig default release
```

Verify the installation:

```
R --version
# R version 4.5.3 (2026-02-28) -- 'Single Candle'
```

Launch RStudio. The default four-pane layout gives you a script editor, a console, an environment inspector, and a files/plots/help browser.

## 1.6. The R console, scripts, and projects

An R session is anchored to a working directory and a set of loaded packages. Two conventions make sessions reproducible.

**Use RStudio projects.** A `.Rproj` file tells RStudio that the enclosing directory is a project; opening the file sets the working directory automatically and isolates the session from the rest of your filesystem. Always work inside a project, never from a stray session rooted in `~/`.

**Prefer scripts over the console.** The console is for quick exploration; scripts are for work that must be reproducible. A script is a `.R` or `.qmd` file you can rerun from top to bottom and obtain the same result. Analyses that live only in the console history evaporate when the session ends.

The `.Rprofile` file in a project runs automatically when you open a session. Use it to set session-wide options like the CRAN mirror or the preferred number of significant digits. `renv` extends this pattern to pin package versions; package and environment management is covered in detail in the practicum companion volume.

## 1.7. Atomic vectors and coercion

Everything in R is built from vectors. Even a single number like `5` is a vector of length 1.

R has five primary atomic types: `logical`, `integer`, `double`, `character`, and `complex`. All elements of an atomic vector share a type. If you try to combine mixed types, R silently coerces them to the most general type in the chain `logical → integer → double → character`.

```
c(TRUE, 1L)         # -> integer: 1, 1
c(TRUE, 1L, 2.5)    # -> double:  1.0, 1.0, 2.5
c(TRUE, 1L, 'x')    # -> character: 'TRUE', '1', 'x'
```

Silent coercion is a common source of bugs. `sum(c(1, 2, NA))` returns `NA`, not `3`. `mean(c(TRUE, FALSE, TRUE))` returns `0.667` because the logicals coerce to `c(1, 0, 1)`, which is numerically correct and often useful but can surprise readers.

**Lists** differ from atomic vectors in that their elements can each carry a different type:

```
list(1, 'a', TRUE)    # length-3 list, three distinct types
```

Operations that work on atomic vectors often do not work on lists; reach for `lapply()`, `sapply()`, or `purrr::map_*()` when iterating over list elements (Chapter 4).

**Special values:**

- `NA` is missing data. Most operations involving `NA` return `NA`.
- `NULL` is the absence of an object. Assigning `NULL` to a list element removes it.
- `NaN` is 'not a number' (for example, `0/0`).
- `Inf` and `-Inf` are the signed infinities (for example, `1/0`).

Check your understanding

**Q:** What does `c(1, 'a')` return, and why?
**A:** The character vector `c('1', 'a')`. R cannot store `1` and `'a'` in the same atomic vector, so it coerces both to the most general type (character). To keep the original types, use `list(1, 'a')`.

## 1.8. Subsetting: [, [[, and $

R offers three subsetting operators and four kinds of indices. Confusion between them is the most common source of errors in a first R course.

**The three operators.**

- `[` returns a subset of the same type as the original. For an atomic vector it returns a vector; for a list it returns a list; for a data frame it returns a data frame.
- `[[` returns a single element, stripping the container. From a list, `[[` gives you the element itself (which might be any type); from an atomic vector, `[[` returns a length-1 atomic of the same type.
- `$name` is shorthand for `[['name']]` on a list or data frame.

**The four index types.**

```
x <- c(10, 20, 30, 40, 50)

x[2]              # positive integer: 20
x[c(1, 3, 5)]     # vector of positive integers: 10, 30, 50
x[-1]             # negative integer: drop element 1 -> 20, 30, 40, 5
x[x > 25]         # logical: 30, 40, 50
x[c('a', 'b')]    # character: works only if x has names
```

For data frames, `[row, col]` applies the subsetting rules along both dimensions. The expression `df[df$x > 0, ]` uses a logical row index (`df$x > 0`) to keep rows where `x` is positive and a missing column index to keep all columns.

```r
df <- data.frame(
  id  = 1:5,
  x   = c(3, -1, 5, 0, 8),
  grp = c('a', 'b', 'a', 'b', 'a')
)

df[df$x > 0, ]        # rows where x is positive
df[df$grp == 'a', ]   # rows in group 'a'
df[, c('id', 'x')]    # two columns, all rows
```

A logical index built from a column that contains `NA` produces `NA` rows, not dropped rows. This surprises people. Use `which()` or `subset()` if you want `NA` rows dropped.

## 1.9. Vectorisation

Vectorisation is applying an operation to every element of a vector at once, rather than writing an explicit loop. It is the central design principle of R: nearly every built-in function is vectorised.

```r
x <- c(1, 2, 3, 4, 5)
x + 10                # 11 12 13 14 15, elementwise
x^2                   # 1 4 9 16 25
sqrt(x)               # 1.00 1.41 1.73 2.00 2.24
```

When two vectors of different lengths combine, R recycles the shorter vector to match the longer:

```r
c(1, 2, 3, 4) + c(10, 20)    # 11 22 13 24  (recycled)
```

Recycling without warning is convenient for scalar + vector arithmetic but dangerous when lengths mismatch by accident. R warns if the longer length is not a multiple of the shorter, but not otherwise.

**Loop vs vectorised.** A concrete benchmark makes the stakes visible:

```r
library(bench)

vectorised_zscore <- function(x) (x - mean(x)) / sd(x)

loop_zscore <- function(x) {
  z <- numeric(length(x))
  m <- mean(x)
  s <- sd(x)
  for (i in seq_along(x)) z[i] <- (x[i] - m) / s
  z
}

bench::mark(
  vectorised = vectorised_zscore(iris$Sepal.Length),
  loop       = loop_zscore(iris$Sepal.Length),
  iterations = 100
)
```

The vectorised version is typically 10–100× faster because the elementwise arithmetic happens in compiled C code, not in R's interpreter. For a vector of length 1 million the gap grows to 1000× or more.

> Check your understanding
>
> **Q:** Why is vectorised R code faster than an equivalent `for` loop that does the same arithmetic?
> **A:** The vectorised operation hands the whole vector to a compiled C routine, which runs the loop internally at native speed. An R-level `for` loop invokes the R interpreter once per element, and the interpreter overhead dominates the actual arithmetic for simple operations.

Not every algorithm vectorises. Sequential updates (MCMC chains, sequential hypothesis tests, change-point detection) require each step to depend on the previous one and cannot be written as single elementwise expressions. For these cases, `Rcpp` (Chapter 3) lets you write a C++ loop that R calls as a single function.

## 1.10. Tidyverse essentials

The tidyverse is a coordinated set of packages (`dplyr`, `tidyr`, `ggplot2`, `purrr`, and others) that share a common grammar for data manipulation and visualisation. `dplyr` provides six core verbs and a pipe operator that chains them.

The native R pipe `|>` (introduced in R 4.1, 2021) passes its left-hand side as the first argument to the right-hand side:

```
x |> mean()          # same as mean(x)
x |> round(digits = 2)  # same as round(x, digits = 2)
```

Earlier tidyverse code uses the magrittr pipe `%>%`. The two are similar; this book uses the native pipe exclusively.

The six `dplyr` verbs apply to data frames:

| Verb | Action |
|---|---|
| `filter(df, condition)` | Keep rows where condition is TRUE |
| `select(df, cols)` | Keep or drop columns |
| `mutate(df, new = ...)` | Add or modify columns |
| `summarise(df, ...)` | Reduce to a single row (or one per group) |
| `arrange(df, col)` | Sort rows |
| `group_by(df, col)` | Group for later verbs |

A typical pipeline:

```
library(dplyr)

ChickWeight |>
  filter(Time <= 14) |>
  group_by(Diet, Time) |>
  summarise(
    mean_weight = mean(weight),
    n           = n(),
    .groups     = 'drop'
```

```
  ) |>
  arrange(Diet, Time)
```

The pipe reads top-to-bottom like an analysis plan: start with `ChickWeight`, keep the first 14 days, group by diet and day, summarise, sort. The same computation in base R would fragment into intermediate variables and nested function calls; the pipe keeps the data flow visible.

Check your understanding

**Q:** What does `df |> group_by(x) |> summarise(m = mean(y))` return that `summarise(df, m = mean(y))` does not?
**A:** A data frame with one row per distinct level of `x`, giving the mean of `y` within each group. Without `group_by()`, the second form collapses the whole data frame to a single row.

## 1.11. Collaborating with an LLM on R essentials

Section Section 1.4 named three decisions that require statistician judgment. The prompts below exercise each one. Treat the LLM's response as a hypothesis to verify, not a result to trust.

### 1.11.1. Type coercion

**Prompt.** 'Read this CSV and compute the mean of the `dose` column', then paste a CSV whose `dose` column mixes numeric strings and the literal string `'NA'`.

**What to watch for.** `read.csv()` treats `'NA'` strings as the missing sentinel by default, but `readr::read_csv()` has different defaults depending on version. A `dose` column with mixed content can silently come in as `character`, and `mean()` on a character vector errors. An LLM may wrap the read in `as.numeric()` without warning you that values that fail to parse become `NA`.

**Verification.** Inspect `str(df$dose)` after the read. Confirm the type is `numeric` and the count of `NA` values matches the count of non-parseable source strings.

### 1.11.2. Base R vs tidyverse

**Prompt.** 'Write R code to compute, for each level of a grouping variable, the mean of another variable, dropping missing values.'

**What to watch for.** An LLM will usually produce the `dplyr` form (`group_by + summarise(mean(x, na.rm = TRUE))`). It may not ask whether the project uses tidyverse at all. Production R code in some settings (pharma, legacy packages, FDA submissions via pharmaverse internals) runs on base R or `data.table` and imports no tidyverse.

**Verification.** Confirm the project's dependency policy before accepting the suggestion. If tidyverse is permitted, keep the LLM's code. If not, ask for the base R equivalent (`aggregate(y ~ group, data = df, FUN = mean)`) or the `data.table` equivalent.

### 1.11.3. Edge cases

**Prompt.** 'Write a function that computes the ratio of two columns in a data frame.'

**What to watch for.** An LLM will typically write `df$x / df$y` without handling the cases that make the ratio undefined: zero denominators (produces `Inf`), negative denominators (changes sign), `NA` inputs (propagate), or zero-row inputs (returns a zero-length numeric). In a biomedical analysis each case signals a real data phenomenon: zero divisor as a structural zero, missing as a censored measurement, and so on.

**Verification.** Ask the LLM to list the inputs on which the function would produce unexpected output. Extend the function to raise an error (with an explanatory message) rather than silently returning `Inf` or `NaN` for cases that indicate broken data.

## 1.12. Exercises

1. Write a function `summarise_numeric(df)` that takes a data frame and returns a tibble with one row per numeric column giving its mean, median, SD, and number of missing values. Do not use any package outside base R.
2. Repeat exercise 1 using `dplyr::summarise()` across all numeric columns. Compare the two implementations for readability and speed.
3. Explain, in at most three sentences, why `sum(c(1, 2, NA)) == NA` returns `NA` rather than `TRUE`.

## 1.13. Further reading

- (Wickham et al., 2023), the canonical first book for R data analysis; the 2nd edition uses the native pipe throughout.
- (Wickham, 2019) Chapters 1-3, essential for reading other people's R code.
- (Grolemund & Wickham, 2017), the 1st edition is still a readable slower-paced introduction to the tidyverse.

## 1.14. Prerequisites answers

1. `c(1, 'a')` coerces both elements to the most permissive atomic type, producing `c('1', 'a')` (character). `list(1, 'a')` preserves the types of each element because lists are not atomic.
2. `df$x > 0` is a logical vector of length `nrow(df)`. R applies logical subsetting along the first dimension of the data frame so that `df[df$x > 0, ]` keeps only the rows where the logical vector is `TRUE`. Rows where `df$x > 0` is `NA` are returned as rows of all `NA`.
3. `x[-1]` drops the first element and returns `2:5`. `x[c(TRUE, FALSE)]` recycles the logical vector to length 5 and returns `c(1, 3, 5)`. `x[[1]]` returns the scalar `1`, which is still a length-1 numeric vector; the `[[ ]]` form is important conceptually (it strips list/vector structure) but on atomic vectors it behaves like `[ ]` with a single index. All three results are numeric vectors of length less than 5.

# 2. Version Control with Git

## 2.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 2.19.

1. What is the primary purpose of version control, and how does it differ from keeping backup copies of files with dated filenames?
2. What is the effect of running `git init` in a directory, and when would you use it rather than `git clone`?
3. What is the difference between `git add` and `git commit`, and why does Git separate these two operations?

## 2.2. Learning objectives

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

- Explain what a commit, a branch, and a remote are, and draw the three-stage workflow (working directory, staging area, repository).
- Initialise a repository, make commits with informative messages, and push to a remote on GitHub.
- Use `git status`, `git diff`, `git log`, and `git blame` to understand the current state and history of a project.
- Create a feature branch, merge it back into `main`, and resolve merge conflicts without losing work.
- Write a `.gitignore` appropriate for an R project, including entries for `renv`, RStudio scratch files, and large outputs.
- Open a pull request, review a colleague's pull request, and explain why pull requests are the dominant form of code review in biostatistical teams.

## 2.3. Orientation

Before we write any statistics, we need a way to keep track of the code we have written. Git is the tool almost every data scientist uses for this. It is notoriously user-hostile on the surface, the error messages are famously unhelpful and the man pages seem to assume you already understand what they are trying to explain, but its core ideas are simple, and the payoff of fluency is enormous. No more `analysis_final_v3_really.R`. No more emailing zipped folders back and forth. No more quiet dread when a collaborator asks 'which version produced the numbers in Table 2?'.

Version control is also the single best tool for building a defensible audit trail for a statistical analysis. When a regulator, a reviewer, or a future-you asks 'why did the point estimate change between the February and April drafts?', a well-kept Git history produces an immediate, specific answer. That answer is far more convincing than a remembered narrative.

## 2.4. The statistician's contribution

The mechanics of Git are boring: `add`, `commit`, `push`, `pull`, `merge`. Anyone, including an LLM, can recite them. What is not boring, and what no tool can automate, is the judgement about *what* and *when* to commit, *what* to write in the commit message, and *when* to branch.

These judgements shape how trustworthy your analysis looks months or years later. They are the statistician's contribution to a workflow that would otherwise dissolve into an opaque pile of incrementally-edited files.

**What belongs in a single commit.** A good commit is one logical change: a bug fix, a new function, a reorganisation of an analysis script. If the description of your commit uses the word 'and', you almost certainly should have split it in two. Atomic commits make `git bisect` usable (you can binary-search the history to find the commit that introduced a bug) and make pull-request review tractable. Lumping three unrelated changes into one commit is a small kindness to your present self and a substantial tax on everyone who reads the history later, including you in six months.

**What a commit message should say.** The imperative first line ('Add bootstrap CI helper', not 'Added bootstrap CI helper' or 'Bootstrap CI stuff') is the convention; the first line should complete the phrase 'If applied, this commit will _____.' For any change whose motivation is not obvious, add a blank line and a longer paragraph explaining *why*. Statistical code especially benefits from this: 'Switch from OLS SEs to HC1 sandwich estimators after residual diagnostics indicated heteroscedasticity' ages far better than 'update SEs'.

**When to branch.** A branch is appropriate whenever you want to explore something that might not work out: a sensitivity analysis that could reveal the primary model is fragile, a refactor that might turn out to be worse than what you started with, a new collaborator's contribution that needs review before it lands. Trivial changes can go straight to `main`. The trap is in between, medium-sized changes where you tell yourself it will only take an hour. It usually does not.

**What not to commit.** `renv.lock`, yes. `.Rproj.user/`, no. Raw data: almost never, and never if it is protected health information or subject to a data-use agreement. Rendered outputs (`.pdf`, `.html`) are usually derived artefacts and should be excluded unless they are the deliverable. This distinction, between source (commit) and output (ignore) — is the same distinction that underlies reproducible research: the source should be sufficient to regenerate the outputs.

These are judgement calls. They depend on the nature of the project, the sensitivity of the data, and the audience for the history you are building. An LLM can generate a `.gitignore` in five seconds, but it cannot tell you whether a particular intermediate file is a reproducible artefact or a costly computation whose result you should preserve.

## 2.5. Why version control?

The concrete benefits of Git are easy to list:

- It tracks every change to every file, with author, timestamp, and message, producing a complete audit trail.

- It allows parallel work: several analysts can edit the same codebase simultaneously without stepping on each other.
- It provides a safety net for experimentation. Branches let you try a risky change knowing you can discard it cleanly.
- It documents institutional knowledge. When a postdoc leaves or a collaborator moves on, their thinking is preserved in the history rather than lost.
- It integrates with tooling (continuous integration, code review, issue tracking) that compounds these benefits.

The less concrete but arguably more important benefit is psychological. When you know you can revert any change in seconds, you become bolder about trying things. You refactor more aggressively, experiment more freely, and make fewer cautious edits hedged against a non-existent worst case. The net effect on code quality is large and compounding.

Check your understanding: Git vs. backup copies

**Question.** What distinguishes Git from simply keeping backup copies of your files (`analysis_v1.R`, `analysis_v2.R`, `analysis_final.R`)?
**Answer.**
Three things. First, Git tracks *changes* (the diff between versions) rather than storing whole-file copies, so the history is compact and readable, you can ask 'what changed between these two versions?' and get a precise answer. Second, every change carries context: an author, a timestamp, and a message explaining *why* the change was made. Third, Git supports collaboration workflows that dated backups cannot: parallel branches, systematic merging, review of proposed changes before they land. The filename trick preserves file states; Git preserves the reasoning behind each state.

## 2.6. The three-stage workflow

The single most important conceptual model in Git is its three-stage workflow:

1. **Working directory**: the files as you are editing them.

2. **Staging area** (also called the ‘index’): changes you have marked for inclusion in the next commit.
3. **Repository**: the committed history.

Many students initially find the staging area confusing. Why not just ‘write code, commit code’? The answer is that the staging area gives you fine-grained control over what goes into each commit. Suppose you spend a morning fixing a bug in `clean.R`, adding a new function to `plot.R`, and adding a sentence to `README.md`. These are three logically distinct changes that should probably be three commits. The staging area lets you stage `clean.R`, commit it with the message ‘Fix off-by-one error in age imputation’; then stage `plot.R`, commit it with ‘Add density-ridge helper’; then stage the README. The resulting history tells three small, comprehensible stories instead of one sprawling one.

Check your understanding: Partial staging

**Question.** Why might you want to stage only some changes rather than committing all modified files at once?
**Answer.**
In a typical analysis session you often make several logically unrelated changes at once: a fix to the data cleaning script, a new visualisation, and a tweak to the README. These should be three commits, not one. Staging only the relevant subset lets each commit carry a focused message. Future readers, including you, debugging this six months from now, can then read the history and understand each change on its own terms. Lumping everything into one commit makes `git bisect` useless and makes code-review discussions hopelessly tangled.

## 2.7. The minimum viable Git workflow

For day-to-day work, the overwhelming majority of Git usage is a small cycle of commands.

```
# initialise a new repository in the current directory
git init
```

```
# see what is modified, staged, and untracked
git status

# stage one file, or stage everything changed
git add scripts/clean.R
git add .

# record the staged snapshot in the history
git commit -m "Fix missing-value handling in cleaning script"

# see the history
git log --oneline -10

# push local commits to the remote (e.g. GitHub)
git push

# fetch remote commits and merge them into your branch
git pull
```

Five of these, `status`, `add`, `commit`, `push`, `pull` — cover perhaps 90% of daily Git use. Mastery of these basics is far more important than memorising obscure options.

A disciplined loop looks like this:

1. `git pull` before you start, to bring in any changes others have pushed.
2. Edit files.
3. `git status` to see what you have changed.
4. `git diff` to inspect the changes.
5. `git add` the changes you want to include in this commit.
6. `git commit -m "..."` with a clear, imperative message.
7. Repeat from step 2 until the change is complete.
8. `git push` to publish.

Commit often, five to ten times per session is not unreasonable for an engaged working day. A granular history is cheap, and it is easier to squash small commits later than to split a big one.

## 2.8. Branches and merging

A branch is a pointer to a commit that moves forward as you make new commits. Branches are, contrary to some intuitions, not copies of your files, switching branches changes which pointer you are on and updates the working directory to match. This is why creating and switching branches in Git is nearly instantaneous, and why branches are a first-class tool for parallel work rather than a heavyweight operation.

```
# create a branch
git branch sensitivity-analysis

# switch to it (traditional)
git checkout sensitivity-analysis

# create and switch in one step (modern, preferred)
git switch -c sensitivity-analysis

# merge it back into main when done
git switch main
git merge sensitivity-analysis
```

The standard mental model: `main` contains the 'truth' of your project, the primary, reviewed, runnable analysis. When you want to explore a sensitivity analysis, test a new method, or take on a risky refactor, you branch. Work proceeds in isolation. If the branch pans out, you merge it into `main`. If it does not, you discard it (`git branch -D branch-name`) and nothing is lost.

Biostatistical examples where branches earn their keep:

- **Sensitivity analyses** on a pre-registered primary model. The main branch preserves the pre-registered analysis; a `sensitivity-MNAR` branch explores missing-not-at-random assumptions. Whether or not you merge, the pre-registered analysis remains inviolate.
- **Alternative methods** compared head-to-head. One branch implements a Cox proportional hazards model, another a parametric model, a third a random-forest approach. Each develops independently; the final paper merges whichever proves most defensible.

- **A collaborator's contribution.** They work on a branch, open a pull request, you review it, and it merges only after the review passes.

Check your understanding: When to branch

**Question.** In what scenarios would creating a branch be beneficial for a statistical analysis project?
**Answer.**
Any time you want to explore something that might not work out and you want `main` to keep representing a known-good state. Common examples: sensitivity analyses on a pre-registered primary model, comparisons between competing statistical methods, risky refactors, contributions from collaborators that need review before they land. For trivial edits, a typo fix, a small prose change, the overhead of a branch is not worth it. The judgement call is in the middle: medium-sized changes where the work is not trivially safe. In practice, err toward branching; the cost of a wasted branch is nearly zero.

When branches have developed in parallel, merging them can either succeed automatically (when the changes touch disjoint lines) or produce a *conflict* (when the same lines were edited differently on each branch). We treat conflict resolution in its own section below.

## 2.9. Remotes: GitHub and collaboration

A *remote* is a named reference to a copy of the repository hosted elsewhere, typically on GitHub. The conventional name for the primary remote is `origin`. Remotes are how repositories synchronise: `git push` sends your commits to the remote, `git pull` brings remote commits into your local copy.

Git is the version control system; GitHub is a company that hosts Git repositories and provides a suite of collaboration tools on top. They are separate layers, even though in modern workflows they are deeply intertwined. GitLab, Bitbucket, and Gitea are similar hosted services; the concepts transfer directly.

A minimal setup sequence for a new project:

```
# locally
cd ~/research/my-analysis
git init
git add .
git commit -m "Initial project structure"

# create a repository on GitHub (via the web UI or gh CLI)
gh repo create my-analysis --private --source=. --remote=origin

# push the initial commit
git push -u origin main
```

The `-u` flag on the first push sets `origin/main` as the upstream tracking branch, so subsequent `git push` and `git pull` commands need no arguments.

RStudio integrates with Git as well. File modifications appear in the Git pane as checkboxes you can stage; commit, pull, and push buttons are one click away. The integration is convenient for the common cycle of pull, edit, stage, commit, push, but it is not a substitute for understanding the underlying concepts. When something goes wrong, and eventually something will, the command line is where you debug.

Check your understanding: RStudio integration

**Question.** What advantages does using Git through RStudio provide compared to using Git from the command line?
**Answer.**
The RStudio Git pane gives a visual view of what files are modified, staged, and untracked, with checkboxes for staging and a built-in diff viewer for reviewing changes. For the common pull-edit-stage-commit-push loop, this is faster than typing commands and less error-prone for beginners. The trade-off is that RStudio exposes only a subset of Git's operations. For branching workflows, conflict resolution, interactive rebasing, or recovering from mistakes, you will fall back to the command line. Most practitioners end up using both: the pane for the simple cycle, the command line when anything unusual happens.

## 2.10. Pull requests and code review

A pull request (PR) is a proposal to merge changes from one branch into another. Instead of pushing directly to `main`, you push a feature branch, open a PR against `main`, and invite review. A PR is far more than a merge mechanism: it is a checkpoint at which team members read the code, ask questions, request revisions, and approve.

The typical PR lifecycle:

1. Create a branch: `git switch -c fix-cleaning-bug`.
2. Make commits.
3. Push the branch: `git push -u origin fix-cleaning-bug`.
4. Open a PR on GitHub with a description explaining *what* and *why*.
5. Reviewers comment; you push further commits to address their concerns.
6. Once approved, the PR is merged into `main`.

PRs are the single most effective mechanism for catching bugs and spreading understanding across a team. In biostatistical projects they also double as a decision log: the PR description and comments capture the analytical reasoning behind a change, accessible years later when someone asks 'why did we use HC1 standard errors here?'.

For solo work, PRs may feel like ceremony. They are still worth using, because the diff view forces a careful read of your own changes before they land, the same hygienic function that a code review performs socially.

## 2.11. Resolving merge conflicts

Conflicts happen when two branches edit the same region of a file in incompatible ways. When `git merge` cannot reconcile them automatically, it leaves markers in the file and stops:

```
<<<<<<< HEAD
model <- lm(bp ~ age + sex, data = df)
=======
model <- lm(bp ~ age + sex + bmi, data = df)
```

```
>>>>>>> feature-add-bmi
```

The text between `<<<<<<< HEAD` and `=======` is the version on your current branch; the text between `=======` and `>>>>>>>` is the version from the incoming branch. Resolving the conflict means editing the file to the state you want (which may be either version, or a combination, or something new), removing the markers, then staging and committing:

```
# edit cleaning.R to resolve the conflict, then
git add cleaning.R
git commit
```

A few rules of thumb:

- Before resolving, understand *why* both changes exist. Each was made with some purpose in mind. The resolution should preserve both intents if possible.
- Never resolve by deleting the markers without reading. A surprisingly common failure mode is to 'resolve' by keeping only one side and silently dropping the other.
- If you are unsure, abort with `git merge --abort` and talk to whoever wrote the other version before trying again.

Conflicts are not a sign of failure. They are a sign that two people were working on the same thing, and Git is correctly refusing to guess. The alternative, silent, incorrect merging, would be worse.

## 2.12. .gitignore and R project structure

Git tracks everything in the working directory by default. This is usually too much. RStudio creates a `.Rproj.user/` directory full of internal scratch files; R writes `.Rhistory` and `.RData` if you let it; renders produce PDFs and HTMLs that should be regenerated from source rather than committed. A `.gitignore` file in the repository root lists patterns that Git will exclude from tracking.

A reasonable starting point for an R project:

```
# RStudio and R scratch
.Rproj.user/
.Rhistory
.RData
.Ruserdata

# renv package cache (lockfile IS committed; cache is not)
renv/library/
renv/local/
renv/cellar/

# rendered output (regenerate from source)
*.html
*.pdf
_book/
_site/
_freeze/

# OS detritus
.DS_Store
Thumbs.db

# data (case by case)
data/raw/
!data/raw/README.md
```

Four rules for what to ignore:

1. **User-specific scratch** (`.Rhistory`, `.RData`, `.DS_Store`). Never useful to others, pure noise in `git status`.
2. **Regenerable outputs.** If you can rebuild it from source with one command, exclude it. Commit the source, not the build product. The exception is when a rendered output is itself a deliverable and the rendering is expensive or non-deterministic.
3. **Large data.** Git is designed for source, not data. Over about 10–50 MB per file, committing data starts to hurt. Over about 100 MB, GitHub refuses. Options: store the data externally and download

in a script; use Git LFS; or commit a small synthetic sample and document access to the real data.

4. **Sensitive information.** Protected health information, passwords, API keys, data-use-agreement-covered datasets. Never. Even if you delete a sensitive file in a later commit, it persists in the history. The correct response to an accidentally-committed secret is to rotate the secret, not to rewrite history.

The standard R project layout that works well with version control:

```
project/
├── .git/                (hidden, Git internals)
├── .gitignore
├── .Rprofile            (project-local R startup)
├── renv.lock            (committed)
├── renv/                (partially committed; see above)
├── README.md            (project description)
├── project.Rproj        (RStudio project file)
├── data/
│   ├── raw/             (gitignored, with a README)
│   └── processed/       (gitignored, regenerable)
├── R/                   (reusable functions)
├── analysis/            (numbered analysis scripts)
└── output/              (figures, tables, mostly gitignored)
```

The principle behind this layout: clear separation of source (committed) from inputs and outputs (usually not), and an analysis pipeline where the source is sufficient to regenerate everything downstream.

Check your understanding: Sensitive data

**Question.** What approaches work for managing sensitive clinical or biological data within a version-controlled project?

**Answer.**

Never commit raw sensitive data to a Git repository, even a private one. Once committed, it lives in the history permanently. The usual pattern: keep a `data/` directory that is `.gitignore`d, and include a `data/README.md` (which *is* committed) documenting the secure

location of the data and the access procedures. For code testing and reviewer access, commit a small synthetic dataset in a separate `test_data/` directory that reproduces the schema of the real data without the PHI. Analysis scripts read from `data/` in production and from `test_data/` in continuous integration.

## 2.13. Commit message style and history hygiene

A brief exhortation, because this is the single practice that most clearly separates experienced users from novices.

**Imperative first line, under 50 characters.** 'Add Cox regression helper', not 'Added Cox regression helper stuff'. It should complete the phrase 'If applied, this commit will _____'.

**Blank line, then a paragraph for anything non-obvious.** Explain *why*, not *what*, the diff shows what. For statistical code, include the analytical reasoning:

```
Switch to HC1 sandwich estimators for all model SEs.

Residual diagnostics on the primary outcome model showed
clear heteroscedasticity (Breusch-Pagan p < 0.001), which
invalidates OLS SEs. HC1 is the variant used in the
preregistered analysis plan. This change affects all
downstream CI widths but not point estimates.
```

**Reference issues or PRs** where relevant: `Closes #42`, `Related to #87`. GitHub auto-links these and will close the issue when the PR merges.

**Match commit granularity to logical changes**, not to time elapsed. A commit is a unit of reasoning, not a unit of work.

Check your understanding: Commit message style

**Question.** How might your commit message style differ between a personal project and a collaborative research project?

**Answer.**
Personal projects can get away with shorter, shorthand messages that make sense to you in the moment. Collaborative research projects need messages that explain not only what changed but *why*, because other readers (including future you) lack your context. For a research project, the commit log doubles as documentation of methodological decisions — it becomes the primary source for the methods section of the paper months later. Good collaborative commit messages tend to have: an imperative first line under 50 characters, a blank line, and a paragraph of motivation referencing the analytical reason, prior discussion, or the specific bug being fixed.

## 2.14. Collaborating with an LLM on Git

Git is a domain where LLM assistance is especially useful and especially prone to silent failure. The commands are precise, the error messages are cryptic, and the consequences of running the wrong command can be irreversible. Three patterns are worth learning explicitly.

**Prompt 1: explaining the current state.** Paste the output of `git status` and `git log --oneline -10` and ask: 'summarise the state of this repository in two sentences, and tell me what I should probably do next.'

*What to watch for.* The model will typically describe the state correctly but may invent a plausible-sounding 'next step' that does not match your actual intent. If it suggests `git push`, confirm that you actually want to publish these commits. If it suggests `git reset`, stop and verify what would be lost.

*Verification.* Re-run `git status` after any action the model suggests. The state should match what the model said it would produce. If it does not, something went wrong.

**Prompt 2: reading a merge conflict.** Paste the contents of a conflicted file, including the `<<<<<<<`, `=======`, and `>>>>>>>` markers, and ask: 'explain what each side of this conflict is doing, and propose a resolution that preserves both intents if possible.'

*What to watch for.* The model is good at naming the surface-level difference (which lines are added on each side) and worse at inferring *why* each change was made. The intention behind a statistical change, 'we added the BMI covariate because the reviewer asked for it', is not visible in the diff. Bring that context yourself.

*Verification.* After resolving, run the relevant tests or sanity checks (`devtools::test()`, spot-check key figures, rerun the analysis). A conflict is 'resolved' only when the downstream outputs are what they should be, not when the markers are gone.

**Prompt 3: drafting a `.gitignore`.** Ask: 'draft a `.gitignore` for an R package project that uses renv, produces a Quarto book, and has a `data/` directory with sensitive CSVs.'

*What to watch for.* LLM-generated `.gitignore` files tend to be thorough and sometimes over-broad. Read every line. In particular, watch for patterns that would exclude files you need, `*.yaml` will exclude `_quarto.yml`, for example. Also watch for missing entries: LLMs sometimes forget `.Rproj.user/`, `renv/library/`, or OS-specific files.

*Verification.* Compare the model's output to `usethis::git_vaccinate()` and `usethis::use_git_ignore()`, which encode the community-standard defaults for R projects. Use the model's version as a draft and the `usethis` defaults as a safety check.

The meta-lesson is the same as elsewhere in this book: the LLM produces a plausible candidate quickly. You are still the one responsible for verifying that it is correct for your situation.

## 2.15. Principle in use

The statistician's contribution to a Git workflow is not the typing of commands; it is the curation of a history that a future reader, reviewer, collaborator, regulator, or yourself, can trust. Three habits do most of the work:

1. Commits are atomic and carry messages that explain the 'why'. The history reads as a sequence of deliberate, understandable decisions.

2. Branches are used for anything non-trivial, so `main` always represents a known-good state.
3. What gets committed (source) is clearly distinguished from what gets ignored (outputs, large data, secrets).

When these habits are in place, an LLM can cheerfully generate the next `.gitignore` or draft the next commit message. When they are not, no amount of tooling will save you.

## 2.16. Exercises

1. Create a new GitHub repository, clone it locally, add a single `.R` file, and push one commit. Verify that the commit appears on GitHub.
2. Introduce a merge conflict on purpose: create a branch, edit line 1 of `README.md`, commit. Switch back to `main`, edit the same line differently, commit. Merge the branch and resolve the conflict. Inspect the resulting history with `git log --graph --oneline` and describe what you see.
3. Using `git log --grep`, find every commit in a course or project repository whose message contains the word 'bootstrap'. Pick one and run `git show <hash>` to inspect the diff.
4. Write a `.gitignore` for an R project that uses `renv` and renders a Quarto book to PDF and HTML. Compare your version to the output of `usethis::use_git_ignore()`. Which entries did each version have that the other did not, and are any of those differences important?
5. Open a pull request against a repository (your own or a collaborator's). In the PR description, explain *what* the change is and *why* you are making it. Ask a colleague to review it. Note which comments, if any, you would not have caught on your own.

## 2.17. Further reading

- (Bryan, 2019), the standard reference for Git in R workflows, including the non-obvious RStudio integration.

- (Blischak et al., 2016), motivates version control for scientists in three pages.
- (Chacon & Straub, 2014), the comprehensive Git reference. Not a first-read book; a second-read-when-stuck reference.
- Learn Git Branching — an interactive visualisation of branching and merging. Recommended for visual learners.

## 2.18. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 2.18.1. Question 1

What is the primary purpose of version control?

- A) To make code run faster
- B) To track changes to files over time
- C) To automatically debug code
- D) To optimise computer memory usage

> **i** Answer
>
> **B.** Version control records a history of changes so you can review, revert, and collaborate.

### 2.18.2. Question 2

Which git command creates a new repository?

- A) `git clone`
- B) `git push`
- C) `git init`
- D) `git commit`

Answer

**C.** `git init` initialises a new empty repository in the current directory. `git clone` copies an existing remote repository.

### 2.18.3. Question 3

Which of the following is the **worst** candidate for a commit message?

- A) `Add sandwich SEs to primary model`
- B) `Fix off-by-one error in date parsing`
- C) `stuff`
- D) `Switch bootstrap from 1000 to 10000 reps`

Answer

**C.** 'stuff' conveys nothing about the change and is useless when reviewing history or bisecting. The other three are all imperative, specific, and informative.

### 2.18.4. Question 4

A colleague accidentally commits a CSV containing protected health information to a private GitHub repository. What is the correct response?

- A) Delete the file in a new commit.
- B) Run `git reset --hard` to the previous commit.
- C) Consider the data exposed, notify your data-use compliance officer, and rotate any affected access.
- D) Rewrite history with `git filter-branch` and continue.

Answer

**C.** Once committed, the file exists in the history. Even aggressive history rewriting does not guarantee the file is gone from all clones, forks, or GitHub caches. Treat it as a disclosure incident and handle

it through the appropriate compliance channel.

## 2.19. Prerequisites answers

1. Version control tracks changes to files over time, enabling you to review history, revert to earlier states, and collaborate without overwriting each other's work. Unlike dated backup copies, Git stores *differences* between versions (not full copies), records metadata for each change (author, time, message), and supports workflows, branching, merging, review, that file backups cannot.
2. `git init` creates a new, empty repository by adding a `.git/` directory to the current folder, making future edits to that folder trackable. Use `git init` when you have local files and want to start tracking them; use `git clone` when you want a local copy of an existing remote repository.
3. `git add` stages changes by recording a snapshot of the working tree for the next commit. `git commit` permanently records the staged snapshot in the repository's history, with an author and a message. Git separates these operations so you can build up a logically coherent commit from only a subset of the changes in your working directory, rather than being forced to commit everything modified at once.

# 3. R Internals

## 3.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 3.19.

1. Given `x <- list(a = 1, b = 2, c = 3)`, what is the type and value returned by `x[["b"]]`, and how does it differ from what is returned by `x["b"]`?
2. What is the primary structural difference between a data frame and a matrix in R, and when does the difference matter?
3. When you modify a single element of a long vector that another variable name also binds to, what does R do internally, and why does this have performance consequences?

## 3.2. Learning objectives

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

- Describe R's memory model in terms of names and values, and explain what 'copy-on-modify' means.
- Use `lobstr::obj_addr()` and `tracemem()` to verify whether a modification triggers a copy.
- Predict and diagnose the $O(n^2)$ cost of growing a vector in a loop; rewrite such code as $O(n)$ with pre-allocation or $O(1)$ with vectorisation.
- Explain lists as recursive vectors, and the semantic distinction between `[`, `[[`, and `$`.
- Explain data frames as lists of columns, and the performance consequences for column-wise vs. row-wise work.

- Use `bench::mark()` to compare implementations rigorously, and `Rprof()`/`summaryRprof()` (or `profvis`) to locate bottlenecks.

## 3.3. Orientation

Writing fast R code requires knowing where your R code is slow, and that requires knowing a little about how R represents objects in memory. This chapter is the minimum you need to reason about performance without leaving R.

It is also the bridge between 'R works' and 'R works efficiently'. Every practising R user has written code that felt inexplicably slow. Most of those incidents trace back to a small number of misunderstandings about R's memory model. The payoff for internalising these ideas is disproportionate: 100× speedups on real analyses are common.

## 3.4. The statistician's contribution

Memory semantics are objective facts about R. An LLM can summarise them in a paragraph. What an LLM cannot do is make the judgement calls that separate working code from defensible analysis code.

**When to optimise, and when not to.** The dominant cost in most statistical analyses is not R's execution time; it is the analyst's time to reason about the model, check results, and communicate findings. Premature optimisation, rewriting clear code into obscure code to save milliseconds, is a common self-inflicted wound. The statistician's judgement is to optimise only the parts of a pipeline that are actually a bottleneck, and to preserve readability everywhere else. This is what profiling is for: it tells you where to spend the effort.

**When to leave R.** Some statistical problems genuinely need more than R's interpreted loops can deliver. Rcpp, data.table, and (for pure-numeric work) vectorised BLAS calls are the standard escape hatches. The question is not 'is this faster?' — C++ is almost always faster, but 'is the speedup worth the cost in code clarity, testability, and maintainability?' For a function

run once per analysis, probably not. For a function inside an MCMC or a bootstrap loop that runs millions of times, probably yes.

**What to trust.** Benchmarks are noisy. A `system.time()` call that runs once can mislead you by 2× either direction for reasons that have nothing to do with your code. Treat a single timing as a rough signal; treat `bench::mark()` output as a measurement. When the stakes are high, a paper's central claim about computational feasibility, or a regulatory submission's runtime, run benchmarks on a quiet machine, repeat them, and report variability.

**Which data structure for which job.** Matrices and data frames look interchangeable until they are not. A simulation that produces a million numeric values per iteration should return a matrix, not a tibble. A downstream analysis that merges, filters, and groups should use a data frame or tibble, not a matrix. The choice is a judgement about what dominates: numeric throughput or ergonomics. Getting it right is worth more than any micro-optimisation.

These decisions shape whether an analysis is fast enough to be practical, readable enough to be reviewed, and correct enough to be trusted. None of them can be automated.

## 3.5. R's memory model: names and values

Every R object has two components: the **value** (the actual data) and the **name** or names that bind to that value. Multiple names can point to the same value, and R does not copy the underlying data until a modification makes the copy necessary. This is 'copy-on-modify'.

```
library(lobstr)

x <- c(1, 2, 3, 4, 5)
y <- x              # y and x now point to the SAME vector

obj_addr(x)
#> [1] "0x7fd7..."
```

```
obj_addr(y)
#> [1] "0x7fd7..."  (identical)
```

The `obj_addr()` function returns the hex address where R has stored the object. After `y <- x`, both names point to the same address. No copying has occurred.

Copy-on-modify kicks in when you try to change a shared object:

```
y[1] <- 99          # modification triggers a copy

obj_addr(x)
#> [1] "0x7fd7..."          (unchanged)
obj_addr(y)
#> [1] "0x7fe9..."          (new address)
```

R sees that another name (`x`) still refers to the original vector, so it makes a copy, modifies the copy, and points `y` at the copy. `x` is untouched.

This design trades some performance for a lot of safety. Without copy-on-modify, assigning `y <- x` and then modifying `y` could silently change `x`, as happens in Python with mutable objects. R chose predictability. For most statistical work, this is the right trade-off.

Check your understanding: Why memory matters

**Question.** Why does understanding R's memory model matter for statistical programming?
**Answer.**
Memory knowledge is what separates code that feels fast from code that feels sluggish for reasons you cannot explain. The copy-on-modify rule is the direct cause of the classic growing-vector-in-a-loop trap ($O(n^2)$ behaviour). The modify-in-place exception for single-binding objects explains why pre-allocation rescues that same loop. Understanding when R copies and when it does not lets you write functions that run dozens of times faster on identical inputs. Ignorance of these rules costs 10–100× speedups on routine analysis code, silently.

## 3.6. Modify-in-place: the single-binding exception

When an object has only one binding, R modifies it in place. No copy is made. This is a crucial exception that underlies most R performance advice.

```
x <- c(1, 2, 3, 4, 5)
obj_addr(x)
#> [1] "0x7fa1..."

x[1] <- 99          # only one name; R modifies in place
obj_addr(x)
#> [1] "0x7fa1..."         (same address)
```

The address is unchanged. R's reference counting saw that `x` was the only binding, concluded that it was safe to modify the existing memory, and did so.

This is what makes pre-allocation effective: once you allocate a result vector and assign it to a single name, further writes into that vector modify it in place. The cost of each write is O(1), and the loop is O(n).

Check your understanding: Without copy-on-modify

**Question.** How would code behave differently if R did not use copy-on-modify semantics?
**Answer.**
Without copy-on-modify, `y <- x` would make `y` an alias for `x`. Modifying `y[1] <- 10` would also change `x[1]`. This is how Python lists behave. It would be faster (no copying) but error-prone: any function that modifies one of its arguments would have effects that ripple outward in ways that are hard to reason about. For interactive statistical work, where reproducibility and readability matter more than raw speed, R chose safety. Environments and some specialised packages (e.g., data.table) opt out of copy-on-modify when the speed gain is worth the added cognitive load.

## 3.7. Growing vectors: the quadratic trap

The single most common self-inflicted performance wound in R code is growing a vector inside a loop:

```
# SLOW: O(n^2)
result <- c()
for (i in 1:50000) {
  result <- c(result, i^2)
}
```

Each iteration copies the entire current vector, appends one element, and re-binds `result` to the new, longer vector. On iteration `i` the vector has size `i - 1`, so the copy costs `i - 1` operations. Summing over `i = 1, 2, ..., n` gives `n(n+1)/2 ≈ O(n²)`.

On a laptop, `n = 50000` takes several seconds. On `n = 500000`, the naive loop is unreasonable. This is not because R is slow; it is because the algorithm is accidentally quadratic.

Three rewrites, in increasing order of goodness:

```
# OK: pre-allocated, O(n)
result <- numeric(50000)
for (i in seq_len(50000)) {
  result[i] <- i^2
}

# Better: vectorised, effectively O(1) in interpreter calls
result <- seq_len(50000)^2
```

The pre-allocated version works because `result` has a single binding after allocation; each assignment modifies in place. The vectorised version pushes the loop into C code inside R's internals, which is both algorithmically better and avoids the interpreter overhead.

Concretely, on typical hardware:

- Grown vector: ~8 seconds.

- Pre-allocated: ~0.05 seconds (160× faster).
- Vectorised: ~0.0005 seconds (10,000× faster than grown).

This is not a pathological example. It is exactly the shape of code many people write when first learning R. The diagnosis is easy once you know what to look for: if a loop ends with `x <- c(x, new_value)` or `x <- rbind(x, new_row)`, you have the quadratic trap.

## 3.8. Lists as recursive vectors

Atomic vectors hold values of a single type, stored contiguously in memory. Lists are different: each element can hold any R object, including another list, and the elements are stored as independent objects with the list holding pointers to them.

```
my_list <- list(
  numbers = 1:5,
  label   = "Alice",
  model   = list(method = "Cox", df = 3)
)
```

This is the recursive part: a list can contain a list can contain a list. Almost every composite object in R is ultimately built on lists: data frames, model outputs, S3/S4/R6 objects, function environments. Learning to think about lists fluently pays enormous dividends.

### 3.8.1. `[` vs `[[` vs `$`

This is the most frequently confused piece of R syntax. The distinction is simple once stated plainly:

- `[` returns a subset *of the container type*. For a list, it returns a list. For a vector, it returns a vector. This is called 'preserving'.
- `[[` extracts *one element, at its own type*. For a list, it returns whatever that element is (a vector, a number, a nested list). This is called 'simplifying'.

- `$` is a shorthand for `[[` by name: `my_list$numbers` is exactly `my_list[["numbers"]]`.

```
my_list[1]       # list of length 1, containing 1:5
my_list[[1]]     # 1:5 itself
my_list$numbers  # 1:5 itself
```

Using `[` when you meant `[[` is a subtle bug generator: downstream code expects a vector but receives a length-1 list, and either fails or silently coerces.

The old mnemonic (Wickham): if a list is a train, `[` returns a smaller train, `[[` returns the contents of one car, and `$` returns the contents of the car with the matching label.

Check your understanding: `[` vs `[[`

**Question.** What is the difference between `my_list[1]` and `my_list[[1]]`?
**Answer.**
`my_list[1]` returns a list containing one element, the first element of `my_list` wrapped in a length-1 list. `my_list[[1]]` returns the element itself, at its own type: a vector, a scalar, a nested list, whatever was stored there. If downstream code expects a vector and you pass `my_list[1]`, it will fail or silently coerce. The `$` operator is a shorthand for `[[` by name. Mastery of this distinction prevents a lot of hours of debugging.

## 3.9. Data frames are lists of columns

A data frame is, internally, a list whose elements are equal-length vectors. Each column is a separate vector stored at its own address; the data frame is a list of pointers to those vectors.

This has three important consequences.

**Column-wise operations are fast.** `df$x` is just retrieving a vector from the list, constant-time, no copying. Arithmetic on a column is a plain vector operation.

**Row-wise operations are slow.** To take row 3, R must pull element 3 from each column vector and assemble them into a new structure. For a data frame with 50 columns, that is 50 pointer dereferences and 50 small copies per row. Running `for (i in seq_len(nrow(df)))` is almost always the wrong approach for more than a few hundred rows. Vectorise along columns, or use `dplyr::group_by()` / `data.table` idioms that are column-oriented under the hood.

**Subsetting preserves the list structure.** `df[1]` returns a data frame with one column (the preserving `[`). `df[[1]]` returns the vector of the first column (the simplifying `[[`). `df$name` returns the column. This is exactly the list semantics from the previous section, applied to a particular kind of list.

Matrices are different: they store all elements as one contiguous vector with a `dim` attribute. This is why:

- Matrix multiplication (`%*%`), decompositions (`solve`, `qr`), and many linear algebra operations are fast on matrices and absent or slow on data frames.
- Matrices require all elements to share a type. Mixing numeric and character columns forces everything to character, usually silently and usually wrong.

Check your understanding: Matrix vs. data frame

**Question.** When should you convert a data frame to a matrix for analysis?
**Answer.**
Convert when: (1) every column is numeric, (2) you are about to do linear algebra, `%*%`, `solve()`, `qr()`, `svd()`, or pass the data to a function that expects a matrix, and (3) the dataset is large enough that the memory-contiguity advantage matters. The conversion itself is cheap. Keep it as a data frame when types are mixed, when you want dplyr/tidyr ergonomics, or when you will display or export it.

> The common mistake is either extreme: keeping numeric-only data in a tibble through an expensive simulation loop, or coercing mixed-type data to a matrix and silently corrupting the character columns.

## 3.10. Factors: efficient categorical storage

A factor stores a categorical variable as integer codes with a separate `levels` attribute mapping codes to labels.

```
f <- factor(c("high", "low", "high", "medium"))
typeof(f)
#> [1] "integer"
as.integer(f)
#> [1] 1 2 1 3
levels(f)
#> [1] "high"   "low"    "medium"
```

For 1,000,000 observations of a two-level variable, storing integers is vastly cheaper than storing 1,000,000 copies of the strings. Factors also make grouping operations (`tapply`, `split`, `dplyr::group_by`) fast: the engine groups on the integer codes, not on string equality.

The classic pitfall: `as.numeric(f)` returns the integer codes, not the original numeric values (if the labels were numeric-looking). For `factor(c("10", "20", "30"))`, `as.numeric(f)` gives `1 2 3`, not `10 20 30`. The correct idiom is `as.numeric(as.character(f))` or `as.numeric(levels(f))[f]`. This trap catches working statisticians frequently enough that it deserves naming.

## 3.11. Benchmarking: `bench::mark()`

`system.time()` measures elapsed time for a single evaluation. It is a useful rough gauge, ‘does this take 0.1 seconds or 10 seconds?’, but noisy. For comparing implementations, `bench::mark()` is the standard.

```
library(bench)

grow <- function(n) {
  result <- c()
  for (i in seq_len(n)) result <- c(result, i^2)
  result
}

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

vectorise <- function(n) seq_len(n)^2

bench::mark(
  grow(5000),
  preallocate(5000),
  vectorise(5000),
  check = TRUE
)
```

`bench::mark()` runs each expression multiple times and reports median, min, max, allocations, and throughput. The `check = TRUE` argument (the default) verifies that every expression returns the same value, which catches bugs where an 'optimised' version silently produces different output.

For honest results, run benchmarks on a quiet machine. Background applications, file indexing, and active browsers all add noise. When results differ by less than 2×, suspect noise; when they differ by 10× or more, suspect a real algorithmic difference.

## 3.12. Profiling: `Rprof()` and `profvis`

Benchmarking tells you how fast *one piece of code* is. Profiling tells you *where* a larger program is spending its time. You cannot optimise effectively

without profiling, because you cannot guess where bottlenecks are.

`Rprof()` samples the call stack at regular intervals and writes the samples to a file. `summaryRprof()` aggregates the results:

```
Rprof("profile.out")
result <- my_analysis()
Rprof(NULL)
summaryRprof("profile.out")$by.self
```

The `by.self` column shows time spent in each function excluding the time it spent calling other functions. The functions at the top of that list are where to focus optimisation effort.

The `profvis` package provides an interactive visualisation of the same data:

```
library(profvis)
profvis({
  result <- my_analysis()
})
```

`profvis` opens an HTML widget in RStudio with a flame graph and line-by-line timing. It is usually the fastest way to find the bottleneck in a function you are unfamiliar with.

The Pareto rule applies strongly to performance: typically 20% of the code accounts for 80% of the runtime. Optimising outside the bottleneck wastes time. Profiling turns that principle into action.

## 3.13. Reference material: environments

Environments are the one major exception to copy-on-modify. When you assign `env2 <- env1` (where both are environments), both names point to the same environment; modifying one modifies the other.

```
e1 <- new.env()
e1$x <- 10
e2 <- e1
e2$x <- 99
e1$x
#> [1] 99       # e1 sees the change made through e2
```

Environments are used deliberately when mutable state is needed: package namespaces, closures, R6 classes, `data.table` internals. For day-to-day statistical programming, the practical consequence is a caveat: if you pass an environment into a function and the function modifies it, the modification is visible outside the function. For all other R objects, function arguments are effectively pass-by-value (because any modification triggers a copy).

## 3.14. Collaborating with an LLM on R internals

Memory semantics are an area where LLMs are often helpful, sometimes subtly wrong, and occasionally confidently wrong in ways that are dangerous. Three patterns work well.

**Prompt 1: explaining observed behaviour.** Paste a short script showing surprising memory or timing behaviour (addresses from `obj_addr()`, timings from `system.time()`) and ask: 'what is R doing here, and why?'

*What to watch for.* LLMs are generally correct about copy-on-modify at the level of explanation, but they sometimes invent plausible-sounding intermediate details (e.g., specific R internal function names that don't exist). Treat the high-level explanation as a starting point; verify the low-level claims against `?lobstr::obj_addr` and Wickham's *Advanced R*, chapters 2–5.

*Verification.* Reproduce the behaviour in an R session. If the LLM predicts 'this operation triggers a copy', check with `tracemem()` or `obj_addr()` before and after. Predictions that match observation stay; predictions that don't get corrected.

**Prompt 2: rewriting a slow function.** Paste the function, any benchmark output, and ask the LLM to rewrite it for speed. Ask it to explain the rewrite.

*What to watch for.* The rewrite will usually be faster. The explanation may or may not be accurate; LLMs tend to cite 'vectorisation' or 'pre-allocation' even when the actual speedup comes from something else (e.g., switching to a BLAS matrix op, or avoiding a repeated lookup). Do not copy the explanation into a methods section without verifying it against profiling evidence.

*Verification.* Benchmark both versions with `bench::mark(..., check = TRUE)`. The `check = TRUE` argument ensures the rewritten function returns the same value. If the rewritten version is a lot faster, profile it to confirm the claimed reason.

**Prompt 3: Rcpp implementation.** For a tight numerical loop inside a simulation or MCMC, ask the LLM to write the same loop in C++ with Rcpp.

*What to watch for.* Rcpp code compiled incorrectly still runs and still returns plausible numbers, a silent-wrong scenario that is much more dangerous than an R error. Edge cases (empty input, `NA`, infinity, integer overflow) are common sources of silent miscalculation in LLM-generated C++. Also watch for index-off-by-one: R is 1-indexed, C++ is 0-indexed, and translations between them are error-prone.

*Verification.* Write a test suite comparing the Rcpp output to the pure-R version on a battery of inputs, including edge cases (zero-length input, `NA`, values near machine precision). Only trust the Rcpp version after it matches R on every test case and the speedup is confirmed on realistic input sizes.

The meta-pattern: for R internals work, an LLM is a good research assistant and a poor authority. It can generate a candidate answer quickly; you are responsible for turning that candidate into a trustworthy one.

## 3.15. Principle in use

Three habits define effective use of R's memory semantics:

1. **Measure before optimising.** Profile first to find the bottleneck. Benchmark before and after to verify the change actually helped.
2. **Pre-allocate or vectorise by default.** The quadratic trap is avoidable once you recognise its shape. Making pre-allocation automatic saves you from ever paying its cost.
3. **Match data structure to workload.** Matrix for dense numeric computation; data frame for mixed-type, column-oriented analysis. Convert between them as the analysis moves between phases.

Internalise these three habits and most R performance problems become non-problems. The deeper memory model — copy-on-modify, reference counting, environments, is the explanation for *why* the habits work, and the material you reach for when the habits are not enough.

## 3.16. Exercises

1. Use `bench::mark()` to compare three implementations of a running mean over a vector of length 10^6: a `for` loop that grows a vector, a `for` loop that pre-allocates, and `cumsum(x) / seq_along(x)`. Which is fastest, and why?
2. Using `lobstr::obj_size()`, measure the size of a list of 1,000 numeric vectors of length 1,000 versus a single numeric vector of length 10^6. Explain the difference.
3. Profile the body of your favourite function from chapter 1 with `profvis`. Identify the line that accounts for the most self-time. Does it match where you would have guessed the bottleneck was?
4. Write a function that simulates a random walk of length `n`. Implement it three ways: growing a vector, with pre-allocation, and with `cumsum(rnorm(n))`. Benchmark all three for `n = 10^3`, `10^4`, `10^5`.
5. Create a factor `f <- factor(c("10", "20", "30"))`. Show what `as.numeric(f)` returns, and then compute the correct numeric vector from `f`. Explain to yourself why `as.numeric(f)` behaves the way it does.

## 3.17. Further reading

- (Wickham, 2019) Chapters 2–5, names, values, copy-on-modify, and function environments. The canonical reference.
- (Dowle & Srinivasan, 2021), `data.table`'s design illustrates the performance implications of opting out of copy-on-modify.
- `?bench::mark` and `?profvis::profvis`, the tool documentation is excellent, short, and includes runnable examples.

## 3.18. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 3.18.1. Question 1

What does the following R code display?

```
x <- list(a = 1, b = 2, c = 3)
x[["b"]]
```

- A) The entire list
- B) A list containing only element 'b'
- C) The value 2
- D) NULL

> **Answer**
>
> **C.** `[[` extracts the element at its own type, here the scalar `2`. `x["b"]` would return a list of length 1.

### 3.18.2. Question 2

What is the primary difference between a data frame and a matrix in R?

- A) Matrices can only contain numbers, while data frames can contain different types in each column
- B) Data frames can only have row names, while matrices can have both row and column names
- C) Matrices can have any number of dimensions, while data frames are limited to 2 dimensions
- D) Data frames must have unique column names, while matrices cannot have named columns

**i** Answer

**A.** A matrix is a single atomic vector with dimensions (uniform type throughout). A data frame is a list of equal-length vectors, so each column can carry a different type.

### 3.18.3. Question 3

In R, what is the result of the following code?

```
x <- 1:3
names(x) <- c("a", "b", "c")
x["b"]
```

- A) 1
- B) 2
- C) 'b'
- D) An error because you can't name a numeric vector

**i** Answer

**B.** Numeric vectors can carry names; subsetting by name returns the element with the matching name, preserving the name as well as the value.

### 3.18.4. Question 4

Consider:

```
x <- c(1, 2, 3, 4, 5)
y <- x
y[1] <- 99
```

After this code runs, what are `x` and `y`?

- A) `x = c(99, 2, 3, 4, 5)`, `y = c(99, 2, 3, 4, 5)`
- B) `x = c(1, 2, 3, 4, 5)`, `y = c(99, 2, 3, 4, 5)`
- C) Both are `c(99, 2, 3, 4, 5)` because R uses reference semantics.
- D) An error: you cannot modify a vector in place.

> **Answer**
>
> **B.** Copy-on-modify. Assigning `y <- x` makes both names point to the same underlying vector. Modifying `y[1]` triggers a copy, so `y` gets a new modified vector while `x` continues pointing to the original.

### 3.18.5. Question 5

Which of the following is an example of the $O(n^2)$ 'growing vector' trap?

- A) `result <- numeric(n); for (i in seq_len(n)) result[i] <- f(i)`
- B) `result <- purrr::map_dbl(seq_len(n), f)`
- C) `result <- c(); for (i in seq_len(n)) result <- c(result, f(i))`
- D) `result <- f(seq_len(n))`

> **Answer**
>
> **C.** Repeatedly using `c(result, new_value)` copies the full current vector at every iteration. Total work is $n(n+1)/2 = O(n^2)$. Options

A, B, and D are all O(n) or better.

## 3.19. Prerequisites answers

1. `x[["b"]]` extracts the element at key `'b'` and returns it at its own type, here the numeric scalar `2`. `x["b"]` returns a *list* of length one containing that element. The difference matters whenever the next operation expects a scalar (or a vector) and cannot cope with a list wrapper.
2. A matrix is a single atomic vector with dimensions: every element has the same type, and values are stored contiguously in memory. A data frame is a list of equal-length vectors, so each column can have a different type. The difference matters whenever a column is character, factor, or logical (forcing a matrix would coerce the whole thing), and whenever performance of linear algebra matters (matrices are much faster for `%*%`, `solve()`, decompositions).
3. R uses copy-on-modify. When another name references the same vector, R creates a copy, modifies the copy, and re-binds the assigned name to the copy, leaving the other name pointing at the unchanged original. The performance consequence: if the vector is long and the operation occurs inside a loop (as in growing a vector with `c()`), the total work becomes $O(n^2)$ rather than $O(n)$. The fix is pre-allocation (so there is only one binding, triggering modify-in-place) or vectorisation.

# 4. Functional Programming

## 4.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 4.20.

1. What is the primary purpose of the `purrr` package? In what situations would you reach for it rather than base R's `lapply`/`sapply`?
2. Among `map()`, `map_dbl()`, `map_chr()`, and `map_lgl()`, how do they differ in return type, and what happens if an element of the result does not match the declared type?
3. In `map(1:5, ~ .x^2)`, what does the tilde (`~`) represent, and how would you rewrite this using the `\()` anonymous function syntax?

## 4.2. Learning objectives

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

- Explain why functions are first-class objects in R, and what 'passing a function as an argument' buys you.
- Write anonymous functions with `\(x) ...` and with the `~ .x` shortcut, and know when each is appropriate.
- Replace a `for` loop with the appropriate `purrr::map_*()` variant, choosing the typed version when you know the expected return type.
- Use `map2()` and `pmap()` for iteration over two or more parallel vectors (as in simulation studies).
- Use `purrr::reduce()` and `purrr::accumulate()` to collapse a list into a single value.
- Apply `purrr::safely()` and `purrr::possibly()` to iterate robustly over messy input that includes failures.

## 4.3. Orientation

A recurring question in applied work is: how do I apply the same operation across many elements of a data structure? Write the computation once as a function, then pass it to an iteration tool that takes care of the mechanics. Once you can write functions, you can write functions that take functions as arguments, and that is where the leverage comes from.

R is fundamentally a functional language. Almost everything you do in R can be expressed as a composition of functions applied to data. The two common iteration toolkits are base R's `apply` family (`lapply`, `sapply`, `vapply`, `tapply`, `apply`) and the tidyverse's `purrr`. Both do the same core job. `purrr` does it with more consistent naming and type guarantees; `apply` is ubiquitous and has no dependencies. This chapter covers both and recommends `purrr` for most work.

## 4.4. The statistician's contribution

`map`, `lapply`, and friends are boring mechanics. Anyone can look them up. The interesting questions are about what you iterate over, what the function returns, and what you do when things go wrong. These shape whether an analysis scales, whether failures are loud or silent, and whether the code is honest about what it computed.

**What does the iteration unit represent?** A bootstrap resample, a patient stratum, a simulation parameter combination, a file of raw data. The iteration unit is a statistical object, not just a loop index. Naming it well in code, `resample`, `stratum`, `params`, `file_path`, is worth the minute it takes, because the same code will be read dozens of times over a project's life.

**What type should the function return?** A single number, a logical, a character, a tibble with one row per group, a fitted model object. The statistician's judgement is to pick the type that makes downstream work easy, not the type that was convenient to produce. If every iteration returns a tibble, `map()` followed by `list_rbind()` gives a tidy result in one step. If iterations return mixed types, the downstream code has to re-sort the zoo.

**What happens when an iteration fails?** Raw iteration aborts on the first failure and returns nothing. For a simulation study with 10,000 parameter combinations, one failing combination should not destroy the other 9,999 results. `safely()` and `possibly()` are the right tools here, but they force a judgement: do you want to record the error and continue (so the failure is visible later), substitute a default value (so the downstream code does not need to special-case missing results), or abort (because the failure indicates a bug in the computation)? Choosing well separates a defensible analysis from one that either silently drops half the data or aborts mysteriously on a corner case.

**When to resist the refactor.** A straightforward `for` loop with a clear loop body is perfectly acceptable R code. The right reason to replace it with `map_*()` is that the functional version is clearer or safer, not 'because loops are bad'. Dogmatic vectorisation that compresses a readable loop into an unreadable chain of `|>`-threaded anonymous functions is a cost without a benefit.

These are judgement calls. They do not have right answers that an LLM can look up. They are what makes functional code legibly statistical rather than showily clever.

## 4.5. Functions as objects

In R, functions are ordinary objects. You can assign a function to a variable, store it in a list, pass it as an argument, and return it from another function. This is what 'functional programming' means in practice.

```
square <- function(x) x^2
square(5)
#> [1] 25

# store functions in a list
funs <- list(sq = square, cb = function(x) x^3)
funs$sq(5)
#> [1] 25
funs$cb(2)
#> [1] 8
```

This flexibility underlies every iteration pattern in this chapter. `map(x, f)` works because `f` is just another object that can be named and passed.

## 4.6. Anonymous functions: three ways

For short functions you use only once, defining them inline avoids cluttering your namespace with single-use names. R offers three forms; they are equivalent for most purposes.

```
# classic anonymous function (verbose)
map_dbl(my_list, function(x) mean(x, na.rm = TRUE))

# R 4.1+ lambda shorthand (recommended in base R)
map_dbl(my_list, \(x) mean(x, na.rm = TRUE))

# purrr tilde shorthand (widely used in tidyverse code)
map_dbl(my_list, ~ mean(.x, na.rm = TRUE))
```

The `\(x) ...` form is the modern recommendation and is easier to read when the argument name matters (`\(patient)`, `\(fold)`, `\(params)`). The `~ .x` form is more compact for single-line expressions and is ubiquitous in older tidyverse code. Neither is wrong; consistency within a file matters more than which one you pick.

`.x` is only meaningful inside `~` formulas. Inside `\(x)`, the argument is whatever you named it. This is a small gotcha that catches people moving code between the two forms.

## 4.7. The base R apply family

The base R iteration tools predate `purrr` and remain widely used. A quick tour:

- **`lapply(X, FUN)`** applies `FUN` to each element of `X` and always returns a list. Predictable but often inconvenient when you wanted a vector.

- **sapply(X, FUN)** is lapply with an attempt to 'simplify' the result into a vector or matrix when possible. Convenient interactively, dangerous in scripts: its return type depends on the input (empty list returns list(), not a numeric vector). Avoid in production code.
- **vapply(X, FUN, FUN.VALUE)** is the safe alternative to sapply. You declare the expected return type (numeric(1), character(1)) and vapply errors if the function returns something else. This is the recommended base R idiom for scripts.
- **tapply(X, INDEX, FUN)** applies FUN to subgroups defined by the factor INDEX. Conceptually similar to group_by |> summarise in dplyr.
- **apply(X, MARGIN, FUN)** works on matrices and arrays. MARGIN = 1 iterates over rows, MARGIN = 2 iterates over columns. Note that apply on a data frame coerces to a matrix first, which is usually not what you want.

```r
my_list <- list(a = 1:10, b = 11:20, c = 21:30)

lapply(my_list, mean)              # list of three means
sapply(my_list, mean)              # named numeric vector
vapply(my_list, mean, numeric(1))  # same, type-checked

mat <- matrix(1:12, nrow = 3)
apply(mat, 2, sum)                 # column sums
apply(mat, 1, sum)                 # row sums
```

The biggest practical wart in the apply family is sapply. Its return type depends on the run-time shape of the result. If every element happens to be a length-1 numeric, you get a vector. If they vary in length, you get a matrix or a list, sometimes silently. In a script that runs on new data, this difference between runs can produce errors far from the sapply call.

## 4.8. The purrr toolkit

purrr reorganises the same operations under a consistent naming scheme. The core principle: **the function name tells you the output type.**

```
library(purrr)

# untyped map: always returns a list
map(my_list, mean)

# typed variants
map_dbl(my_list, mean)    # numeric vector
map_int(my_list, length)  # integer vector
map_chr(my_list, ~ paste("mean:", round(mean(.x), 2)))
map_lgl(my_list, ~ all(.x > 5))
```

The typed variants check the return type element-by-element and error immediately if something is wrong. This is the main reason to prefer them over `sapply`:

```
# errors immediately: "result must be length 1"
map_dbl(list(1:3, "a"), mean)
```

The same call with `sapply` would silently coerce or return a list, producing a downstream bug that is hard to trace.

Rule of thumb: if you know what type you expect, use the typed variant. Use `map()` only when you genuinely need a list back (for example, when each iteration produces a fitted model object).

Check your understanding: typed `map` vs. `sapply`

**Question.** Why is `map_dbl()` preferred over `sapply()` in scripts and packages?
**Answer.**
`sapply()`'s return type depends on the run-time shape of the output. For a normal case it may return a numeric vector; for an edge case (empty input, one element returning a different length, mixed types) it silently returns a list, a matrix, or something else. Downstream code that expected a vector then fails or silently coerces. `map_dbl()` declares the expected output type up front: each element must be a length-1 numeric, or the call errors loudly at the point of failure. Loud errors at the source are cheaper to debug than silent type changes

that surface later.

## 4.9. map2() and pmap(): parallel iteration

`map()` iterates over one input. Real analyses often iterate over parallel inputs: a vector of sample sizes and a matching vector of means, a tibble of simulation parameters with one row per combination.

```
# map2: two parallel inputs
means <- c(0, 5, 10)
sds   <- c(1, 2, 3)

samples <- map2(means, sds, \(m, s) rnorm(100, m, s))
length(samples)
#> [1] 3
```

`map2` passes element `i` of each input to the function, iterating in lockstep. Typed variants exist: `map2_dbl`, `map2_chr`, `map2_lgl`, `map2_int`.

For three or more parallel inputs, use `pmap()`:

```
params <- list(
  n    = c(100, 200, 300),
  mean = c(0,   5,  10),
  sd   = c(1,   2,   3)
)

samples <- pmap(params, \(n, mean, sd) rnorm(n, mean, sd))
```

`pmap` matches arguments by name when the input list is named (which makes the call site readable), or by position otherwise. The input list is conceptually a tibble: each element is a column, each row is one parameter combination.

For simulation studies, `pmap` on a tibble of parameters is the canonical pattern. Each row is one simulation scenario; each column is a tuning parameter.

## 4.10. `imap()`: iteration with an index or name

`imap(.x, .f)` passes both the value and its name (or index, if unnamed) to the function:

```
imap_chr(my_list, \(x, nm) paste(nm, ":", mean(x)))
#>          a          b          c
#>   "a : 5.5" "b : 15.5" "c : 25.5"
```

Useful when the iteration needs to know which element it is processing, for example, labelling plots or files by group name.

## 4.11. `walk()` for side effects

`map()` and its typed variants are for iteration that produces a value. Iteration that exists only for its side effect, printing, saving a file, drawing a plot, is cleaner with `walk()`:

```
walk2(
  plots,
  paste0("plot_", names(plots), ".png"),
  \(p, path) ggsave(path, p, width = 6, height = 4)
)
```

`walk()` returns its input invisibly, so the call does not clutter the console when used interactively, and so it composes nicely inside a pipe.

## 4.12. `reduce()` and `accumulate()`

`reduce()` collapses a list into a single value by repeatedly applying a binary function:

```
reduce(1:5, `+`)
#> [1] 15        (equivalent to 1 + 2 + 3 + 4 + 5)

reduce(list(c(1,2), c(2,3), c(3,4)), union)
#> [1] 1 2 3 4
```

`reduce2()` takes two parallel lists. `accumulate()` is like `reduce()` but returns all the intermediate values, not just the final one, useful for running totals, cumulative products, and anything else whose history you want to keep.

For most statistical work, `reduce()` is used for combining results: merging a list of data frames into one, unioning a list of sets, collapsing a list of summary statistics into an aggregate.

## 4.13. Error handling: `safely()` and `possibly()`

Iteration that aborts on the first failure is fragile. In a simulation with 10,000 runs, one failing run should not destroy the other 9,999. `purrr` offers two wrappers that convert a function into one that handles its own errors.

**`safely(f)`** returns a function that always returns a list with two components: `result` (the output, or `NULL` on failure) and `error` (`NULL` on success, or the error object on failure).

```
safe_log <- safely(log)

x <- list(1, 2, "a", 4)
results <- map(x, safe_log)

# separate successes from failures
map_lgl(results, \(r) is.null(r$error))
#> [1]  TRUE  TRUE FALSE  TRUE

# just the values for successful cases
map(results, "result") |> compact()
```

**`possibly(f, otherwise = NA)`** returns a function that returns `otherwise` on error instead of aborting. Simpler than `safely` when you do not need the error object:

```
maybe_log <- possibly(log, otherwise = NA_real_)
map_dbl(x, maybe_log)
#> [1] 0.0000000 0.6931472        NA 1.3862944
```

The judgement call: `safely()` preserves information about why each iteration failed, at the cost of a more complex return structure. `possibly()` substitutes a default value, losing diagnostic information but producing a clean numeric vector. For one-off exploratory work, `possibly()` often suffices. For a production simulation where failures carry information, use `safely()` and inspect the errors.

Check your understanding: `safely` vs. `possibly`

**Question.** When would you use `safely()` rather than `possibly()`?
**Answer.**
Use `safely()` when the error messages matter, when you need to inspect later which iterations failed and why, or when the pattern of failures is itself informative about the data (e.g., 'every run with n > 10,000 ran out of memory'). Use `possibly()` when you know the fallback behaviour is appropriate and you do not need the diagnostic information. `possibly()` returns a clean vector of the same type as the successful cases; `safely()` returns a list of success/error pairs that must be post-processed. For polished, reproducible simulation code, `safely()` is usually the right choice: the small extra complexity is worth the diagnostic richness.

## 4.14. Worked example: fitting multiple models

The 'fit the same model to each stratum' pattern is a daily occurrence in biostatistics. `purrr` makes it a one-liner.

```
library(dplyr)
library(purrr)

models <- mtcars |>
  split(mtcars$cyl) |>
  map(\(df) lm(mpg ~ wt + hp, data = df))

# extract R-squared for each
map_dbl(models, \(m) summary(m)$r.squared)
#>         4         6         8
#> 0.6867517 0.7591355 0.7299243

# extract coefficients as a tibble
map_dfr(models, \(m) as_tibble(t(coef(m))), .id = "cyl")
```

The `.id = "cyl"` argument of `map_dfr` (or, in newer `purrr`, `map(...) |> list_rbind(names_to = "cyl")`) inserts the list names as a new column, so the grouping variable travels with the results.

A related pattern iterates over bootstrap resamples, cross- validation folds, or Monte Carlo replicates. In each case, the iteration unit is a statistical object, and the function returns a summary (coefficient, $R^2$, AUC, …) per iteration. Collecting them with `map_dfr` or `map(...) |> list_rbind()` gives a tidy tibble ready for downstream plotting or summarisation.

Check your understanding: fitting models per group

**Question.** When you split a data frame by a grouping variable and fit the same model to each subset, why is the result naturally a list rather than a data frame?

**Answer.**

Each fitted model is a complex object with dozens of components (coefficients, residuals, fitted values, model frame, call, terms, …). It does not fit neatly into a single column of a data frame. A list preserves each model as a first-class object and lets downstream code extract whatever components matter for the analysis, $R^2$, coefficients, predicted values, diagnostic plots. Once you have extracted a summary

*per model*, that summary (often a tibble or a scalar) does fit into a data frame, and `map_dfr` or `list_rbind` produces the tidy result. The list-of-models → tibble-of-summaries pattern is the canonical functional shape for this kind of analysis.

## 4.15. Collaborating with an LLM on iteration

LLMs are good at producing first drafts of `purrr` code; they are less good at diagnosing why a pipeline produced unexpected output. Three patterns are worth learning explicitly.

**Prompt 1: rewriting a loop.** Paste a `for` loop and ask: 'rewrite this using `purrr` so the output is a tibble, one row per iteration. Preserve the iteration index as a column.'

*What to watch for.* The rewrite will probably use `map_dfr()` or `map() |> list_rbind()`. Verify the column types are what you expect, LLMs sometimes emit code that implicitly coerces a factor to character, or a Date to numeric, because each iteration returns a slightly different structure. Check with `dplyr::glimpse()` on both the original loop's output and the rewritten version.

*Verification.* Run both versions on at least three inputs (normal case, edge case, one empty iteration) and compare with `waldo::compare(original, rewritten)`. Treat any difference, no matter how trivial-looking, as a bug to investigate.

**Prompt 2: choosing between `map` variants.** Describe the shape of your input (a list of vectors, a named list of tibbles, two parallel vectors of parameters) and the shape you want the output to take, and ask: 'which `purrr` function is appropriate, and why?'

*What to watch for.* LLMs sometimes default to `map()` and `map_dfr()` even when a typed variant would catch bugs earlier. If you know the output is numeric, ask explicitly for `map_dbl()`; if the iteration is over a tibble of parameters, ask for `pmap()`. A second common pattern: LLMs sometimes suggest `map2()` when a plain `map()` over one input would do, or vice versa.

*Verification.* Trace through the call mentally. `map2(a, b, f)` requires `a` and `b` to have the same length; `pmap(list(a, b), f)` is equivalent. `map_dbl` requires every element to be a length-1 numeric. Check these constraints against your real data.

**Prompt 3: error handling.** Describe a function that sometimes fails and ask: 'wrap this with `safely()` so I can iterate over 10,000 inputs without losing the successful results when one fails. Return a tibble with columns for the result and the error message.'

*What to watch for.* The main risks are shape mismatches: `safely(f)` returns a list with `result` and `error` components, so the downstream tibble assembly needs to flatten that structure carefully. LLMs sometimes produce code that silently discards the error component, making failures invisible.

*Verification.* Run the pipeline on inputs that are known to include failures (for example, divide by zero, pass a negative number to `sqrt`, pass a missing file path to `read_csv`). Confirm that the output tibble has the expected number of rows, that the error column is populated for the known-failing inputs, and that the result column is non-`NULL` only for inputs you expect to succeed.

The meta-lesson: `purrr` code is short, which makes it tempting to accept LLM output at a glance. Short does not mean obviously-correct. Read every `map_*()` choice and every anonymous function with the same care you would give a named function.

## 4.16. Principle in use

Three habits separate effective functional code from clever functional code:

1. **Pick the iteration unit deliberately.** Name it. `map(resamples, fit_model)` reads better than `map(x, f)`.
2. **Declare return types.** Typed `map_*()` variants catch bugs at the source. Resort to untyped `map()` only when iterations genuinely produce heterogeneous outputs.

3. **Plan for failure.** For any non-trivial iteration, wrap the function with `safely()` or `possibly()`. Silent failures in iteration are the silent failures most likely to reach production.

Combine these three habits and `purrr` code becomes the clearest way to express a large class of statistical computations. Ignore them and it becomes another source of subtle bugs, just at a higher level of abstraction than a `for` loop would have been.

## 4.17. Exercises

1. Without using any `apply`/`map` functions, write `my_map()`: it takes a list `x` and a function `f` and returns a list of the same length. Test it on a simple example. Compare your implementation to the source of `purrr::map()` (it is short).
2. Use `purrr::map_dfr()` (or `map() |> list_rbind()`) to read every CSV in a directory and return a single tibble with a `source_file` column. Handle the case where one file is malformed using `safely()`.
3. Use `purrr::reduce()` to implement `sum()` for a numeric vector from scratch (without calling `sum()` or `+` on more than two numbers at a time).
4. Write a simulation that varies `n`, `mean`, and `sd` across a grid of values using `pmap()` over a tibble of parameters. For each combination, fit a one-sample t-test and extract the p-value. Return a tibble of results.
5. Run the simulation from exercise 4 with a bug that makes some parameter combinations fail (e.g., `n < 2`). Rewrite the pipeline with `safely()` so the successful runs are preserved and the failures are reported.

## 4.18. Further reading

- (Wickham, 2019) Chapter 9, the canonical treatment of functionals and `purrr`.
- (Wickham et al., 2023) iteration chapter, applied introduction paired with data-analysis examples.

- Jenny Bryan, *Iteration for data science* , a particularly clear treatment of the row-oriented/column-oriented distinction.

## 4.19. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 4.19.1. Question 1

What is the primary purpose of the `purrr` package in R?

- A) Data visualisation
- B) Statistical modelling
- C) Functional programming
- D) Database connectivity

> **Answer**
>
> **C.** `purrr` is a functional-programming toolkit that provides `map_*()` and related functions for iterating over vectors and lists with consistent, type-safe interfaces.

### 4.19.2. Question 2

Which `purrr` function would you use to apply a function to each element of a list and return the results as a numeric vector?

- A) `map()`
- B) `map_dbl()`
- C) `map_chr()`
- D) `map_lgl()`

Answer

**B.** `map_dbl()` returns a double (numeric) vector and errors if any element is not a length-1 numeric.

### 4.19.3. Question 3

In `purrr`, what does the tilde (`~`) operator represent in expressions like `map(1:5, ~ .x^2)`?

- A) A mathematical operator for exponentiation
- B) A shorthand for creating anonymous functions
- C) A pipe operator similar to `%>%`
- D) A logical negation operator

Answer

**B.** The tilde constructs a single-argument anonymous function where `.x` refers to the formal argument.

### 4.19.4. Question 4

Which of the following is the main risk of using `sapply()` in scripts and packages?

- A) It is slower than `lapply()`.
- B) Its return type depends on the run-time shape of the result, producing silent type changes between runs on different data.
- C) It does not support anonymous functions.
- D) It is deprecated in recent versions of R.

Answer

**B.** This is the core problem `vapply()` (in base R) and the typed `map_*()` variants (in `purrr`) were designed to fix.

### 4.19.5. Question 5

You have a function `fit_one(stratum)` that fits a model and sometimes fails. You want to iterate over 1000 strata, preserve successful fits, and record which strata failed with their error messages. Which tool is appropriate?

- A) `map()`
- B) `purrr::possibly(fit_one, otherwise = NA)`
- C) `purrr::safely(fit_one)`
- D) `purrr::walk(strata, fit_one)`

> **i** Answer
>
> **C.** `safely()` preserves both the result (for successful strata) and the error (for failing strata), which is what you need when you want to report *which* strata failed and *why*. `possibly()` discards the error; plain `map()` aborts at the first failure; `walk()` is for side effects.

## 4.20. Prerequisites answers

1. `purrr` provides a consistent family of functional-programming tools, chiefly the `map_*()` functions, which apply a function over a vector or list and return a predictable type. You reach for it when you want the behaviour of `lapply`/`sapply` but with type safety, consistent naming, and tidyverse-compatible interfaces. For one-off interactive work, `sapply` is fine; for scripts and packages, `map_dbl`, `map_chr`, etc. are safer because they error at the source when the expected type is not produced.
2. `map()` always returns a list, regardless of what its function returned. `map_dbl()`, `map_chr()`, and `map_lgl()` return atomic vectors of the named type and require each iteration to return a length-1 value of the matching type; they error immediately if an iteration returns something else. `map()` accepts heterogeneous output; the typed variants enforce homogeneity.
3. The tilde is a shortcut for a single-argument anonymous function whose body uses `.x` as the formal argument. `map(1:5, ~ .x^2)` is

equivalent to `map(1:5, \(x) x^2)` in R 4.1+. Both compute the squares of 1 through 5 and return the results as a list.

# Part II.

# Numerical Methods

# 5. Matrix Algebra in R

## 5.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 5.22.

1. State the identity for the inverse of a product of two invertible matrices, $(AB)^{-1}$. Why is the order of the factors reversed?
2. State the identity for the transpose of a product of two matrices, $(AB)^T$. How does the structural reason resemble the inverse identity?
3. Given an invertible matrix $A$, write two equivalent expressions for 'the inverse of the transpose of $A$' and explain why they are equal.

## 5.2. Learning objectives

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

- Create, subset, and modify matrices using base R, and avoid the common silent bugs (dimension drop, column-major fill, coercion to character).
- Distinguish between the * (elementwise) and `%*%` (matrix product) operators and explain the dimension requirements for each.
- Solve a linear system $Ax = b$ without inverting $A$, and explain why `solve(A) %*% b` is numerically and computationally wrong.
- Use `crossprod()`, `tcrossprod()`, `rowSums()`, `colSums()`, `rowMeans()`, `colMeans()` for efficient reductions.
- Compute OLS coefficients from the normal equations, and explain why `lm()` uses QR instead.
- Diagnose a near-singular matrix via `kappa()` and know the standard remedies (regularisation, pseudo-inverse).

## 5.3. Orientation

Every model in this book, linear regression, GLM, mixed models, ridge, lasso, Cox, factor analysis, reduces to a linear algebra computation on a carefully constructed matrix. This chapter develops the vocabulary and the numerical intuition you need for the rest of the book.

Under the hood, `lm()` performs a QR decomposition on the design matrix; `prcomp()` performs an SVD on a centred data matrix; mixed models, GLMs, and penalised regression all reduce to iterative linear algebra. Learning to think directly in matrices, rather than through the cushion of formula syntax, is the move that makes these methods transparent rather than magical.

The critical syntactic point, repeated on purpose: `*` is elementwise, `%*%` is matrix multiplication. Confusing them produces wrong answers rather than errors, and that class of bug is expensive to find.

## 5.4. The statistician's contribution

Matrix algebra is a domain where the mechanics, how to multiply, invert, decompose, are objective and lookupable. The judgement calls are elsewhere, and they separate correct numerical code from code that silently produces garbage.

**What to compute, and what to avoid computing.** The canonical example: if you want $\beta = (X^T X)^{-1} X^T y$, the operation you actually perform is *not* an explicit inverse followed by a multiply. It is `solve(crossprod(X), crossprod(X, y))` — or, better, a QR decomposition, which is what `lm()` does. Explicit inverses are slower and numerically less stable. The statistician's judgement is to refuse to compute an inverse unless the inverse itself is the object of interest (a variance-covariance matrix, typically).

**Condition number as an early-warning signal.** A near- singular design matrix produces coefficients whose error bars are meaningless. `kappa(A)` returns the condition number; values above $10^{10}$ or so signal that the computation is walking on a knife edge. The right response is not to run it anyway and see what happens. It is to diagnose (redundant columns?

collinearity? scaling issues?) and either fix the data or switch to a regularised method.

**Matrix vs. data frame.** Keep numeric data for linear algebra in matrices; keep mixed-type data for analysis in data frames. Converting back and forth is cheap; keeping numeric simulation output in a tibble through a million- iteration loop is expensive. The common failure mode is to leave data as a tibble because it was convenient to import, and then spend 100× more compute than necessary on the analysis. The reverse failure mode is to coerce a mixed-type data frame to a matrix and silently turn every numeric column into a character.

**Dimension hygiene.** The most common runtime failure in matrix code is 'non-conformable arguments', a dimension mismatch. The fix is not fancier error handling. It is the habit of checking `dim(A)` and `dim(B)` before multiplying, or writing the expected dimensions as a comment next to each matrix construction. When dimensions fail, the statistician knows what the matrices *should* be; the compiler does not.

These decisions are what make numerical statistical code trustworthy. None of them can be read off a reference card.

## 5.5. Matrix fundamentals

A matrix is a two-dimensional array of values (typically numeric) with a fixed number of rows and columns. Standard statistical notation uses bold uppercase letters: $\mathbf{A}$, with entries $a_{ij}$ indexed as *row, column*. An $m \times n$ matrix has $m$ rows and $n$ columns.

Special cases that will recur:

- Row vector ($1 \times n$) and column vector ($n \times 1$).
- Square matrix ($m = n$).
- Identity matrix $\mathbf{I}$: square, ones on the diagonal, zeros elsewhere. `diag(n)` produces the $n \times n$ identity.
- Diagonal matrix: nonzero only on the diagonal. `diag(v)` produces a diagonal matrix from vector `v`.
- Symmetric matrix: $\mathbf{A} = \mathbf{A}^T$. Covariance matrices are symmetric.

- Orthogonal matrix: $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$. The columns form an orthonormal basis. Rotations, reflections, and $\mathbf{Q}$ from a QR decomposition are all orthogonal.
- Positive definite matrix: symmetric, with $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ for all nonzero $\mathbf{x}$. Covariance matrices of non-degenerate data are positive definite; Cholesky decomposition applies.

## 5.6. Creating matrices

Four idioms cover nearly every practical case.

```
# 1. matrix() from a vector
# NOTE: column-major fill by default
A <- matrix(1:6, nrow = 2, ncol = 3)
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6

# row-major if you insist
matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)

# 2. rbind/cbind to assemble from pieces
X <- cbind(1, matrix(rnorm(n * p), n, p))   # design matrix

# 3. diag() for identity and diagonal matrices
I4 <- diag(4)
D  <- diag(c(1, 2, 3))

# 4. as.matrix() to coerce a numeric data frame
M <- as.matrix(my_numeric_df)
```

Column-major fill is the single most surprising behaviour for users coming from Python (NumPy defaults to row-major) or from spreadsheet intuition. The consequence at the machine level is that column operations are cache-friendly in R; this is why `colSums()` is faster than `rowSums()` on large matrices.

The `as.matrix()` gotcha: if any column of the data frame is non-numeric (character, factor), the whole matrix coerces to character. Silent. Common bug. When in doubt, check `storage.mode(M)` after the coercion.

Dimnames via `dimnames(A) <- list(row_names, col_names)` or the `dimnames` argument to `matrix()` make downstream code far more readable when matrices carry semantic meaning (subject IDs, gene names, covariate labels).

## 5.7. Indexing and subsetting

R uses `A[i, j]` for matrix indexing, with three patterns worth memorising:

```
A[i, ]          # row i, returned as a vector (dimension drop!)
A[, j]          # column j, returned as a vector
A[i:k, j:l]     # submatrix
```

The dimension drop is the source of the most common matrix bug in R. If downstream code expects a matrix and receives a vector, operations silently break. The fix:

```
A[2, , drop = FALSE]    # 1 x ncol(A) matrix
A[, 3, drop = FALSE]    # nrow(A) x 1 matrix
```

Logical indexing extends naturally:

```
A[A > 0]                    # vector of positive entries (loses sha
A[rowSums(A) > 0, ]         # rows with at least one positive entry
```

Assignment uses the same syntax on the left-hand side. Watch for recycling on the right-hand side: `A[, j] <- 0` fills column `j` with zeros (recycling the scalar); if you assign a wrong-length vector, R will recycle and you will get a bug rather than an error.

Check your understanding: dimension drop

**Question.** You write `rowMeans(X[i, ])` to compute the means across a subset of rows of `X`, but get an error. What went wrong?
**Answer.**
If `i` is a single index, `X[i, ]` returns a vector (not a matrix), and `rowMeans` refuses to operate on a vector. The fix is `X[i, , drop = FALSE]`, which preserves the matrix shape even when only one row is selected. This is among the most common dimension-drop bugs in R. In longer pipelines, the drop may happen far upstream of the error, which is why the habit of always writing `drop = FALSE` in matrix subsetting inside production code pays off.

## 5.8. Elementwise vs. matrix operations

The operators:

- `+`, `-`, `*`, `/` are **elementwise** and require identical dimensions (or recycling against a scalar).
- `%*%` is **matrix multiplication** and requires `ncol(A) == nrow(B)`; the result has shape `nrow(A) x ncol(B)`.

```
A <- matrix(1:4, 2, 2)
B <- matrix(5:8, 2, 2)

A * B        # elementwise product
#>      [,1] [,2]
#> [1,]    5   21
#> [2,]   12   32

A %*% B      # matrix product
#>      [,1] [,2]
#> [1,]   23   31
#> [2,]   34   46
```

Transpose is `t(A)`; the dimensions flip: if $A$ is $m \times n$, then $A^T$ is $n \times m$. The essential identity:

$$(AB)^T = B^T A^T$$

Scalar multiplication uses *: `2 * A` scales every element.

Check your understanding: * vs. %*%

**Question.** You multiply a $100 \times 5$ matrix by a $5 \times 1$ vector using `X * beta` and get 'non- conformable arrays' or an unexpected result. What is the correct operator, and what would `X * beta` actually compute?
**Answer.**
The correct operator for the matrix-vector product is %*%: `X %*% beta` produces a $100 \times 1$ vector of linear predictions. `X * beta` is elementwise; for a $100 \times 5$ matrix and a length-5 vector, R recycles the vector column-by-column, multiplying each column of `X` by the corresponding element of `beta` without summing. That can happen silently without an error and produce a matrix that looks plausible but is not the linear predictor. This confusion is the single most expensive bug in first-year applied R code. Hammer the distinction.

## 5.9. Solving linear systems

The central operation in statistical computing: solve $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$.

**Wrong:** `solve(A) %*% b`. This computes the full inverse $\mathbf{A}^{-1}$, then multiplies. It is roughly twice as expensive as the right way and numerically worse: the inverse amplifies small errors, and intermediate steps lose precision on ill-conditioned $\mathbf{A}$.

**Right:** `solve(A, b)`. This calls LAPACK's LU solver directly, skipping the inverse.

```
A <- matrix(c(2, 1, 1, 3), 2, 2)
b <- c(4, 5)
x <- solve(A, b)
A %*% x                # ≈ b
```

The same pattern applies to $\mathbf{A}^{-1}\mathbf{B}$: write `solve(A, B)`, not `solve(A) %*% B`.

Use an explicit `solve(A)` only when the inverse itself is the object of interest. The canonical case in regression is the coefficient variance-covariance matrix $\sigma^2(\mathbf{X}^T\mathbf{X})^{-1}$: you need the full inverse because you extract its diagonal for standard errors.

Condition number: `kappa(A)` (more precisely `kappa(A, exact = TRUE)`). A condition number above $10^{10}$ means you are losing about 10 decimal digits of precision to the computation; above $10^{14}$, with double-precision arithmetic, you have essentially no precision left.

```
kappa(A, exact = TRUE)
```

Remedies for near-singular systems:

- **Drop redundant columns.** If two columns of `X` are exactly or nearly collinear, one of them carries no new information.
- **Rescale.** Columns on wildly different scales inflate condition number without reflecting a real problem. Centre and scale before solving.
- **Ridge regularisation.** Replace $X^TX$ with $X^TX + \lambda I$ for small $\lambda > 0$. This is mathematically equivalent to a prior on the coefficients and numerically equivalent to regularising the problem.
- **Pseudo-inverse via SVD.** `MASS::ginv(A)` computes the Moore-Penrose pseudo-inverse, which is well defined even when $A$ is singular.

## 5.10. `crossprod()` and `tcrossprod()`

For the extremely common patterns $X^TX$ and $XX^T$, R provides specialised functions:

```
crossprod(X)          # t(X) %*% X
tcrossprod(X)         # X %*% t(X)
crossprod(X, y)       # t(X) %*% y
```

`crossprod(X)` is roughly twice as fast as `t(X) %*% X` because it exploits the symmetry of the result (it computes only the upper triangle and mirrors). This is a small optimisation for small matrices and a substantial one for large ones. In any code that computes the normal equations, using `crossprod` is a free speedup.

## 5.11. Vectorised reductions

For row and column reductions, base R provides specialised functions that are much faster than `apply`:

```
rowSums(A)
colSums(A)
rowMeans(A)
colMeans(A)
```

These dispatch to tight C loops. For these specific operations, always prefer them over `apply(A, 1, sum)` or `apply(A, 2, mean)`. The speedup is typically 10–50×.

`apply(A, MARGIN, FUN)` remains useful when `FUN` is custom (e.g., `apply(A, 1, quantile, probs = 0.9)`). But avoid it for the standard reductions.

## 5.12. BLAS and why R's linear algebra is fast

R delegates matrix multiplication, solves, and decompositions to BLAS (Basic Linear Algebra Subprograms) and LAPACK. These are Fortran/C libraries that have been optimised over decades with cache blocking, SIMD instructions, and multithreading.

The practical consequence: `A %*% B` for 1000×1000 matrices runs in a fraction of a second on a laptop. A double `for` loop computing the same quantity runs in tens of seconds or more.

```
n <- 500
A <- matrix(rnorm(n * n), n)
B <- matrix(rnorm(n * n), n)

system.time({
  C <- matrix(0, n, n)
  for (i in 1:n) for (j in 1:n) C[i, j] <- sum(A[i, ] * B[, j])
})
#>    user  system elapsed
#>  15.300   0.040  15.380

system.time(C2 <- A %*% B)
#>    user  system elapsed
#>   0.140   0.010   0.050
```

Typical speedup: 100× to 1000×, depending on BLAS configuration. Stock R ships with reference BLAS; installing OpenBLAS or Intel MKL often produces another 5×–20× on matrix-heavy code with no code changes. `sessionInfo()` reports which BLAS is active.

**Rule of thumb.** If you find yourself writing nested loops over matrix indices, pause. Almost always, the computation can be expressed as a matrix product, a reduction, or a broadcast, and almost always, doing so is 100× or more faster.

## 5.13. Eigenvalues and eigenvectors

For a square matrix $\mathbf{A}$, a nonzero vector $\mathbf{v}$ is an *eigenvector* with *eigenvalue* $\lambda$ if $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$. Geometrically, $\mathbf{A}$ stretches $\mathbf{v}$ without rotating it.

```
A <- matrix(c(4, 1, 1, 3), 2, 2)
e <- eigen(A)
e$values
e$vectors

# verify: A v = lambda v
```

```
A %*% e$vectors[, 1]
e$values[1] * e$vectors[, 1]
```

For **symmetric** matrices (e.g., covariance matrices), eigenvalues are real and eigenvectors are orthogonal. This is why PCA is clean: PCA is the eigendecomposition of the covariance matrix. The eigenvectors are the principal- component directions; the eigenvalues are the variances explained.

Other applications:

- **Spectral clustering**: eigenvectors of a graph Laplacian.
- **PageRank-style centrality**: dominant eigenvector of an adjacency matrix.
- **Stability of dynamical systems**: eigenvalues of the Jacobian.

The next chapter (decompositions) generalises this idea to SVD, which works for non-square and non-symmetric matrices and underlies much of numerical linear algebra.

## 5.14. Sparse matrices: the Matrix package

Base R matrices store every element, including zeros. For problems where most entries are zero, text data, graph adjacency, design matrices with many dummy variables, genomic data, this is wasteful in both memory and computation.

The `Matrix` package provides sparse matrix classes that store only the nonzero entries and dispatch to specialised algorithms that skip zero blocks.

```
library(Matrix)
S <- Matrix(0, 1000, 1000, sparse = TRUE)
S[1, 1]     <- 1
S[500, 500] <- 1
object.size(S)
```

Classes to know:

- `dgCMatrix`: general sparse, column-compressed storage. The default for most computations.
- `dgTMatrix`: triplet (row, col, value) form, convenient for construction.
- `dsCMatrix`: symmetric sparse.

Many modelling packages accept or return `Matrix` objects natively: `glmnet`, `lme4`, `MatrixModels`. Work involving genomics, NLP, or networks will encounter these regularly.

## 5.15. Worked example: OLS from the normal equations

The least-squares coefficient vector is

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Implementing this directly in R makes the formula concrete:

```
set.seed(1)
n <- 100; p <- 3
X <- cbind(1, matrix(rnorm(n * p), n, p))
y <- rnorm(n)

XtX <- crossprod(X)              # t(X) %*% X, faster
Xty <- crossprod(X, y)           # t(X) %*% y
beta_hat <- solve(XtX, Xty)      # solve, not inverse

# compare to lm()
beta_lm <- coef(lm(y ~ X[, -1]))
all.equal(as.vector(beta_hat), as.vector(beta_lm))
#> [1] TRUE
```

A few things are hiding in plain sight:

- We used `crossprod` instead of `t(X) %*% X`. The results are identical up to rounding; `crossprod` is about twice as fast.

- We used `solve(XtX, Xty)` instead of `solve(XtX) %*% Xty`. Same result, faster, more accurate.
- We did not invert $\mathbf{X}^T\mathbf{X}$. If we needed coefficient standard errors, we would compute the full inverse once for that purpose and extract its diagonal.

`lm()` does not actually use this formula. It uses QR. The reason is numerical: the condition number of $X$ is the square root of the condition number of $X^TX$, so the QR route is numerically stable by a factor of roughly the condition number. The next chapter unpacks this.

Check your understanding: `solve(A, b)` vs. `solve(A) %*% b`

**Question.** For a $1000 \times 1000$ system, about how much faster is `solve(A, b)` than `solve(A) %*% b`? And why is the second form numerically worse?
**Answer.**
Roughly $2\times$ faster. `solve(A, b)` performs one LU factorisation and a back-substitution, totalling about $\frac{2}{3}n^3$ floating-point operations. `solve(A) %*% b` performs the same LU, then completes the inverse (another $\frac{n^3}{3}$ FLOPs to invert a triangular matrix), then multiplies by $b$ ($n^2$ FLOPs more). About 50% more work than the direct solve, plus one materialised dense matrix that the direct path avoids.
The numerical harm is more important than the speed. The explicit inverse materialises a matrix whose entries can be large in magnitude even when the solve itself is well conditioned; those large entries are then multiplied by `b` and summed, amplifying rounding error. `solve(A, b)` avoids ever forming the inverse, so the intermediate quantities remain close in magnitude to the final answer, and precision is preserved.

## 5.16. Troubleshooting: common matrix errors

Three errors account for perhaps 80% of the matrix bugs users hit:

1. **Non-conformable arguments.** Dimensions do not match for the operation requested. Always inspect `dim(A)` and `dim(B)` before

multiplying. In pipelines, print the dimensions at each step until you find the mismatch.
2. **Computationally singular.** `solve(A)` fails because $A$ is (near) singular. Diagnose with `kappa(A)`; remedy with regularisation ($+\lambda I$) or with `MASS::ginv(A)` (Moore-Penrose pseudo-inverse).
3. **Silent dimension drop.** `A[i, ]` becomes a vector and downstream matrix operations misbehave. Use `drop = FALSE`.

Debugging habit: when something is wrong, print `dim()` of every matrix in the pipeline. The bug is almost always a dimension mismatch introduced upstream.

## 5.17. Collaborating with an LLM on matrix algebra

Matrix algebra is an area where LLMs are generally reliable about formulas and generally unreliable about numerical pitfalls. Three patterns work well.

**Prompt 1: translating math to R.** Paste the mathematical expression you want to compute and ask: 'translate this to idiomatic R, prefer `crossprod` over `t(X) %*% X`, and prefer `solve(A, b)` over explicit inverses.'

*What to watch for.* Without the nudge in the prompt, LLMs often produce `t(X) %*% X` and `solve(X'X) %*% X'y` because those expressions match the visual form of the formula. The nudges in the prompt shift it toward numerically sound idioms. Read the output and check that the dimensions make sense.

*Verification.* Run the code on a small known example (e.g., a $3 \times 2$ design matrix where you can compute the answer by hand) and compare. Then run on your real data and compare to `lm()` or the equivalent standard routine.

**Prompt 2: diagnosing a numerical failure.** Paste the failing code, the error message, and `kappa(A)`, and ask: 'is this a conditioning problem, a dimension problem, or something else?'

*What to watch for.* LLMs can diagnose conditioning correctly from `kappa(A)` output. They are less reliable at distinguishing 'your matrix is genuinely singular' from 'your matrix is fine but your workflow has a bug upstream'. If the suggested fix involves heavy regularisation, suspect that the diagnosis is wrong and the real problem is a bug, not conditioning.

*Verification.* Before applying any suggested regularisation, check the data: are there duplicate rows, exactly collinear columns, or extreme outliers? Those often surface as 'singular matrix' errors but are not fixed by regularisation.

**Prompt 3: benchmarking.** Ask the LLM to set up a `bench::mark()` comparison between `solve(A, b)`, `solve(A) %*% b`, and `qr.solve(A, b)` on a $1000 \times 1000$ matrix.

*What to watch for.* The benchmark output itself is fine. The interpretation can mislead: LLMs sometimes attribute speed differences to the wrong cause (e.g., claiming `qr.solve` is faster because of 'less memory allocation' when the real reason is the specific LAPACK routine). Do not copy the explanation into a methods section without verifying.

*Verification.* Run the benchmark. The ordering (`solve(A, b) < qr.solve(A, b) < solve(A) %*% b`) should be stable across runs and platforms. If your results differ substantially from published benchmarks, suspect BLAS configuration (reference BLAS vs. OpenBLAS vs. MKL) rather than an R-level difference.

## 5.18. Principle in use

Four habits define effective use of matrix algebra in R:

1. **Distinguish * from %*% reflexively.** It is the syntactic error with the highest cost-per-character in statistical code.
2. **Solve systems, do not invert matrices.** Compute an inverse only when the inverse itself is the object of interest.
3. **Use `crossprod` and vectorised reductions.** Free speedups that also read more clearly once the idioms are familiar.

4. **Check dimensions and condition numbers.** Dimension mismatches and ill-conditioning are the two big silent failure modes. Make their diagnosis routine, not exceptional.

These habits do not make linear algebra easy, the math is what the math is, but they make the R encoding of that math trustworthy.

## 5.19. Exercises

1. Generate a random $500 \times 500$ matrix `A` and vector `b`. Solve $Ax = b$ three ways: `solve(A) %*% b`, `solve(A, b)`, and `qr.solve(A, b)`. Time each and compare the solutions with `all.equal()`. Which is fastest? Do all three agree?
2. Compute `crossprod(X)` and `t(X) %*% X` on a $10{,}000 \times 50$ matrix. Benchmark both. Explain which is faster and why. Is the difference a rounding error or a true performance gap?
3. Construct a symmetric positive-definite matrix by `A %*% t(A) + diag(n) * 1e-6`. Verify positive- definiteness with `eigen()` or `chol()`. What does `kappa()` return? What happens if you omit the ridge term?
4. Implement OLS for `y ~ X` from scratch using `crossprod(X)` and `solve(...)`. Also compute the residual sum of squares, $\hat{\sigma}^2$, the coefficient variance-covariance matrix, standard errors, t-statistics, and p-values. Compare every element of your output to `summary(lm(y ~ X - 1))`. (Strip the intercept if your X already has a column of 1s.)
5. Construct a matrix $X$ with two nearly collinear columns (e.g., `x2 = x1 + rnorm(n, sd = 1e-8)`). Compute `kappa(crossprod(X))`. Solve the normal equations. How unstable is $\widehat{\beta}$? Try `solve(crossprod(X) + 0.01 * diag(p), ...)` (ridge) and compare.

## 5.20. Further reading

- (Golub & Van Loan, 2013), the reference for matrix computations.
- (Gentle, 2024), more recent, with stronger statistical orientation.
- (Strang, 2016), the MIT OpenCourseWare videos are the canonical linear-algebra refresher.

- (Petersen & Pedersen, 2012), quick reference for the matrix identities that appear in statistics papers.

## 5.21. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 5.21.1. Question 1

Which of the following identities correctly represents the inverse of a product of two matrices?

- A) $(AB)^{-1} = A^{-1}B^{-1}$
- B) $(AB)^{-1} = B^{-1}A^{-1}$
- C) $(AB)^{-1} = (A^{-1})^T(B^{-1})^T$
- D) $(AB)^{-1} = (B^{-1}A^{-1})^T$

> **i** Answer
>
> **B.** Inversion reverses the order of a matrix product because you must undo $B$ before undoing $A$.

### 5.21.2. Question 2

Which identity correctly describes the transpose of a product of two matrices?

- A) $(AB)^T = A^TB^T$
- B) $(AB)^T = B^TA^T$
- C) $(AB)^T = A^T + B^T$
- D) $(AB)^T = A^{-1}B^{-1}$

Answer

**B.** Transposition, like inversion, reverses the order of a matrix product.

### 5.21.3. Question 3

Let $A$ be an invertible matrix. Which of the following identities involving transposition and inversion is valid?

- A) $(A^T)^{-1} = (A^{-1})^T$
- B) $(A^T)^{-1} = A^{-1}$
- C) $(A^{-1})^T = A^T$
- D) $(A^T)^{-1} = A^T$

Answer

**A.** Inversion and transposition commute on invertible matrices.

### 5.21.4. Question 4

Which expression is numerically preferred for solving $\mathbf{Ax} = \mathbf{b}$?

- A) `solve(A) %*% b`
- B) `solve(A, b)`
- C) `ginv(A) %*% b`
- D) `A / b`

Answer

**B.** `solve(A, b)` calls an LU solver directly. Option A computes the full inverse first (slower, less stable). Option C uses a pseudo-inverse unnecessarily. Option D is not a valid matrix operation.

### 5.21.5. Question 5

You extract a single row with `X[i, ]` and pass it to `rowMeans()`. R throws an error. What happened?

- A) `rowMeans` does not accept numeric input.
- B) `X[i, ]` returns a vector because of R's default dimension drop; `rowMeans` requires a matrix.
- C) The row is empty.
- D) `X` is a tibble, not a matrix.

> **i** Answer
>
> **B.** Use `X[i, , drop = FALSE]` to preserve the matrix structure. This is among the most common silent-then- loud bugs in matrix code.

## 5.22. Prerequisites answers

1. $(AB)^{-1} = B^{-1}A^{-1}$. The order reverses because you must undo $B$ before undoing $A$: $(AB)(B^{-1}A^{-1}) = A(BB^{-1})A^{-1} = AIA^{-1} = I$.
2. $(AB)^T = B^T A^T$. Like the inverse identity, the order reverses. The structural reason is the same: transposition, like inversion, reverses the composition of linear maps. Algebraically, the proof is the same kind of telescoping.
3. $(A^T)^{-1} = (A^{-1})^T$. Both equal the matrix that, when multiplied by $A^T$, yields the identity. Inversion and transposition commute on invertible matrices, so either ordering produces the same result. One practical use: in the OLS variance-covariance formula, $(X^T X)^{-1}$ is symmetric, so its transpose equals itself; this identity is what guarantees that.

# 6. Matrix Decompositions

## 6.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 6.17.

1. What is LU decomposition, and what class of matrices does it apply to?
2. In the QR decomposition $A = QR$, what structural property does $Q$ have, and what structural property does $R$ have?
3. Among QR, LU, Cholesky, and SVD, which decomposition does `lm()` use internally, and why?

## 6.2. Learning objectives

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

- State the definition, uniqueness conditions, and computational cost of the LU, Cholesky, QR, eigendecomposition, and singular-value decompositions.
- Choose the appropriate decomposition for a given computational task: solving $Ax = b$, fitting OLS, inverting a covariance matrix, diagnosing rank, or compressing data.
- Compute OLS coefficients via `qr()` and explain why this is more numerically stable than the normal equations.
- Use the SVD to compute a Moore-Penrose pseudoinverse and a low-rank (rank-$k$) approximation.
- Use the Cholesky decomposition for multivariate-normal simulation, Mahalanobis distances, and efficient solves on symmetric positive-definite systems.

## 6.3. Orientation

Direct methods for numerical linear algebra all factor a matrix into simpler pieces. The three you will actually use often in biostatistics are QR (for regression), Cholesky (for positive-definite systems, including covariance matrices), and SVD (for rank-deficient or near-singular problems). LU is the engine behind `solve()` but appears less often in user code. Eigendecomposition appears via its product — principal components, spectral methods, rather than by name.

Every decomposition factors $A$ into a product with special structure: triangular, orthogonal, diagonal. The structure is what makes downstream computations fast or stable. Solving a triangular system is $O(n^2)$ instead of $O(n^3)$; an orthogonal transformation preserves norms and so does not amplify rounding error. The decompositions are therefore not mathematical curiosities; they are the reason statistical software can fit models on realistic data at all.

## 6.4. The statistician's contribution

Decompositions are an area where the *mathematics* is fully determined, a Cholesky decomposition of a given SPD matrix is unique up to sign, but the *engineering* judgement is not. That judgement is where the statistician contributes.

**Pick the decomposition that matches the problem's structure.** For a symmetric positive-definite matrix (covariance, $X^TX$), Cholesky is twice as fast as LU and numerically stable. For a general square matrix that needs a solve, LU (via `solve()`). For a rectangular matrix in a least-squares problem, QR. For rank diagnosis or low-rank approximation, SVD. Using the wrong one wastes cycles or, in ill-conditioned problems, amplifies error.

**Prefer decomposition results to explicit inverses.** The same principle as in the previous chapter, elevated. `chol(Sigma)` plus a triangular solve is faster and more stable than computing `solve(Sigma)` and multiplying. If you find yourself computing an explicit inverse, ask whether a decomposition

would give you what you actually need, a determinant, a Mahalanobis distance, a Cholesky factor for simulation, without materialising $\Sigma^{-1}$.

**Rank deficiency is a signal, not a bug to work around.** When `lm()` tells you a design matrix is rank deficient, the answer is not 'use a pseudoinverse and move on'. It is to understand *why* the design is rank deficient. Collinear covariates, an interaction with no variation in one cell, a factor level with zero observations: each of these is diagnostic information about the data, not a numerical nit to smooth over. The SVD makes the rank visible; it is the statistician's job to decide what to do about it.

**Low-rank approximations have a statistical interpretation.** Truncated SVD is not generic 'compression'. In PCA, the truncated SVD is the variance-maximising projection onto $k$ dimensions. In factor analysis, it is the low-rank structure of the observed correlations. In collaborative filtering, it is a latent-factor model. The number $k$ should be chosen with a statistical justification (variance explained, cross-validated reconstruction error, substantive interpretation), not picked arbitrarily because '30 looks like a round number'.

These are the decisions no textbook algorithm can make. They are what separate numerical routines from defensible statistical analyses.

## 6.5. Decompositions at a glance

| Name | Factors | Requires | Cost | Typical use |
|---|---|---|---|---|
| LU | $PA = LU$ | square, nonsingular | $\sim n^3/3$ | `solve(A, b)`; the default direct solver |
| Cholesky | $A = LL^T$ | symmetric positive definite | $\sim n^3/6$ | Covariance matrices; MVN simulation |
| QR | $A = QR$ | $n \geq p$ | $\sim 2np^2$ | Least squares; `lm()` internals |
| Eigen | $A = V\Lambda V^{-1}$ | square | $\sim n^3$ | Symmetric matrices → PCA, spectral methods |
| SVD | $A = U\Sigma V^T$ | any | $\sim mn^2$ | Rank, pseudoinverse, low-rank approximation |

The costs are for dense matrices. Sparse variants exist and are faster by factors that depend on sparsity pattern.

## 6.6. Cholesky decomposition

For a symmetric positive-definite matrix $A$, the Cholesky decomposition is a factorisation

$$A = LL^T$$

with $L$ a lower triangular matrix with positive diagonal. (R's `chol()` returns $L^T$, upper triangular, for historical reasons.)

```
X <- matrix(rnorm(12), 4, 3)
Sigma <- crossprod(X)       # symmetric PSD
R <- chol(Sigma)            # upper triangular, Sigma = t(R) %*% R
all.equal(t(R) %*% R, Sigma)
#> [1] TRUE
```

Why Cholesky matters in statistics:

1. **Fast solves on SPD systems.** `chol(Sigma)` followed by two triangular solves costs about half what `solve(Sigma)` costs and is numerically more stable. R's `chol2inv()` uses exactly this route.
2. **Positive-definiteness test.** If `chol()` succeeds, the matrix is SPD. If it fails with an error, it is not. This is the fastest way to diagnose 'is my covariance matrix genuinely a covariance matrix?'
3. **Multivariate normal simulation.** To simulate $\mathbf{x} \sim N(\mu, \Sigma)$:

   ```
   L <- t(chol(Sigma))                 # lower triangular
   x <- mu + L %*% rnorm(ncol(Sigma))
   ```

   A single Cholesky suffices for any number of draws.
4. **Mahalanobis distance.** Instead of `t(x) %*% solve(Sigma) %*% x`, use `sum(backsolve(R, x, transpose = TRUE)^2)`, which uses the Cholesky factor and triangular solves.

The one gotcha: matrices that are *theoretically* SPD can be *numerically* non-SPD because of floating-point noise, a covariance matrix estimated from data with one exactly collinear pair of columns will have a tiny-negative eigenvalue. The standard workaround is to add a small ridge: `chol(Sigma + 1e-8 * diag(p))`. Whether to do this silently or to treat it as a modelling problem is the statistician's call.

## 6.7. QR decomposition

For any $n \times p$ matrix $A$ with $n \geq p$, QR produces

$$A = QR$$

where $Q$ has orthonormal columns ($Q^T Q = I$) and $R$ is upper triangular. R's `qr()` stores the factors in a compact form; helpers `qr.Q()`, `qr.R()`, and `qr.coef()` extract them.

```
set.seed(1)
n <- 100; p <- 3
X <- cbind(1, matrix(rnorm(n * p), n, p))
y <- rnorm(n)

qrX <- qr(X)
Q <- qr.Q(qrX)
R <- qr.R(qrX)

beta_qr <- qr.coef(qrX, y)
beta_lm <- coef(lm(y ~ X[, -1]))
all.equal(as.vector(beta_qr), as.vector(beta_lm))
#> [1] TRUE
```

**Why `lm()` uses QR, not the normal equations.** The normal equations $(X^T X)\hat{\beta} = X^T y$ are mathematically correct. But the condition number of $X^T X$ is the *square* of the condition number of $X$:

$$\kappa(X^T X) = \kappa(X)^2$$

If $X$ is ill-conditioned, collinear predictors, widely different scales, polynomial expansions with high-order terms — then $X^TX$ is spectacularly ill-conditioned. In finite- precision arithmetic, this can cost you 5, 10, or more decimal digits of accuracy in $\hat{\beta}$.

QR skips $X^TX$ entirely. It operates on $X$ directly, with orthogonal transformations that preserve norms and therefore do not amplify errors. The computed $\hat{\beta}$ is as accurate as the input data allows.

For a well-conditioned $X$, the two routes agree to many digits. For ill-conditioned $X$, QR wins by a factor equal to the condition number. The Longley dataset is a classic demonstration; the normal-equations route can disagree with QR in the 4th decimal place on coefficients that are nominally correct to 8.

**Computational cost.** QR on $X$ is about $2np^2$ FLOPs (Householder reflections) vs. normal equations at $np^2 + p^3/3$. For $n \gg p$ (the usual statistical case), the costs are similar; QR's stability advantage comes free.

**Column pivoting.** R's default `qr(X)` uses LINPACK (`LAPACK = FALSE`) for backward compatibility. Pass `LAPACK = TRUE` to use the modern LAPACK routines, which provide column pivoting. The permutation is returned in `qrX$pivot`, and the decomposition satisfies $AP = QR$. This is how R detects rank deficiency: the diagonal of $R$ after pivoting has small values for redundant columns.

Check your understanding: why not the normal equations?

**Question.** `lm()` could compute $\hat{\beta} = (X^TX)^{-1}X^Ty$ directly. Why doesn't it?

**Answer.**

The condition number of $X^TX$ equals the condition number of $X$ squared. If $X$ is ill-conditioned (collinear predictors, polynomial regression with high-degree terms, covariates on vastly different scales), $X^TX$ is catastrophically ill-conditioned. In finite-precision arithmetic, you lose roughly $\log_{10} \kappa(X)^2$ significant digits in $\hat{\beta}$. QR operates directly on $X$ with orthogonal transformations, preserving norms and therefore preserving precision. For a well-conditioned $X$, both routes

produce the same answer to rounding error. For an ill-conditioned $X$, QR is vastly more trustworthy. Cost is comparable for $n \gg p$; the stability advantage is essentially free. `lm()` uses QR for these reasons.

## 6.8. LU decomposition

For any square nonsingular matrix $A$, the LU decomposition (with partial pivoting) is

$$PA = LU$$

where $L$ is lower triangular with unit diagonal, $U$ is upper triangular, and $P$ is a row-permutation matrix selected for numerical stability.

R does not expose the LU factorisation directly (there is no `lu()` in base R). Its effect appears inside `solve(A, b)`, which calls LAPACK's `dgesv` routine. For solving general linear systems, this is the default direct method.

The partial-pivoting strategy chooses each pivot as the largest-magnitude element in its column, which bounds error growth. Complete pivoting (both row and column permutations) gives tighter theoretical bounds but rarely outperforms partial pivoting on real matrices and is more expensive, so partial pivoting is the standard.

For sparse matrices, specialised LU implementations (`Matrix::lu`) use graph-theoretic reordering to minimise fill-in (zeros in $A$ that become nonzero in $L$ or $U$).

For symmetric positive-definite systems, Cholesky is preferred: half the FLOPs and numerically cleaner. For general systems, LU is the default.

## 6.9. Eigendecomposition

For a square matrix $A$, an eigenpair $(\lambda, v)$ satisfies $Av = \lambda v$. The eigendecomposition collects all eigenpairs:

$$A = V\Lambda V^{-1}$$

where $V$ has the eigenvectors as columns and $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$.

In R, `eigen(A)` returns a list with components `values` (eigenvalues, in descending order of magnitude) and `vectors` (eigenvectors in the same order):

```
A <- matrix(c(4, 1, 1, 3), 2, 2)
e <- eigen(A)
e$values
#> [1] 4.618034 2.381966
A %*% e$vectors[, 1] - e$values[1] * e$vectors[, 1]
#> [1] 0 0      (numerical zero)
```

For **symmetric** matrices, eigenvalues are real and eigenvectors are orthogonal: $V^{-1} = V^T$, so $A = V\Lambda V^T$. This is why statistical applications of eigendecomposition almost always concern symmetric inputs (covariance matrices, correlation matrices, graph Laplacians): the decomposition is numerically well-behaved, and the orthogonality of $V$ has a clean statistical meaning (principal components are uncorrelated).

For non-symmetric matrices, eigendecomposition may not exist (deficient matrices) or may have complex eigenvalues. In statistical practice, non-symmetric eigenproblems are rare; when they arise, the SVD is usually a better tool.

**Cost.** The full eigendecomposition of a general $n \times n$ matrix is $O(n^3)$. For symmetric matrices, R uses a tridiagonalisation plus divide-and-conquer algorithm, with the same asymptotic cost but a smaller constant.

**Partial eigendecomposition.** If only the top $k$ eigenpairs are needed (often true in high-dimensional statistics), Krylov methods like Lanczos achieve $O(kn^2)$ cost, substantially faster than the full decomposition. The `RSpectra` package wraps the standard implementation.

## 6.10. Singular value decomposition (SVD)

For any $m \times n$ matrix $A$,

$$A = U\Sigma V^T$$

where $U$ is $m \times m$ orthogonal, $V$ is $n \times n$ orthogonal, and $\Sigma$ is $m \times n$ diagonal with non-negative entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$. The $\sigma_i$ are the *singular values* of $A$.

```
A <- matrix(rnorm(30), 6, 5)
s <- svd(A)
str(s)          # $d (singular values), $u, $v
all.equal(A, s$u %*% diag(s$d) %*% t(s$v))
#> [1] TRUE
```

**Why the SVD is the most versatile decomposition.**

1. It exists for every matrix, without structural requirements (no symmetry, no positive definiteness, no squareness).
2. Singular values are non-negative real numbers, unlike eigenvalues of general matrices which can be complex.
3. The largest singular value is the matrix's 2-norm; the smallest nonzero singular value is the reciprocal of the condition number. Both quantities drop out of the SVD directly.
4. SVD gives the rank: the number of nonzero singular values. In practice, 'nonzero' is interpreted relative to a tolerance, because singular values below roughly $\epsilon\|A\|$ (with $\epsilon \approx 10^{-16}$) are numerically indistinguishable from zero.

**Economy (thin) SVD.** For $m \geq n$, `svd(A, nu = n, nv = n)` (or the default `svd(A)`) returns $U$ as $m \times n$, $\Sigma$ as length-$n$ diagonal, and $V$ as $n \times n$. This is both cheaper and smaller in memory than the full SVD.

**Cost.** $O(mn \min(m, n))$, dominated by an initial bidiagonalisation step. For large rectangular matrices, this is expensive but tractable. For huge sparse matrices, randomised SVD methods (e.g., `rsvd::rsvd`) reduce cost substantially when only the top few singular values are needed.

### 6.10.1. Pseudoinverse via SVD

The Moore-Penrose pseudoinverse $A^+$ of any matrix $A$ is

$$A^+ = V\Sigma^+ U^T$$

where $\Sigma^+$ has each positive singular value $\sigma_i$ replaced by $1/\sigma_i$ and each zero singular value left as zero. `MASS::ginv(A)` implements exactly this.

The pseudoinverse satisfies the four Moore-Penrose axioms ($AA^+A = A$, $A^+AA^+ = A^+$, $(AA^+)^T = AA^+$, $(A^+A)^T = A^+A$) and gives the minimum-norm least-squares solution to $Ax = b$: $\hat{x} = A^+b$ minimises $\|Ax - b\|$ and, among all minimisers, minimises $\|x\|$. For rank-deficient regression, this is what a naive 'just solve it' approach is implicitly doing.

### 6.10.2. Low-rank approximation

The Eckart-Young-Mirsky theorem: the best rank-$k$ approximation to $A$ (in the Frobenius or 2-norm) is

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T = U_k \Sigma_k V_k^T$$

obtained by truncating the SVD to the top $k$ singular triples. The approximation error equals $\sigma_{k+1}^2 + \cdots + \sigma_{\min(m,n)}^2$ (Frobenius) or $\sigma_{k+1}$ (2-norm).

```
svd_A <- svd(A)
k <- 3
A_k <- svd_A$u[, 1:k] %*% diag(svd_A$d[1:k]) %*% t(svd_A$v[, 1:k])
```

Applications:

- **PCA.** Truncated SVD on the centred data matrix gives the top-$k$ principal components. The singular values squared (divided by $n-1$) are the variances of the principal components.

- **Image compression.** A rank-$k$ approximation stores $k(m+n+1)$ numbers instead of $mn$. For natural images, surprisingly small $k$ captures most of the structure.
- **Collaborative filtering.** Matrix completion via truncated SVD implements a latent-factor model for recommender systems.
- **Denoising.** If the signal is low-rank and the noise is full-rank, truncating to the signal's rank removes noise.

Choosing $k$ is a modelling decision, not a mechanical one. Common approaches: cumulative variance explained ('keep components until 95% of variance is captured'), scree-plot inspection (look for an 'elbow'), cross-validation on a reconstruction error, and parallel analysis (compare observed eigenvalues against those of a random matrix of the same shape).

Check your understanding: why SVD for rank deficiency

**Question.** You have a design matrix with two exactly collinear columns. `solve(crossprod(X), crossprod(X, y))` errors. What does `svd(X)` show, and how does the pseudoinverse route handle the problem?

**Answer.**

`svd(X)` reveals the rank deficiency: one singular value will be effectively zero (numerical noise, but orders of magnitude smaller than the others). The rank of $X$ is the count of non-negligible singular values. `MASS::ginv(X)` computes the Moore-Penrose pseudoinverse by inverting only the non-zero singular values (zero stays zero); the resulting $\hat{\beta} = \text{ginv}(X)y$ is the minimum- norm solution. `lm()` handles this by dropping one of the collinear columns (reported in the `aliased` field of the summary); the pseudoinverse solution does not drop columns but distributes the coefficient weight across collinear columns. Both are defensible; which is right depends on what the analyst meant by including both columns in the first place. The SVD makes the rank visible so that the choice is deliberate rather than silent.

## 6.11. Worked example: OLS via QR

```
set.seed(1)
n <- 100; p <- 3
X <- cbind(1, matrix(rnorm(n * p), n, p))
y <- rnorm(n)

qrX <- qr(X)
beta <- qr.coef(qrX, y)

# residuals via qr.resid
res <- qr.resid(qrX, y)
rss <- sum(res^2)

# residual variance
sigma2 <- rss / (n - ncol(X))

# standard errors: diag((X'X)^-1) * sigma^2
# via R from the QR: (X'X)^-1 = solve(R) %*% t(solve(R))
R <- qr.R(qrX)
Rinv <- backsolve(R, diag(ncol(R)))
vcov_beta <- sigma2 * tcrossprod(Rinv)
se <- sqrt(diag(vcov_beta))

# compare to lm()
fit <- lm(y ~ X[, -1])
all.equal(as.vector(beta), as.vector(coef(fit)))
#> [1] TRUE
all.equal(as.vector(se), as.vector(summary(fit)$coefficients[, 2]))
#> [1] TRUE
```

Two idioms in this code repay study. `qr.resid()` computes $(I - QQ^T)y$ in one call instead of forming $QQ^T$ explicitly. `backsolve(R, diag(p))` inverts a triangular matrix by solving $Rx = e_i$ for each column $e_i$ of the identity, cheaper and more stable than `solve(R)`.

## 6.12. Collaborating with an LLM on decompositions

Decompositions are mathematically precise but numerically subtle. LLMs are helpful for the math and variable on the numerics. Three patterns:

**Prompt 1: picking the right decomposition.** Describe the matrix (symmetric? positive definite? rectangular? ill-conditioned?) and the downstream goal (solve? rank? low-rank approximation? simulation?). Ask: 'which decomposition is appropriate and why?'

*What to watch for.* The answer will usually cite the right one. Check the reasoning against the table at the top of this chapter. LLMs sometimes recommend SVD when Cholesky would do, because SVD is more 'general'; that generality is not free.

*Verification.* Run the proposed decomposition and check that the downstream result agrees with a cross-reference (e.g., the SVD pseudoinverse should agree with `lm()` on a full-rank problem).

**Prompt 2: translating math to R.** Paste the algebraic expression (e.g., 'the Cholesky factor of $X^TX + \lambda I$') and ask for idiomatic R.

*What to watch for.* Hand-written code that mixes `t(X) %*% X` with `solve(...)` when `chol(crossprod(X) + lambda * diag(p))` plus `backsolve` would be faster and more stable. Nudge the prompt explicitly toward decomposition-based idioms.

*Verification.* Compare against a reference implementation for small examples, and watch for the numerical behaviour as $\lambda \to 0$ (should approach the unregularised OLS solution) and $\lambda \to \infty$ (should approach zero).

**Prompt 3: choosing $k$ for truncation.** Describe the data and goal and ask: 'how should I choose $k$ for a rank-$k$ SVD approximation?'

*What to watch for.* Answers often default to '95% variance explained'. That is a rule of thumb, not a theorem. For exploratory PCA it is reasonable; for denoising or a downstream predictive task, cross-validated reconstruction error is usually better.

*Verification.* Whatever rule you use, compute the reconstruction at several values of $k$ and look at the scree plot or the CV curve. The right $k$ is often visually obvious on real data and rarely matches the 'default' rule.

The meta-lesson: LLMs are good at recalling the names, formulas, and canonical uses of decompositions, and reasonable at translating math to R. They are less reliable at the choice between decompositions on a given problem and the choice of truncation parameter. Those are engineering judgements informed by data.

## 6.13. Principle in use

Three habits characterise effective use of decompositions:

1. **Let the decomposition match the structure.** SPD → Cholesky. Rectangular least squares → QR. Rank diagnosis or near-singular → SVD. Using the general-purpose tool when a specialised one applies costs time and precision.
2. **Prefer decomposition-based solves to explicit inverses.** Same principle as matrix algebra, but with the additional leverage that the decomposition gives you extras (determinants, Mahalanobis distances, simulation) for free.
3. **Treat rank deficiency as information.** The SVD makes rank visible. Use it to understand *why* the design matrix is deficient, not just to smooth over the resulting errors.

## 6.14. Exercises

1. Implement OLS two ways: via the normal equations (`solve(crossprod(X), crossprod(X, y))`) and via QR (`qr.coef(qr(X), y)`). Compare the coefficients on a well-conditioned problem and on the Longley dataset (`datasets::longley`). Explain any discrepancy. How does `kappa(X)` differ between the two problems?
2. Show numerically that the eigenvalues of `crossprod(X)` are the squared singular values of `X`. Verify on a random $100 \times 5$ matrix using `eigen` and `svd`.
3. Use `svd()` to compute the Moore-Penrose pseudoinverse of a rank-2 matrix of size $5 \times 3$. Verify the four pseudoinverse axioms numerically.

4. Generate a $200 \times 200$ symmetric positive-definite matrix. Simulate 10,000 draws from $N(0, \Sigma)$ using Cholesky. Check that the sample covariance matrix agrees with $\Sigma$ to the accuracy expected at that sample size.
5. Download a small grayscale image and represent it as a matrix of pixel intensities. Compute its SVD. Plot the image reconstructed from the top $k$ singular triples for $k \in \{5, 20, 50, 100\}$. At what $k$ does the reconstruction become visually indistinguishable from the original?

## 6.15. Further reading

- (Golub & Van Loan, 2013) Chapters 4–5, LU, Cholesky, QR, and SVD with algorithmic detail. The canonical reference.
- (Gentle, 2024), more accessible, with explicit statistical framing.
- (Trefethen & Bau III, 1997), the text that made numerical linear algebra accessible to a generation.
- (Halko et al., 2011), the standard reference for randomised algorithms for large-scale matrix decomposition.

## 6.16. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 6.16.1. Question 1

Which of the following statements about LU decomposition is true?

- A) LU decomposition requires the matrix to be symmetric and positive definite.
- B) LU decomposition factors a square matrix into a lower and an upper triangular matrix.
- C) LU decomposition applies only to diagonal matrices.
- D) LU decomposition requires the matrix to be orthogonal.

Answer

**B.** LU decomposition writes $A = LU$ (with partial pivoting, $PA = LU$) for any square nonsingular matrix.

### 6.16.2. Question 2

In the QR decomposition $A = QR$, which of the following is always true?

- A) $R$ is lower triangular.
- B) $Q$ is orthogonal and $R$ is upper triangular.
- C) $Q$ is diagonal and $R$ is symmetric.
- D) QR decomposition only applies to square matrices.

Answer

**B.** $Q$ has orthonormal columns ($Q^TQ = I$) and $R$ is upper triangular. QR applies to any matrix with $n \geq p$.

### 6.16.3. Question 3

Why does `lm()` use QR decomposition instead of the normal equations?

- A) QR is always faster.
- B) QR avoids forming $X^TX$, whose condition number is $\kappa(X)^2$; QR preserves $\kappa(X)$.
- C) QR is the only method that produces coefficient standard errors.
- D) The normal equations do not exist for rectangular matrices.

Answer

**B.** The stability argument is the reason. Cost is similar; accuracy is substantially better for ill- conditioned designs.

### 6.16.4. Question 4

Which decomposition would you use to simulate from a multivariate normal $N(\mu, \Sigma)$?

- A) LU of $\Sigma$.
- B) QR of $\Sigma$.
- C) Cholesky of $\Sigma$.
- D) SVD of the data matrix.

**i** Answer

**C.** If $L$ is the lower-triangular Cholesky factor of $\Sigma$ and $z \sim N(0, I)$, then $\mu + Lz \sim N(\mu, \Sigma)$. Cholesky is half the cost of LU on SPD matrices and numerically cleaner.

### 6.16.5. Question 5

The Eckart-Young-Mirsky theorem states that the best rank- $k$ approximation to $A$ in the Frobenius norm is:

- A) The first $k$ rows of $A$.
- B) The first $k$ columns of $A$.
- C) The truncated SVD using the top $k$ singular triples.
- D) Any rank-$k$ matrix whose Frobenius norm equals that of $A$.

**i** Answer

**C.** This is why truncated SVD underlies PCA, image compression, matrix completion, and low-rank denoising.

## 6.17. Prerequisites answers

1. LU decomposition factors a square matrix $A$ into a lower triangular $L$ and an upper triangular $U$ such that $A = LU$ (in practice, with partial pivoting, $PA = LU$). It applies to any square nonsingular

matrix. It is the standard direct method for solving general linear systems and underlies `solve()`.

2. In $A = QR$, $Q$ is orthogonal ($Q^TQ = I$) and $R$ is upper triangular. For rectangular $A$ with $n \geq p$, a thin QR gives $Q$ as $n \times p$ and $R$ as $p \times p$. The orthogonality of $Q$ is why QR-based solves are numerically stable.
3. `lm()` uses QR. QR applied to $X$ has condition number $\kappa(X)$, whereas forming the normal equations $X^TX$ squares the condition number to $\kappa(X)^2$ and amplifies numerical error for ill-conditioned designs. For a well-conditioned $X$ the two routes agree to rounding error; for ill-conditioned $X$ they can differ by many decimal digits, always in favour of QR.

# 7. Optimization and Numerical Methods

## 7.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 7.21.

1. In gradient descent applied to a minimisation problem, in which direction does each step move, and why?
2. What is the risk of setting the learning rate too large in gradient descent?
3. What is the primary advantage of BFGS over Newton's method, and what property of an optimisation problem guarantees a unique global optimum regardless of starting point?

## 7.2. Learning objectives

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

- Formulate a statistical estimation problem as an optimisation of an objective function, and recognise when that objective is convex.
- State the Newton and quasi-Newton (BFGS) update rules and their cost per iteration.
- Use `optim()` with the main method options (`Nelder-Mead`, `BFGS`, `L-BFGS-B`, `CG`) and interpret their convergence diagnostics.
- Write a log-likelihood function and maximise it, handling the standard numerical issues (log-sum-exp, parameter transformations, scaling).

- Use `numDeriv::grad()` and `numDeriv::hessian()` to check an analytic gradient, and compute Wald standard errors from an observed Hessian.
- Diagnose common failure modes: non-convergence, boundary solutions, flat likelihoods, and identifiability problems.

## 7.3. Orientation

Likelihood-based inference is optimisation in disguise. Every MLE, every penalised regression, every mixed-effects fit, every Cox model, every neural network ultimately calls a general-purpose optimiser. Understanding what the optimiser is doing, and when it is failing, is the difference between producing results and producing plausible-looking noise.

The optimiser in R that covers most real cases is `optim()`. Underneath, it provides several algorithms. Choosing between them, writing an objective function that behaves well numerically, and diagnosing failures when they happen are the skills this chapter develops.

## 7.4. The statistician's contribution

Optimisation algorithms are mechanical. BFGS is BFGS; Newton-Raphson is Newton-Raphson. What is *not* mechanical, and what no optimiser can automate, is the judgement that surrounds the optimisation.

**Is this problem convex?** Convex problems have a unique global optimum; any descent method converges to it regardless of starting point. Non-convex problems have multiple local optima; the answer you get depends on where you started, and the 'answer' with the best objective value is not guaranteed to be the one you want. Before optimising, ask: is the likelihood concave (equivalently, is the negative log- likelihood convex)? For GLMs with canonical links, yes. For mixture models, neural networks, and many penalised problems, no. The statistician's job is to know which case they are in and to pick an optimisation strategy accordingly.

**Is the parameterisation sensible?** An optimiser sees the objective function you give it, nothing more. Parameters on wildly different scales (one

in units of $10^{-6}$, another in units of $10^{6}$) produce an ill-conditioned Hessian that makes every optimiser struggle. Parameters that should be positive (variances, rates) pass through `exp()`; probabilities pass through `plogis()`; correlations pass through Fisher's $z$. Working on the transformed scale makes the problem unconstrained and better-conditioned. The optimiser does not know or care; the analyst does.

**Is the objective numerically stable?** Naively-coded log- likelihoods overflow or underflow with alarming frequency when the data get large. The standard fixes, working on the log scale, log-sum-exp for mixtures, centring covariates — are the statistician's responsibility. An optimiser that diverges because the likelihood overflowed is not a bug in the optimiser.

**When has the optimiser actually converged?** `$convergence == 0` means 'the algorithm thinks it stopped because it was happy', not 'we have found the MLE'. Flat likelihoods, boundary solutions, and identifiability problems can all produce successful-looking output with wrong parameter estimates. Checking the Hessian for positive definiteness, starting from multiple initial values, and computing Wald standard errors are all defensive moves that verify convergence was genuine.

These judgement calls separate code that fits a model from code that fits a model defensibly. The optimiser handles the arithmetic; the statistician handles everything else.

## 7.5. Optimisation in statistical computing

Most statistical methods can be written as: choose parameters $\theta$ to minimise (or maximise) some objective $f(\theta)$.

- **Least squares.** $f(\beta) = \|y - X\beta\|^2$.
- **Maximum likelihood.** $f(\theta) = -\log L(\theta; y)$.
- **Penalised regression.** $f(\beta) = \|y - X\beta\|^2 + \lambda P(\beta)$ for ridge ($P = \|\beta\|^2$), lasso ($P = \|\beta\|_1$), elastic net (a mix).
- **K-means clustering.** Sum of within-cluster distances to centroids.
- **Neural networks.** Training loss plus regularisation.

Some of these have closed-form solutions (OLS, ridge). Most do not, and need iterative numerical optimisation.

R's core optimisation functions:

- `optimize(f, interval, ...)`, one-dimensional. Combines golden-section search and parabolic interpolation. Fast and reliable for univariate problems.
- `optim(par, fn, gr = NULL, method = "Nelder-Mead", ...)` — general-purpose, multidimensional. Default method is Nelder-Mead; supports BFGS, L-BFGS-B, CG, SANN, and Brent. Pass `method = "BFGS"` for smooth problems.
- `nlminb(start, objective, gradient = NULL, hessian = NULL, ...)` — box-constrained, uses an interior trust-region method. Often more robust than `optim()` for MLEs with many parameters.
- `uniroot(f, interval, ...)`, finds one-dimensional roots, useful for scoring equations.

For specialised problems, dedicated packages (`quadprog`, `nloptr`, `optimx`, `glmnet`) offer better performance or better convergence.

## 7.6. Convexity and why it matters

A function $f$ is *convex* if for any $x$, $y$, and $t \in [0, 1]$,

$$f(tx + (1-t)y) \leq tf(x) + (1-t)f(y).$$

Geometrically: the line segment connecting any two points on the graph lies above (or on) the graph. Equivalent conditions: for a twice-differentiable $f$, the Hessian is positive semi-definite everywhere.

Convex problems are well-behaved:

- Any local minimum is the global minimum.
- Descent methods converge to the global optimum regardless of starting point.
- Convergence rates are provably fast.

Statistical examples of convex problems:

- OLS (quadratic in $\beta$).

- Ridge regression ($L_2$-penalised OLS).
- Logistic regression (the log-likelihood is concave; the negative log-likelihood is convex).
- Poisson regression with log link.
- LASSO (convex but non-smooth; needs specialised methods).

Non-convex statistical examples: mixture models (multiple local maxima; EM gives a local optimum), neural networks (many local optima), most latent-variable models, and many penalised problems with non-convex penalties (SCAD, MCP).

Practical consequences:

- For convex problems, use BFGS or the problem-specific closed form. Single starting point is fine.
- For non-convex problems, use multiple starting points (random restarts or domain-informed starts) and report the best. Or use specialised algorithms (EM, stochastic methods).

Check your understanding: identifying convexity

**Question.** Which of these are convex optimisation problems: (a) OLS linear regression, (b) logistic regression MLE, (c) k-means clustering, (d) Gaussian mixture model MLE?
**Answer.**

(a) and (b) are convex: the OLS objective is quadratic with a positive-definite Hessian; the logistic log- likelihood is concave (so the negative log-likelihood is convex). (c) and (d) are not convex: k-means has many local optima corresponding to different centroid assignments; Gaussian mixture MLE has multiple local maxima (including a 'label-switching' symmetry plus genuinely different solutions). The practical implication: for (a) and (b), any reasonable optimiser gets the answer. For (c) and (d), starting point matters and multi-start is advisable.

## 7.7. Gradient descent

Gradient descent is the conceptual foundation for most continuous optimisation. The update rule for minimising $f(\theta)$:

$$\theta_{k+1} = \theta_k - \alpha_k \nabla f(\theta_k)$$

where $\alpha_k$ is the step size (learning rate). The gradient $\nabla f$ points in the direction of steepest *ascent*; the minus sign flips that to descent.

```
gradient_descent <- function(f, grad_f, theta0,
                             alpha = 0.01, n_iter = 100) {
  theta <- theta0
  for (i in seq_len(n_iter)) {
    theta <- theta - alpha * grad_f(theta)
  }
  theta
}
```

**Step size is critical.** Too large and the iterates oscillate or diverge; too small and convergence is impractically slow. For convex Lipschitz-smooth $f$ with Lipschitz constant $L$, the rule $\alpha \leq 1/L$ guarantees convergence. In practice, adaptive step sizes (backtracking line search, or methods like Adam that maintain per-parameter step sizes) are far more robust than a fixed $\alpha$.

**Convergence rate.** For convex smooth $f$, gradient descent converges at rate $O(1/k)$ in function values. For strongly convex $f$, the rate improves to linear. For ill- conditioned problems (large ratio of largest to smallest eigenvalue of the Hessian), progress zigzags in a narrow valley, and convergence can be painfully slow.

**Gradient descent in statistics is uncommon as the algorithm of choice** (Newton or quasi-Newton dominate for small-to-medium problems), but it is the foundation of stochastic gradient descent (SGD) and its variants (Adam, RMSProp, AdaGrad), which power nearly all modern deep-learning training. For large-sample statistical problems where a single gradient evaluation is expensive, SGD on mini-batches is often the right tool.

## 7.8. Newton's method

Newton's method uses curvature information (the Hessian matrix $H_f$ of second derivatives) in addition to the gradient:

$$\theta_{k+1} = \theta_k - H_f(\theta_k)^{-1} \nabla f(\theta_k).$$

The intuition: Newton's method approximates $f$ locally by a quadratic (the second-order Taylor expansion) and jumps to the minimum of that quadratic.

**Convergence rate:** quadratic, near a strong local optimum. This means the number of correct digits roughly doubles each iteration. For well-behaved problems, Newton converges in a handful of iterations.

**Costs:** each iteration requires computing the Hessian ($O(n^2)$ storage, $O(n^2)$ or more per element) and solving a linear system with it ($O(n^3)$). For $n$ in the tens or hundreds, fine; for $n$ in the thousands or more, prohibitive.

**Weaknesses:**

- Requires the Hessian. For complicated objectives, analytical Hessians are tedious to derive and code correctly.
- The Hessian must be positive definite at the current point. For non-convex problems, this fails; the 'step' may point uphill or toward a saddle. Practical implementations modify the Hessian (e.g., Levenberg-Marquardt adds a ridge) to ensure a descent direction.
- Far from the optimum, the quadratic approximation is unreliable, and a full Newton step may diverge. Line search or trust-region safeguards are standard.

**Fisher scoring** is a variant in which the Hessian is replaced by the expected information matrix (the Fisher information). This is what R's `glm()` uses internally: the observed-information Newton step and the Fisher-scoring step coincide for GLMs with canonical links. When they differ, Fisher scoring is often more stable.

## 7.9. Quasi-Newton: BFGS and L-BFGS

The BFGS method (Broyden-Fletcher-Goldfarb-Shanno) approximates the inverse Hessian from successive gradient differences. It is the default for `optim()` and is the workhorse for moderate-sized MLE problems.

The update maintains an approximation $B_k$ to the inverse Hessian, updated at each step using the change in parameters $s_k = \theta_{k+1} - \theta_k$ and the change in gradients $y_k = \nabla f(\theta_{k+1}) - \nabla f(\theta_k)$. The resulting $B_k$ remains positive definite (ensuring a descent direction) and converges to the true inverse Hessian near the optimum.

**Advantages over Newton:**

- No explicit Hessian computation. Only gradient evaluations.
- $O(n^2)$ per iteration instead of $O(n^3)$.
- Guaranteed descent direction via the positive-definiteness of $B_k$.

**Convergence rate:** superlinear (between linear and quadratic). In practice, often as fast as Newton on real problems and vastly easier to implement.

**L-BFGS** (limited-memory BFGS) stores only the last few (typically 3–20) pairs of $s_k$ and $y_k$ rather than the full $B_k$, reducing memory from $O(n^2)$ to $O(nm)$ where $m$ is the memory length. This makes it practical for very high-dimensional problems (millions of parameters). L-BFGS-B adds box constraints: $\ell_i \leq \theta_i \leq u_i$.

Check your understanding: BFGS vs. Newton

**Question.** Why is BFGS usually preferred to Newton's method in practice, even though Newton has faster asymptotic convergence?
**Answer.**
Three reasons. First, BFGS does not require computing or inverting the Hessian, which is either expensive ($O(n^3)$ per iteration) or requires code for the Hessian that is tedious and error-prone to derive. Second, the BFGS approximation is guaranteed positive definite, so the step is always a descent direction; Newton's raw step can point uphill in non-convex regions. Third, on real problems the BFGS approximation

converges to the true Hessian near the optimum, so the superlinear rate is almost as fast as Newton's quadratic rate in practice. The fast asymptotic rate of Newton's method matters most when you are already very close to the optimum; BFGS's robustness matters for getting there in the first place.

## 7.10. `optim()` in practice

`optim()` is the general-purpose optimiser:

```
result <- optim(
  par    = starting_values,
  fn     = objective_function,
  gr     = gradient_function,   # optional but recommended
  method = "BFGS",
  hessian = TRUE,               # return observed Hessian
  control = list(maxit = 200, reltol = 1e-10)
)
```

The output:

- `$par`: final parameter values.
- `$value`: objective at `$par`.
- `$convergence`: 0 means converged (by the algorithm's own stopping rules); nonzero means something else happened.
- `$counts`: function and gradient evaluations.
- `$message`: a human-readable explanation if convergence failed.
- `$hessian`: if requested, the observed Hessian at `$par`.

**Methods:**

- `"Nelder-Mead"`, the default. Simplex method; no gradients needed; robust to non-smooth or noisy objectives; slow.
- `"BFGS"`. Quasi-Newton, gradient-based. The preferred choice for smooth problems where the gradient is available; specify it explicitly with `method = "BFGS"`.

- `"L-BFGS-B"`, BFGS with box constraints (`lower = ..., upper = ...` in the call). Use when parameters have natural bounds.
- `"Nelder-Mead"`, simplex method; no gradients needed; robust to non-smooth or noisy objectives; slow.
- `"CG"`, conjugate gradient; low memory, can be slow.
- `"SANN"`, simulated annealing; for rough non-convex problems where you suspect local optima. Slow and approximate.

**Minimisation is the default.** To maximise (e.g., a log- likelihood), pass the *negative* of your objective, or set `control = list(fnscale = -1)`.

## 7.11. Writing a log-likelihood

For a sample $y_1, \dots, y_n$ iid from a density $p(y; \theta)$, the log-likelihood is

$$\ell(\theta) = \sum_{i=1}^{n} \log p(y_i; \theta).$$

Log-likelihoods are strongly preferred to likelihoods for three numerical reasons:

1. Products of many small probabilities underflow; sums of their logs do not.
2. The log transforms many statistical models into quadratic or near-quadratic objectives that optimise cleanly.
3. Derivatives of logs are simpler than derivatives of products.

A minimal example: maximum likelihood for the mean and variance of a normal sample.

```
neg_loglik_normal <- function(theta, y) {
  mu       <- theta[1]
  log_sigma <- theta[2]                # optimise on log scale
  sigma    <- exp(log_sigma)
  -sum(dnorm(y, mean = mu, sd = sigma, log = TRUE))
}
```

```r
set.seed(1); y <- rnorm(500, mean = 3, sd = 2)

fit <- optim(
  par     = c(0, 0),                  # start at mu=0, log_sigma=0
  fn      = neg_loglik_normal,
  y       = y,
  method  = "BFGS",
  hessian = TRUE
)

c(mu_hat    = fit$par[1],
  sigma_hat = exp(fit$par[2]))
mean(y); sd(y)                        # compare
```

Three idioms in this code are worth internalising.

**1. `log = TRUE` in density functions.** `dnorm(y, log = TRUE)` returns $\log p(y)$ directly, avoiding underflow when probabilities are tiny.

**2. Optimise on the log scale for positive parameters.** We parameterised by $\log \sigma$ instead of $\sigma$ so the optimiser works on an unconstrained space ($\sigma > 0$ becomes $\log \sigma \in \mathbb{R}$). This is a universal trick: log for positive parameters, logit for probabilities, Fisher $z$ for correlations.

**3. Pass data via `...`.** `optim(par, fn, y = y)` passes `y` to `fn` without making it a global variable. For reproducibility and testability, this is substantially cleaner than closures over global state.

## 7.12. Gradients: analytic, numerical, automatic

Most optimisers work better with analytic gradients than with numerical approximations. Numerical gradients (finite differences) are noisy and expensive.

**Analytic.** Derive by hand; code carefully. Always check against a numerical gradient before trusting it:

```r
library(numDeriv)

grad_analytic <- function(theta, y) {
  # ... analytical gradient ...
}

theta0 <- c(0, 0)
grad_num <- numDeriv::grad(neg_loglik_normal, theta0, y = y)
grad_an  <- grad_analytic(theta0, y)
max(abs(grad_num - grad_an))         # should be ~1e-8
```

If analytic and numerical disagree by more than rounding, there is a bug. This is non-negotiable: silently wrong gradients produce silently wrong parameter estimates, and the optimiser will cheerfully converge to the wrong answer.

**Numerical.** `optim()` falls back to finite differences if `gr = NULL`. Acceptable for small problems; slow and imprecise for large ones.

**Automatic differentiation.** Packages like `torch`, `tensorflow`, and `TMB` compute exact derivatives from the computation graph. For complicated likelihoods, auto-diff is often easier than hand-derived gradients and just as fast. In base R, auto-diff is not standard; in Stan (via `rstan`), it is the default.

## 7.13. Wald standard errors from the Hessian

For an MLE $\hat{\theta}$, the variance-covariance matrix is asymptotically the inverse of the observed information matrix:

$$\widehat{\mathrm{Var}}(\hat{\theta}) = H^{-1}$$

where $H$ is the Hessian of the *negative* log-likelihood at $\hat{\theta}$. `optim(..., hessian = TRUE)` returns this matrix.

```
vcov_hat <- solve(fit$hessian)
se <- sqrt(diag(vcov_hat))
se
```

Wald $z$-statistics and confidence intervals follow directly. Two caveats:

- The Hessian must be positive definite for this to make sense. If any diagonal of `vcov_hat` is negative, or if `chol()` on `fit$hessian` fails, the optimiser did not find a proper maximum.
- The parameterisation matters. If you optimised $\log \sigma$ and want an SE for $\sigma$, apply the delta method: $\mathrm{SE}(\sigma) \approx \sigma \cdot \mathrm{SE}(\log \sigma)$.

## 7.14. Common failure modes

Five failure modes account for most optimisation bugs:

**1. Non-convergence.** `fit$convergence != 0`, or reached `maxit`. Try: more iterations, a different method, better starting values, scale parameters so they are all $O(1)$.

**2. Boundary solutions.** The MLE is at (or close to) the boundary of the parameter space, a variance hitting zero, a probability hitting 1. Wald inference breaks down; profile likelihoods or bootstrap are more reliable.

**3. Flat likelihood.** The likelihood is nearly constant along some direction in parameter space. The optimiser wanders; the Hessian is nearly singular. Usually signals non-identifiability (too many parameters for the data to determine). Remove the redundant parameter or add a prior.

**4. Multiple starting points give different answers.** The problem is non-convex. Run from many starts; take the best. Or switch to a method designed for non-convex problems (EM, simulated annealing, stochastic methods).

**5. Numerical overflow/underflow.** Probabilities of zero, infinite likelihoods. Usually caused by naively coded likelihoods that forgot to work on the log scale, or by categorical models with a predictor that perfectly separates the response (Firth correction or penalised likelihood is the standard fix).

Diagnostic ritual before trusting any `optim()` result:

1. Check `fit$convergence == 0`.
2. Check `fit$message` if it is nonzero.
3. Compute `eigen(fit$hessian)$values`. All positive? Good. Any negative or near-zero? Suspect.
4. Re-run from two or three random starting points. Same answer? Good. Different answers? Non-convex or multiple optima.
5. Compare against a closed form or a known-good implementation if available.

Check your understanding: diagnosing failure

**Question.** `optim()` reports `$convergence = 0` and returns parameters, but the Hessian has one negative eigenvalue. What likely went wrong, and what would you do?
**Answer.**
`$convergence = 0` only means the optimiser's internal stopping criterion fired; it does not mean a true minimum was found. A negative eigenvalue of the Hessian at the reported optimum means the point is a saddle, not a minimum: the objective decreases in at least one direction. Likely causes: the optimiser stopped early because of a flat region, or the problem is non-convex and the optimiser is lodged near a saddle. Remedies: try a different starting point, increase `maxit` and decrease `reltol`, or switch methods (e.g., from BFGS to Nelder-Mead or a trust-region method in `nlminb`). Until the Hessian is positive definite, do not trust the reported parameter values.

## 7.15. Collaborating with an LLM on optimisation

Optimisation is a domain where LLMs can produce working code quickly and also produce silently wrong code quickly. Three patterns work well.

**Prompt 1: writing a log-likelihood.** Describe the model (e.g., 'a gamma regression with log link'), provide a data frame with column names, and ask: 'write the negative log- likelihood as a function `nll(theta, data)`

using a log-parameterised rate, and return a call to `optim` that maximises it.'

*What to watch for.* Common mistakes: missing the Jacobian of a parameter transformation, forgetting to use `log = TRUE` in `dgamma()`, hard-coded column names that do not match the data, returning the positive instead of negative log-likelihood for a minimiser. Review every line.

*Verification.* Pick a case where the MLE has a closed form or where a canonical R implementation exists (e.g., `glm(..., family = Gamma(link = "log"))`), fit with the LLM- generated code, and check that the coefficients and SEs agree to several digits. If they disagree, the LLM's code is wrong.

**Prompt 2: deriving a gradient.** Paste the log-likelihood function and ask: 'derive the analytical gradient and write it as a function `grad(theta, data)` that returns a vector of the same length as `theta`.'

*What to watch for.* LLMs are often confidently wrong about derivatives, especially when the likelihood involves a parameter transformation. Treat the output as a draft.

*Verification.* Always check against a numerical gradient:

```
theta0 <- c(0.1, 0.2, -0.3)
grad_num <- numDeriv::grad(nll, theta0, data = dat)
grad_an  <- grad_function(theta0, data = dat)
max(abs(grad_num - grad_an))
```

Difference should be at most $10^{-6}$ or so. Anything larger is a bug.

**Prompt 3: diagnosing non-convergence.** Describe what you tried (method, starting values, convergence code and message, eigenvalues of the Hessian) and ask: 'what is likely going wrong, and what would you try next?'

*What to watch for.* LLMs are reasonable at pattern-matching classic failure modes (boundary solutions, flat likelihoods, separation in logistic regression). They are less good at novel data-specific problems. Treat the suggestions as a checklist, not as the answer.

*Verification.* Whatever fix is suggested, try it and check whether the Hessian is positive definite afterwards. A 'fix' that gets `$convergence = 0` but leaves the Hessian indefinite has not actually fixed the problem.

## 7.16. Worked example: logistic regression from scratch

```
set.seed(1)
n <- 500
X <- cbind(1, matrix(rnorm(n * 2), n, 2))
beta_true <- c(-1, 0.5, -0.3)
p <- plogis(X %*% beta_true)
y <- rbinom(n, 1, p)

neg_loglik_logistic <- function(beta, X, y) {
  eta <- X %*% beta
  # stable: sum(y * eta - log(1 + exp(eta)))
  -sum(y * eta - log1p(exp(eta)))
}

fit <- optim(
  par    = rep(0, ncol(X)),
  fn     = neg_loglik_logistic,
  X      = X,
  y      = y,
  method = "BFGS",
  hessian = TRUE
)

# compare to glm()
beta_glm <- coef(glm(y ~ X - 1, family = binomial))
cbind(optim = fit$par, glm = beta_glm)

# Wald SEs from observed Hessian
```

```
se <- sqrt(diag(solve(fit$hessian)))
se
```

Two idioms worth memorising:

- `log1p(exp(eta))` instead of `log(1 + exp(eta))`. For large negative `eta`, `1 + exp(eta)` is numerically equal to 1 and the log is zero; `log1p` preserves the precision. For very large positive `eta`, both overflow; the standard log-sum-exp trick (not shown here for brevity) is the robust fix.
- `sum(y * eta - log1p(exp(eta)))` is the stable form of the logistic log-likelihood, avoiding the formation of `plogis(eta)` and subsequent log.

## 7.17. Principle in use

Three habits characterise effective use of optimisation in R:

1. **Think before optimising.** Is the problem convex? Is the parameterisation sensible? Is the objective numerically stable? The optimiser is only as good as the problem you hand it.
2. **Verify convergence.** `$convergence = 0` is necessary but not sufficient. Check the Hessian. Run from multiple starting points. Compare against a canonical implementation when possible.
3. **Check gradients.** If you wrote an analytic gradient, compare it to `numDeriv::grad()`. If they disagree, the analytic one is wrong. A wrong gradient will silently mislead the optimiser into the wrong answer.

These three habits catch nearly all the ‘I ran `optim()` and got a weird answer’ bugs.

## 7.18. Exercises

1. Maximise the log-likelihood of a normal sample by hand using Newton's method. Start from the sample mean and SD plus noise; confirm convergence in three iterations.
2. Use `optim(..., hessian = TRUE)` on a logistic regression likelihood and use the Hessian to compute Wald standard errors. Compare to `summary(glm_fit)`.
3. Construct a simple non-identifiable model (e.g., $y \sim x_1 + x_2 + x_3$ with $x_3 = x_1 + x_2$). Show that `optim()` fails to converge and explain what the flat likelihood looks like (inspect `fit$hessian`).
4. For a two-parameter Gamma MLE, code the negative log- likelihood and its analytic gradient. Verify the gradient against `numDeriv::grad()`. Then fit it with `optim()` and compare to `MASS::fitdistr(x, "gamma")`.
5. Take a logistic regression where the response is perfectly separated by a predictor (all $y = 1$ when $x > 0$, all $y = 0$ otherwise). Fit with `glm()` and observe the warning. Fit with `brglm::brglm()` (Firth correction) and explain why the Firth version produces a finite coefficient.

## 7.19. Further reading

- (Nocedal & Wright, 2006), the modern reference; Chapters 1–6 cover the methods behind `optim()`.
- (Boyd & Vandenberghe, 2004), the definitive convex optimisation text; free online at https://web.stanford.edu/~boyd/cvxbook.
- (Lange, 2010), written for statisticians, with EM and MM alongside classical methods.
- (Hastie et al., 2015), grounds penalised optimisation in statistical learning.

## 7.20. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 7.20.1. Question 1

What does gradient descent do in each iteration to find an optimum?

- A) Computes random points to explore the function space
- B) Takes a step in the direction of the gradient
- C) Takes a step in the direction opposite to the gradient
- D) Evaluates the function at fixed intervals

> **i** Answer
>
> **C.** The gradient points uphill; to minimise, move in the opposite direction.

### 7.20.2. Question 2

When using gradient descent to minimise a function, what determines the direction of each step?

- A) The negative gradient (pointing downhill)
- B) The positive gradient (pointing uphill)
- C) A random direction to avoid local minima
- D) The second derivative of the function

> **i** Answer
>
> **A.** The negative gradient is the direction of steepest descent.

### 7.20.3. Question 3

Which of the following is a common challenge when using gradient descent with a fixed learning rate?

- A) The algorithm always converges too quickly
- B) It requires calculating second derivatives
- C) If the learning rate is too large, the algorithm may diverge
- D) It can only be used for one-dimensional problems

> **i** Answer
>
> **C.** Large steps can overshoot the minimum and amplify instead of reducing the objective.

### 7.20.4. Question 4

Which of the following statements about gradient descent is TRUE?

- A) It requires computing and inverting the Hessian matrix at each iteration
- B) It always converges to the global minimum regardless of the starting point
- C) It updates parameters by moving in the direction of steepest descent
- D) It converges in a single step for quadratic functions

> **i** Answer
>
> **C.** Steepest descent uses only first-derivative (gradient) information, not the Hessian.

### 7.20.5. Question 5

In the context of optimisation algorithms, what is the primary advantage of BFGS over Newton's method?

- A) BFGS guarantees finding the global optimum for non-convex functions
- B) BFGS requires fewer iterations to reach the optimum
- C) BFGS approximates the Hessian without explicitly computing second derivatives
- D) BFGS works better for non-differentiable objective functions

> **Answer**
>
> **C.** BFGS builds an approximation to the inverse Hessian from successive gradient differences, avoiding the cost and numerical trouble of forming a true Hessian.

### 7.20.6. Question 6

Which type of optimisation problem is characterised by having a unique global optimum?

- A) Non-smooth optimisation problems
- B) Convex optimisation problems
- C) Non-convex optimisation problems
- D) Constrained optimisation problems

> **Answer**
>
> **B.** Convexity implies any local minimum is also the unique global minimum.

### 7.20.7. Question 7

`optim()` returns `$convergence = 0`, but when you compute `eigen(fit$hessian)$v` you see one negative eigenvalue. The best interpretation is:

- A) The answer is correct; `$convergence = 0` is definitive.
- B) The optimiser found a saddle point or is stuck in a flat region, and the parameter estimates should not be trusted.

- C) `optim()` sometimes reports wrong `convergence` codes; ignore the Hessian.
- D) Round the parameters and report them as the MLE.

**i** Answer

**B.** A true minimum has a positive-definite Hessian. A negative eigenvalue indicates a saddle or an indefinite region. Re-run from different starting values or with a different method before trusting the output.

## 7.21. Prerequisites answers

1. Each step moves in the direction *opposite* to the gradient (the negative gradient direction), because the gradient points in the direction of steepest *ascent* and we want to descend.
2. The iterates may overshoot the minimum and diverge. Practical implementations use a line search or backtracking to adapt the step size to the local curvature. Fixed learning rates that work for one problem often fail for another.
3. BFGS approximates the inverse Hessian from successive gradient differences, avoiding the $O(n^3)$ cost of forming and inverting a Hessian each iteration, and guaranteeing a descent direction (because the approximation is positive definite). A *convex* optimisation problem has a unique global optimum, so any descent method converges to the same point regardless of starting value.

# Part III.

# Computational Statistics

# 8. Simulation Study Design

## 8.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 8.20.

1. What is the primary reason for calling `set.seed()` at the start of a simulation?
2. In a Monte Carlo simulation, how does the standard error of a simulated quantity change as the number of replications increases from $R$ to $4R$?
3. When comparing two statistical methods by simulation, what is the benefit of using common random numbers across methods?

## 8.2. Learning objectives

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

- State what a simulation study can and cannot establish.
- Design a simulation with a clear data-generating mechanism, estimands, methods, and performance measures (the ADEMP framework).
- Use `set.seed()` and `RNGkind()` correctly to produce reproducible output.
- Generate random variates from common parametric families with `r*()` functions and from multivariate normal distributions via Cholesky or `MASS::mvrnorm`.
- Assess Type-I error, power, coverage, and bias for a proposed method, and report each with a Monte Carlo standard error.

- Use common random numbers across competing methods to reduce variance in a simulation comparison.
- Recognise when parallelisation is worth the overhead, and use `parallel::clusterSetRNGStream()` for reproducible parallel simulations.

## 8.3. Orientation

A simulation study is an experiment in which you know the truth. You specify a data-generating mechanism (so you know the true parameter values), run a method on the simulated data, and compare the method's output to the known truth. Iterating across many replicates produces estimates of how the method behaves.

Well-designed simulations tell you whether a proposed method controls error rates, is efficient, and is robust to violations of its assumptions. Badly designed simulations mislead, and the misleading conclusions often look convincing because the methodology has the superficial shape of an experiment. This chapter is about how to tell the two apart.

The basic ingredients are always the same: a data-generating mechanism (DGM) under your control, an estimand (the true quantity you want to estimate), one or more methods (the procedures you are evaluating), and performance measures (bias, variance, coverage, error rates, computational cost). The ADEMP framework assembles these into a reporting structure that has become the de facto standard in biostatistical methodology papers.

## 8.4. The statistician's contribution

Simulation studies are not automation-friendly. An LLM can generate a simulation loop quickly; what it cannot generate is the research question that makes the simulation meaningful or the adversarial checks that make its conclusions trustworthy.

**Ask the right question.** The most common simulation failure is not a bug in the code; it is a simulation that does not address any question worth

asking. 'Does method A outperform method B?' is rarely a precise enough question. 'At what sample size, under what distributional conditions, for what effect size, does method A control Type-I error while method B does not?' is specific enough to be answerable. The sharpening of question is the statistician's job.

**Make the DGM honest.** A simulation in which every assumption of the method is exactly met will make every reasonable method look great. The interesting question is robustness: what happens when assumptions are violated in ways the method's theory does not cover? Heavy-tailed errors, heteroscedasticity, missing data, measurement error, contamination, these are what real data bring. A simulation that omits them proves the method works on simulation data, which is a weaker claim than it sounds.

**Pick performance measures that match the claim.** If the paper's claim is 'method A has correct coverage', measure coverage. If it is 'method A has smaller MSE', measure MSE. If it is 'method A is robust to contamination', measure bias under contamination. Measuring power when the claim is about coverage is a category error that peer reviewers miss more often than they should.

**Report Monte Carlo uncertainty.** Every simulation result is an estimate based on $R$ replicates. It has its own standard error. Reporting 'method A's power is 0.82 vs. method B's 0.81' without the Monte Carlo SE (about 0.013 at $R = 1000$) is dishonest: the difference is within the simulation noise. Reporting the SE makes this visible.

**Resist the temptation to keep simulating until the answer looks right.** Pre-register the design, the number of replicates, and the performance measures before running the simulation. Simulation experiments are as susceptible to garden-of-forking-paths bias as real experiments.

These judgements are what separate a simulation that could have been a data dredge from a simulation that survives adversarial review. An LLM will happily execute a poorly designed simulation as efficiently as a well-designed one.

## 8.5. The ADEMP framework

Morris, White, and Crowther (2019) propose ADEMP as the standard reporting framework for simulation studies:

- **A**ims. What question is the simulation answering?
- **D**ata-generating mechanism. How are the simulated datasets created? What distributions, parameter values, sample sizes, and correlation structures are used?
- **E**stimand. What true quantity is the method trying to estimate? Be explicit: a regression coefficient, a treatment effect, a tail probability, a coverage probability.
- **M**ethods. Which procedures are being evaluated? State each method precisely enough that another researcher could reproduce it.
- **P**erformance measures. Which properties are being measured? Bias, mean squared error, coverage of a $(1 - \alpha)$ confidence interval, Type-I error rate, power at specific alternatives. Each should be quantified with a Monte Carlo standard error.

An ADEMP-structured methods section makes a simulation study reviewable; a narrative description that conflates the components is much harder to assess. When designing your own simulation, write the ADEMP specification *before* writing any code. When reading someone else's simulation, map their description onto ADEMP; gaps in their exposition usually point to weaknesses in their design.

## 8.6. Random number generation in R

R generates pseudorandom numbers with the Mersenne Twister by default, a high-quality generator with period $2^{19937} - 1$, vastly longer than any simulation will ever consume. The generator is deterministic: given the same seed, it produces the same sequence of draws.

```
set.seed(42)
runif(3)
#> [1] 0.9148060 0.9370754 0.2861395
```

```
set.seed(42)
runif(3)
#> [1] 0.9148060 0.9370754 0.2861395   # identical
```

`set.seed(42)` initialises the generator's state to a deterministic value derived from the integer 42. Subsequent calls to random functions draw from the resulting sequence. For reproducible simulations, call `set.seed()` at the beginning.

**Reproducibility in the face of refactoring.** A simulation that gives results 'exactly' as reported in the paper requires that (a) R's default RNG has not changed between the original run and the reproduction, (b) the order of RNG calls is preserved, and (c) the simulation was run with the same number of replicates. Bit-identical reproduction across R versions is not guaranteed; approximate reproduction (results within Monte Carlo noise) is the more realistic goal.

**`.Random.seed`** is a special variable holding the current state of the RNG. You can save and restore it:

```
saved_state <- .Random.seed
x <- runif(1)
.Random.seed <- saved_state
y <- runif(1)           # identical to x
```

This is occasionally useful when you want two independent replicates to share a specific random draw, or when debugging a simulation that fails on a particular replicate.

**`RNGkind()`** selects the generator. For parallel simulations, `RNGkind("L'Ecuyer-CMRG")` is recommended because it provides independent streams across workers (see the parallelisation section below). Changing `RNGkind` changes the sequence even for the same seed, so document it if you use anything other than the default.

Check your understanding: set.seed before a simulation

**Question.** Why call `set.seed()` at the start of a simulation, and what does setting the seed *not* guarantee?
**Answer.**
`set.seed()` fixes the pseudorandom stream, so running the simulation twice produces identical output, essential for reproducibility of tables, figures, and results reported in a paper. It does *not* guarantee that different R versions will produce identical output (RNG implementations have occasionally changed), that different operating systems will agree bit-for-bit, or that a simulation whose code calls random functions in a different order will produce the same results. For cross-version reproducibility, record the R version, the `RNGkind()`, and the seed, and store a copy of the simulated data or results alongside the code.

## 8.7. Generating random variates

R's `r*()` functions cover the common distributions:

```
runif(n)                          # uniform
rnorm(n, mean, sd)                # normal
rbinom(n, size, prob)             # binomial
rpois(n, lambda)                  # Poisson
rexp(n, rate)                     # exponential
rgamma(n, shape, rate)            # gamma
rbeta(n, shape1, shape2)          # beta
rt(n, df)                         # t
rchisq(n, df)                     # chi-square
rf(n, df1, df2)                   # F
rmultinom(n, size, prob)          # multinomial
```

Each has a corresponding `d*()` (density), `p*()` (distribution function), and `q*()` (quantile function).

**Inverse transform.** For any distribution with a computable quantile function, if $U \sim \text{Uniform}(0, 1)$, then $F^{-1}(U)$ has distribution $F$. This is

why `qnorm(runif(n))` is equivalent to `rnorm(n)`. For custom distributions where only `qdist` is available, the inverse transform is the standard constructor.

**Acceptance-rejection.** For distributions without a convenient quantile function, generate a candidate from a proposal distribution and accept with probability equal to the ratio of target to envelope densities. R's internals use specialised acceptance-rejection schemes for many of the standard distributions.

**Custom distributions.** If you need to sample from a density $f$ you can evaluate (up to a constant), MCMC is usually more reliable than hand-rolling rejection sampling. `MCMCpack`, `rstan`, and `nimble` are the standard tools.

## 8.8. Multivariate normal

For a covariance matrix $\Sigma$ and mean $\mu$, a draw from $N(\mu, \Sigma)$ is constructed as $\mu + Lz$ where $L$ is the Cholesky factor of $\Sigma$ and $z \sim N(0, I)$:

```
Sigma <- matrix(c(1, 0.5, 0.5, 1), 2, 2)
mu    <- c(0, 1)
L     <- t(chol(Sigma))                # lower triangular
z     <- matrix(rnorm(2 * 1000), 1000, 2)
x     <- matrix(mu, 1000, 2, byrow = TRUE) + z %*% t(L)
```

`MASS::mvrnorm(n, mu, Sigma)` wraps this (using eigendecomposition by default for better behaviour on near-singular $\Sigma$). For large simulations where $\Sigma$ is fixed across replicates, factor once and reuse $L$ rather than re-factoring every iteration.

## 8.9. Designing a simple simulation

A concrete example: compare the t-test and the Wilcoxon rank-sum test under log-normal data.

**A.** Does the t-test remain valid when the outcome is log-normal with a mean shift? How does its power compare to the Wilcoxon rank-sum test?

**D.** Two independent samples of size $n$, one from Lognormal$(0, 1)$, one from Lognormal$(\delta, 1)$ for $\delta \in \{0, 0.5, 1\}$. Sample sizes $n \in \{20, 50, 200\}$.

**E.** The difference in means (corresponding to a location shift of $\delta$ on the log scale). Under $\delta = 0$, the null hypothesis is true.

**M.** The two-sided two-sample t-test with `t.test()` and the Wilcoxon rank-sum test with `wilcox.test()`, each at $\alpha = 0.05$.

**P.** Type-I error (rejection rate under $\delta = 0$) and power (rejection rate under $\delta > 0$).

```
simulate_one <- function(n, delta) {
  y1 <- rlnorm(n, meanlog = 0,      sdlog = 1)
  y2 <- rlnorm(n, meanlog = delta,  sdlog = 1)
  c(
    t_reject = t.test(y1, y2)$p.value       < 0.05,
    w_reject = wilcox.test(y1, y2)$p.value  < 0.05
  )
}

run_scenario <- function(n, delta, R = 5000) {
  results <- replicate(R, simulate_one(n, delta))
  rowMeans(results)
}

design <- expand.grid(
  n     = c(20, 50, 200),
  delta = c(0, 0.5, 1)
)

set.seed(1)
design$results <- Map(run_scenario, design$n, design$delta)
```

This is a 3×3 factorial design with 5000 replicates per cell. At $\delta = 0$, the 'rejection rate' estimates the Type-I error; at $\delta > 0$, it estimates the power.

## 8.10. Monte Carlo error

Every simulation output is an estimate. For a rejection rate $\hat{p}$ estimated from $R$ independent replicates, the standard error is

$$\mathrm{SE}(\hat{p}) = \sqrt{\hat{p}(1-\hat{p})/R}.$$

At $R = 1000$ and $\hat{p} = 0.80$, this is about 0.013. Differences in power smaller than roughly 0.03 are within Monte Carlo noise and should not be reported as real.

For continuous outputs (bias, MSE), the Monte Carlo SE is $s/\sqrt{R}$ where $s$ is the SD of the per-replicate estimates.

**How many replicates?** Target the precision you need. For a Type-I error of 0.05, a Monte Carlo SE of 0.005 requires about $R = 2000$. For detecting power differences of 0.01, you need $R \approx 20{,}000$. Report whatever $R$ you used, and always report the Monte Carlo SE.

Check your understanding: Monte Carlo SE scaling

**Question.** At $R = 1000$ the Monte Carlo SE of an estimated power of 0.8 is about 0.013. What $R$ would reduce the SE to 0.005?
**Answer.**
SE $\propto 1/\sqrt{R}$. Reducing the SE by a factor of $0.013/0.005 = 2.6$ requires increasing $R$ by a factor of $2.6^2 \approx 6.8$. So $R \approx 6800$. More generally, doubling precision requires quadrupling replicates, which is the same inverse-square-root law that governs all Monte Carlo error. Plan the computational budget before running: if $R = 10^6$ replicates would take a week on your laptop, reconsider the target precision.

## 8.11. Common random numbers

When comparing two methods on the same simulated dataset, use *the same* simulated dataset for both methods. This 'common random numbers' (CRN) scheme induces positive correlation between the two methods' estimates, reducing the variance of their difference:

$$\mathrm{Var}(\hat{\theta}_A - \hat{\theta}_B) = \mathrm{Var}(\hat{\theta}_A) + \mathrm{Var}(\hat{\theta}_B) - 2\mathrm{Cov}(\hat{\theta}_A, \hat{\theta}_B).$$

If the methods are applied to independent datasets, the covariance is zero. If they are applied to the same dataset, the covariance is positive (both estimates see the same noise), and the variance of the difference drops.

Practically: generate one dataset per replicate; apply both methods to it. The pattern already appears in the `simulate_one` function above.

CRN is essentially the simulation analogue of a paired design in experimental statistics. The variance reduction can be dramatic, factors of 5 or more are common for methods that are correlated in their responses to the data.

Check your understanding: common random numbers

**Question.** You want to compare the power of method A vs. method B. Why is generating one dataset per replicate and applying both methods to it better than generating two separate datasets (one for each method)?
**Answer.**
Because the methods see the same random noise, their rejection decisions are positively correlated. The variance of the difference in their rejection rates is therefore smaller than it would be under independent datasets. For a given $R$, CRN produces a tighter estimate of the difference. This is the Monte Carlo analogue of a paired study: comparing within a unit is more powerful than comparing across units when the unit-level variability is large. The variance reduction is often a factor of 3–10 in practice, buying the equivalent of many thousands of extra replicates for free.

## 8.12. Parallelisation

Simulation studies are ‘embarrassingly parallel’: each replicate is independent of the others. Scaling to multiple cores is straightforward and usually produces near-linear speedup until the number of workers approaches the number of physical cores.

```
library(parallel)

RNGkind("L'Ecuyer-CMRG")              # supports independent streams
set.seed(1)

cl <- makeCluster(4)
clusterSetRNGStream(cl, iseed = 1)    # reproducible, independent str

clusterExport(cl, c("simulate_one"))

results <- parSapply(cl, seq_len(5000),
                     function(i) simulate_one(n = 50, delta = 0.5))

stopCluster(cl)

rowMeans(results)
```

Two things are going on here:

1. `RNGkind("L'Ecuyer-CMRG")` plus `clusterSetRNGStream` set up a generator that can produce independent, non- overlapping streams for each worker. Without this, different workers may produce correlated (or identical) sequences, silently biasing results.
2. `clusterExport` copies named objects to each worker. Functions and data the workers will need must be exported.

For modern code, `future`, `future.apply`, and `furrr` provide a cleaner interface that abstracts over sequential/parallel/distributed execution:

```
library(future.apply)
plan(multisession, workers = 4)
results <- future_sapply(seq_len(5000),
                         function(i) simulate_one(n = 50, delta = 0.5
                         future.seed = TRUE)
```

`future.seed = TRUE` handles the RNG setup automatically, which is the main source of bugs in parallel simulations.

**When is parallelisation worth it?** The overhead of starting workers and shipping data is non-trivial. For a replicate that takes milliseconds, parallelising 5000 replicates across 4 cores may be slower than the serial version. For a replicate that takes seconds or more, parallelisation scales well. A rule of thumb: if your serial simulation takes more than about 10 minutes, parallelisation is worth setting up.

## 8.13. Reporting simulation results

Simulation output is high-dimensional: methods × scenarios × performance measures × replicates. Effective reporting extracts the headline findings into a small number of carefully designed figures.

**Faceted line plots** work well for performance-by- sample-size curves. Sample size on x-axis, performance metric on y-axis, methods as colours, scenarios as facets. Add error bars or confidence bands for Monte Carlo error.

**Heatmaps** work well for two-factor designs with many levels. Scenario on one axis, method on another, colour encodes performance.

**Tables** complement figures when precise values matter. Report performance measures and Monte Carlo SEs side by side. Do not report differences smaller than the Monte Carlo SE as if they were real.

An often-neglected piece: explicitly report what was simulated (ADEMP), how many replicates, the RNG used, the seed, and the compute environment (R version, OS, number of cores). This information is what makes the simulation reproducible.

## 8.14. Worked example: coverage of a confidence interval

A classical use case: does a 95% confidence interval achieve its nominal coverage?

```r
simulate_coverage <- function(n, R = 5000) {
  set.seed(42)
  covered <- logical(R)
  for (i in seq_len(R)) {
    x <- rnorm(n, mean = 3, sd = 2)
    ci <- t.test(x)$conf.int
    covered[i] <- ci[1] <= 3 & 3 <= ci[2]
  }
  cov_rate <- mean(covered)
  mcse <- sqrt(cov_rate * (1 - cov_rate) / R)
  c(coverage = cov_rate, mcse = mcse)
}

simulate_coverage(n = 30)
#> coverage     mcse
#>    0.949    0.003
```

A coverage rate within about 2 Monte Carlo SEs of the nominal 0.95 is consistent with correct coverage. Substantial deviations flag a problem, either an incorrect CI procedure, a violation of assumptions the CI requires, or a bug in the simulation code itself.

## 8.15. Collaborating with an LLM on simulation design

Simulation studies are a domain where LLMs can generate working code very quickly, and where the most important errors are design errors the LLM would not flag.

**Prompt 1: drafting an ADEMP specification.** Describe the question in plain English and ask: 'write an ADEMP specification for a simulation study addressing this question. Make every component explicit.'

*What to watch for.* The output will usually be well-structured but may pick default choices (sample sizes, effect sizes, distributions) that do not match the scientific question. Treat it as a draft. In particular, the 'D' (data-generating mechanism) is where most simulations go wrong: make sure the DGM reflects the real data, not a convenient idealisation.

*Verification.* Have a second person (or yourself a day later) read the ADEMP and predict what the simulation will conclude. If the prediction is obvious from the DGM, the simulation is not adding information. If the prediction is unclear, the simulation is worth running.

**Prompt 2: writing the simulation loop.** Paste the ADEMP and ask: 'write an R implementation using `set.seed()`, parallel execution via future.apply, and returning a tibble with one row per (scenario, replicate, method).'

*What to watch for.* The loop will probably be correct for the serial case. For the parallel case, watch for the RNG setup: the L'Ecuyer-CMRG generator and `future.seed = TRUE` are the standard incantation, and LLMs sometimes forget them. Silent RNG correlation across workers is a classic failure mode.

*Verification.* Run the simulation with a small $R$ and print intermediate results. Check: are the scenarios all there? Do both methods appear? Is the output shape right? Then scale up.

**Prompt 3: computing Monte Carlo SEs.** Paste the summary output and ask: 'compute the Monte Carlo standard error for each performance measure. For proportions use $\sqrt{p(1-p)/R}$; for continuous measures use the empirical SD divided by $\sqrt{R}$.'

*What to watch for.* LLMs sometimes produce confidence intervals using the wrong formula (e.g., treating coverage like a continuous measurement and using empirical SD). Match the formula to the type of quantity.

*Verification.* For a proportion, the MCSE can be double-checked against `binom.test` or a bootstrap. For a continuous measure, the MCSE is the sample SD of the per-replicate estimates, divided by $\sqrt{R}$.

The meta-lesson: an LLM can write a simulation loop faster than you can, but it cannot ensure that the simulation is asking a useful question or that the conclusions are robust to Monte Carlo noise. Those remain the analyst's job.

## 8.16. Principle in use

Three habits define a defensible simulation study:

1. **Write the ADEMP before writing code.** Aims, DGM, estimand, methods, performance, explicit in writing. If any of these is vague, sharpen before continuing.
2. **Report Monte Carlo standard errors for every performance measure.** Differences smaller than 2 MCSEs are noise. Make the noise visible.
3. **Treat the DGM as where the interesting judgement lives.** Simulating only idealised data proves only that a method works on idealised data. Include violations of assumptions the method depends on.

## 8.17. Exercises

1. Design a simulation to assess the Type-I error of a one-sample t-test under three distributions: normal, log-normal, and $t_3$. Report error rates with Monte Carlo SEs for each.
2. Extend exercise 1 to a $3 \times 3$ factorial design (sample size $n \in \{20, 100, 500\}$; distribution $\in \{\text{normal}, \text{log-normal}, t_3\}$). Present results as a `ggplot2` faceted plot.
3. Use `set.seed()` to produce a simulation whose output is bit-reproducible. Change your R version (or run on a different OS) and confirm that results remain identical.
4. Run the t-test vs. Wilcoxon simulation from this chapter with and without common random numbers. Compare the Monte Carlo SE of the difference in power. How large is the variance reduction?
5. Parallelise the simulation from exercise 2 using `future.apply::future_sapply`. Verify that you get the same results (within Monte Carlo noise) as the serial version. Time both. At what $R$ does parallel become faster?

## 8.18. Further reading

- (Morris et al., 2019), the ADEMP framework is the standard for biostatistical simulation studies.
- (Rizzo, 2019), book-length coverage of statistical computing in R.
- Pawel et al. on simulation study reporting , concrete examples of good and bad simulation reports.

## 8.19. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 8.19.1. Question 1

What is the primary reason for setting a specific random seed in simulation studies?

- A) To make the simulation run faster
- B) To ensure reproducibility of results
- C) To reduce the number of required replications
- D) To eliminate Monte Carlo error

> **i** Answer
>
> **B.** Setting a seed fixes the pseudorandom stream so the same simulation run produces the same results each time.

### 8.19.2. Question 2

In a Monte Carlo simulation study, what happens to the standard error as the number of replications increases?

- A) It increases proportionally
- B) It remains constant

- C) It decreases at a rate proportional to the square root of the number of replications
- D) It decreases linearly with the number of replications

> **Answer**
>
> **C.** Monte Carlo standard error is $O(1/\sqrt{R})$.

### 8.19.3. Question 3

When comparing statistical methods in a simulation study, which technique helps reduce extraneous variation and provides a more fair comparison?

- A) Using different random seeds for each method
- B) Using common random numbers across methods
- C) Always using the default random number generator
- D) Increasing the sample size until all methods converge

> **Answer**
>
> **B.** Common random numbers couple the realisations across methods, cancelling sampling variability that is not due to the methods themselves.

### 8.19.4. Question 4

A simulation reports a power of 0.82 for method A and 0.80 for method B, with $R = 1000$ replicates. Which of the following is the most defensible conclusion?

- A) Method A has higher power than method B; the difference is real.
- B) The difference of 0.02 is smaller than roughly 2 Monte Carlo SEs at this $R$, so the simulation cannot distinguish the two.
- C) Method A is worse than method B because $0.82 < 0.80$ on a reverse scale.
- D) The simulation must be rerun with the same seed.

Answer

**B.** At $R = 1000$, the Monte Carlo SE for a proportion near 0.8 is roughly 0.013, and the SE for the *difference* is larger still. The nominal 0.02 gap is within sampling noise. Either increase $R$ or report the result as inconclusive.

### 8.19.5. Question 5

In a parallel simulation using `parallel::clusterSetRNGStream()`, the purpose of the L'Ecuyer-CMRG generator is to:

- A) Speed up random number generation by 10×.
- B) Produce independent, non-overlapping streams of random numbers across workers, preventing silent correlation between workers.
- C) Provide cryptographically secure random numbers.
- D) Replace the need for `set.seed()`.

Answer

**B.** Without it, workers can draw from correlated (or identical) sequences, silently invalidating the simulation.

## 8.20. Prerequisites answers

1. To ensure reproducibility: setting a seed before the simulation makes the same sequence of pseudorandom draws appear on each run, so results are exactly reproducible. This is essential for papers, reviewable work, and debugging.
2. The Monte Carlo standard error is proportional to $1/\sqrt{R}$. Multiplying $R$ by 4 halves the standard error. This inverse-square-root law means that increasing precision is expensive: quadrupling $R$ to halve the SE, a 100-fold increase in $R$ to reduce SE by 10.

3. Common random numbers (CRN) force both methods to be evaluated on the same realised random draws, so any difference in their outputs reflects the methods themselves rather than sampling noise in the inputs. This reduces variance in the comparison and allows smaller true differences to be detected. It is the simulation analogue of a paired design in experimental statistics.

# 9. The Bootstrap

## 9.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 9.19.

1. What is the primary problem the bootstrap is designed to solve?
2. Given $B$ bootstrap replications $\widehat{\theta}_1^*, \ldots, \widehat{\theta}_B^*$ of a statistic, how do you compute the bootstrap standard error?
3. Why is Monte Carlo approximation used in practice rather than computing the exact bootstrap distribution over all $n^n$ possible resamples?

## 9.2. Learning objectives

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

- Explain the plug-in principle that underlies the bootstrap.
- Implement a nonparametric bootstrap in base R.
- Compute percentile, basic, BCa, and studentised bootstrap confidence intervals.
- Construct bootstrap p-values via null-enforcing resampling.
- Apply case and residual bootstrap to regression coefficients.
- Recognise when the bootstrap fails (extreme quantiles, time series, dependent data).
- Use the `boot` package for standard cases.

## 9.3. Orientation

The bootstrap (Efron, 1979; Efron & Tibshirani, 1993) is the most useful general-purpose inference tool of the last 50 years. It works when analytic standard errors are impossible or wrong, and it fails in interesting, learnable ways when it does fail. Every biostatistician should be comfortable rolling their own bootstrap.

## 9.4. The statistician's contribution

Before handing a dataset to a large language model with the instruction to 'bootstrap the standard error', four decisions require the statistician's judgment. An LLM will execute a nonparametric iid bootstrap by default. Whether that is the *appropriate* bootstrap for the problem at hand is a determination the model cannot make without statistician input.

**1. Is the bootstrap appropriate for this statistic at all?** The bootstrap fails for extrema (sample max, min, modes), for non-smooth statistics (change-points, mode estimates), for heavy-tailed populations with infinite variance, and for samples below about $n = 20$. An LLM will produce a numerical answer in every case; the number will be misleading in these cases. Recognising which case you are in is the statistician's job.

**2. What is the dependence structure of the data?** An iid bootstrap on time series, clustered data, or longitudinal repeated measures systematically *underestimates* variance. The confidence interval will be too narrow, sometimes dramatically so. Only you can detect the dependence in the data and specify a block bootstrap, a cluster bootstrap, or a parametric alternative that respects the structure.

**3. Which confidence-interval method matches the statistic's behaviour?** The percentile method is transformation-respecting and easy to explain, but undercoverage is common in small samples. BCa is second-order accurate and preferred for publication. The studentised method has the best coverage but requires a bootstrap-within-bootstrap. An LLM typically defaults to the percentile method regardless of whether the bootstrap distribution is skewed; selecting a more appropriate method is the statistician's responsibility.

**4. Is the bootstrap distribution itself plausible?** Always plot the histogram of the $B$ replicates. If it is wildly skewed, truncated sharply at a boundary, or multimodal, either the bootstrap is failing for one of the reasons above or the sampling distribution is genuinely unusual. Both demand investigation; neither should be silently accepted as an answer.

In summary: the LLM provides an efficient implementation of the bootstrap. The statistician determines *what gets implemented*, *on what data*, and *how to interpret the output*. The remainder of this chapter provides the technical vocabulary needed to make those determinations.

## 9.5. The plug-in principle

The bootstrap rests on a simple and consequential idea. We do not know the true population distribution $F$, but we do know the **empirical distribution** $\hat{F}_n$, the distribution that places probability $1/n$ on each observed data point. The *plug-in principle* says: to estimate any functional $\theta = t(F)$, substitute $\hat{F}_n$ for $F$ and compute $\hat{\theta} = t(\hat{F}_n)$. The sample mean, median, quantiles, variance, and correlation are all plug-in estimates.

For *sampling variability*, the plug-in goes one step further. The sampling distribution of $\hat{\theta} - \theta$ under repeated draws from $F$ is approximated by the distribution of $\hat{\theta}^* - \hat{\theta}$ under repeated draws from $\hat{F}_n$. The right-hand side is something we can compute by resampling. The left-hand side is what we actually want to know. The bootstrap is the computational substitution of one for the other.

The substitution is justified by the Glivenko-Cantelli theorem: $\hat{F}_n$ converges uniformly to $F$ as $n$ grows. For smooth statistics, the higher-order accurate bootstrap intervals (BCa, studentised) achieve a coverage error of $O(1/n)$, faster than the $O(1/\sqrt{n})$ rate of a central-limit-theorem interval. The plain percentile bootstrap is only first-order accurate ($O(1/\sqrt{n})$ coverage error), the same rate as the CLT interval; it is worth using primarily for its non-parametric applicability rather than for higher-order accuracy.

> **Check your understanding**
>
> **Q:** What is the empirical distribution $\hat{F}_n$ in the bootstrap context?
> **A:** The discrete distribution that places probability $1/n$ on each of the $n$ observed data points in the original sample.

## 9.6. A simple nonparametric bootstrap

The nonparametric bootstrap is eight lines of base R:

```r
# Original data
x <- c(5, 10, 15, 20, 25)
B <- 1000
boot_means <- numeric(B)
set.seed(47)
for (b in seq_len(B)) {
  resample       <- sample(x, size = length(x),
                           replace = TRUE)
  boot_means[b]  <- mean(resample)
}
sd(boot_means)   # bootstrap standard error
```

Three features of this loop define the nonparametric bootstrap:

1. **Same size.** Each resample has the same length as the original sample.
2. **With replacement.** Without replacement every resample would be a permutation of the original, with no new variability.
3. **Repeated $B$ times.** $B$ is the number of Monte Carlo draws; 1000 is adequate for standard errors, 2000 or more for confidence intervals, 10,000 or more for extreme quantiles.

The vector `boot_means` is the **bootstrap distribution** of the sample mean. Its sample standard deviation is the bootstrap estimate of the standard error of $\bar{x}$. Its quantiles give a percentile confidence interval.

Check your understanding

**Q:** Why do we sample with replacement in the bootstrap?
**A:** Sampling with replacement creates variability between bootstrap samples. Without replacement, every resample would be a permutation of the original data, with no new information about sampling variability.

## 9.7. Bootstrap and sampling distributions

The bootstrap distribution and the sampling distribution are not the same thing; they differ in three ways.

First, **centering**. The sampling distribution is centered at the true parameter $\theta$. The bootstrap distribution is centered at the observed estimate $\hat{\theta}$. For this reason the bootstrap cannot correct a biased estimator: a biased $\hat{\theta}$ produces a bootstrap distribution biased by the same amount in the same direction.

Second, **support**. The sampling distribution can in principle produce any value the true distribution can. The nonparametric bootstrap distribution is restricted to functions of the observed values. This shows up when estimating the maximum of a bounded distribution, where the bootstrap cannot produce values larger than the observed maximum.

Third, **variability**. The bootstrap distribution has slightly less spread than the sampling distribution, by a factor of roughly $\sqrt{(n-1)/n}$. The bias is negligible for large $n$ but compounds with the other small-sample weaknesses of the percentile method discussed below.

The bootstrap *does* faithfully capture the shape of the sampling distribution, skewness, multimodality, and the boundary effects that arise near constrained parameters (such as a correlation coefficient near $\pm 1$). These shape diagnostics inform the choice of confidence-interval method.

Check your understanding

**Q:** Can the bootstrap be used to improve our point estimate $\hat{\theta}$?
**A:** Generally, no. The bootstrap assesses the *variability* of $\hat{\theta}$; it does not change the point estimate. The mean of the bootstrap distribution is approximately $\hat{\theta}$ itself, and a biased $\hat{\theta}$ produces a bootstrap distribution biased in the same direction.

## 9.8. Estimating standard errors

Let $\hat{\theta}^*_1, \dots, \hat{\theta}^*_B$ denote the bootstrap replications and $\bar{\theta}^* = B^{-1}\sum_b \hat{\theta}^*_b$ their mean. The bootstrap estimate of the standard error of $\hat{\theta}$ is

$$\widehat{\mathrm{SE}}_B(\hat{\theta}) = \sqrt{\frac{1}{B-1}\sum_{b=1}^{B}(\hat{\theta}^*_b - \bar{\theta}^*)^2}.$$

In R, this is `sd(boot_replicates)`. The principal value of the bootstrap SE is in cases where an analytic formula for $\mathrm{SE}(\hat{\theta})$ is unavailable or depends on assumptions the data do not satisfy: medians, trimmed means, correlation coefficients, and any custom statistic you can compute from a sample.

The **Monte Carlo error** of the bootstrap SE is approximately $\widehat{\mathrm{SE}}_B/\sqrt{2B}$. With $B = 1000$ this is roughly 2.2% of the bootstrap SE itself, which is negligible for most purposes.

Check your understanding

**Q:** What statistical problems might become easier if we could repeatedly sample from the population?
**A:** We could empirically estimate standard errors, create confidence intervals without relying on parametric assumptions, and better understand sampling distributions for complex statistics. The bootstrap approximates this ideal using resampling from the observed data.

## 9.9. Confidence-interval flavours

Several bootstrap confidence-interval recipes exist, each trading simplicity for coverage accuracy. All assume $B$ is large enough to estimate the required quantiles.

**Percentile interval.** The simplest: take the $\alpha/2$ and $1-\alpha/2$ quantiles of the bootstrap distribution.

```
# Percentile CI by hand
quantile(boot_means, c(0.025, 0.975))
```

Transformation-respecting (a percentile interval on $\log\hat{\theta}$ exponentiates to a percentile interval on $\hat{\theta}$), but first-order accurate only. Tends to undercover in small samples.

**Basic interval.** Reflects the bootstrap distribution around $\hat{\theta}$:

$$[2\hat{\theta} - q_{1-\alpha/2},\ 2\hat{\theta} - q_{\alpha/2}],$$

where $q_\alpha$ is the $\alpha$ quantile of the bootstrap distribution. Corrects a subtle bias the percentile method inherits from centering at $\hat{\theta}$, but loses the transformation-respecting property.

**BCa interval.** Bias-corrected and accelerated. Second-order accurate. Adjusts the percentile endpoints using a bias correction (read from the bootstrap distribution's median) and an acceleration constant (estimated by jackknife). Available in `boot::boot.ci(type = 'bca')`. Preferred for publication-grade intervals.

**Studentised interval.** Bootstraps a pivot $(\hat{\theta}^* - \hat{\theta})/\widehat{\mathrm{SE}}^*$, then inverts to an interval on $\theta$. Second-order accurate and often has the best coverage, but requires a bootstrap-within-bootstrap computation to estimate the inner standard error.

In practice: reach for the percentile interval first; switch to BCa if the sample is small or the statistic is skewed; use the studentised interval only when the cost of the double bootstrap is justified.

## 9.10. Hypothesis testing by resampling

A confidence interval and a hypothesis test answer overlapping questions about a parameter, and the bootstrap supplies both. Hypothesis testing requires one extra construction: resampling must occur under conditions in which the null hypothesis is true. This **null enforcement** step is the load-bearing distinction between a bootstrap confidence interval and a bootstrap test.

For a two-sample mean comparison, $H_0 : \mu_A = \mu_B$, the standard construction is **pool and center**. Concatenate the two samples after centering each at its own sample mean; the pooled vector has mean zero by construction, and resampling from it yields two new groups whose populations have identical means. Compute the observed test statistic on the original data, repeat the calculation on each bootstrap replicate drawn from the centered pool, and report the proportion of replicates whose absolute statistic meets or exceeds the observed.

```
# Pool-and-center bootstrap test of H0: mu_A = mu_B
a <- sleep$extra[sleep$group == 1]
b <- sleep$extra[sleep$group == 2]
obs_diff <- mean(a) - mean(b)

pooled <- c(a - mean(a), b - mean(b))
B <- 10000
set.seed(47)
null_diffs <- replicate(B, {
  a_star <- sample(pooled, length(a), replace = TRUE)
  b_star <- sample(pooled, length(b), replace = TRUE)
  mean(a_star) - mean(b_star)
})
mean(abs(null_diffs) >= abs(obs_diff))
```

The two-sided p-value is `mean(abs(null_diffs) >= abs(obs_diff))`. For a one-sided alternative, drop the absolute values and adjust the comparison direction. Monte Carlo error on the p-value is approximately $\sqrt{p(1-p)/B}$, around 0.002 near $p = 0.05$ when $B = 10{,}000$.

Two comparisons clarify what the bootstrap test is doing. **Permutation tests** also generate a null distribution by shuffling group labels, but without replacement. Permutation tests are exact under exchangeability and are preferable when the assumption is defensible. The bootstrap test is approximate but extends naturally to settings where exchangeability fails: paired data, regression coefficients, or any statistic with a bootstrap implementation. **Classical t-tests** make stronger distributional assumptions: a pool-and-center bootstrap parallels the equal-variance two-sample t-test, while `t.test()` defaults to Welch's unequal-variance test. The methods produce different p-values not because of Monte Carlo error but because they implement different null hypotheses.

The pool-and-center construction generalises. Testing a correlation against zero is achieved by resampling pairs $(x_i, y_i^*)$ where the $y$ values are reshuffled to break the dependence. Testing a single regression coefficient against zero is achieved by bootstrapping the residuals of the reduced model and rebuilding the response under the null. The principle is constant: design a resampling scheme under which the null is true, then read off the tail probability of the observed statistic.

Check your understanding

**Q:** Why must the bootstrap resampling step enforce the null hypothesis when conducting a test, while no such adjustment is needed for a confidence interval?
**A:** A confidence interval describes the variability of $\hat{\theta}$ around the unknown population parameter; the bootstrap distribution centered at $\hat{\theta}$ is exactly the reference distribution wanted. A hypothesis test asks whether the data are compatible with a specified null value $\theta_0$, which requires the distribution of $\hat{\theta}$ under $\theta = \theta_0$. That in turn requires resampling from a population modified to satisfy the null.

## 9.11. Monte Carlo implementation

The theoretical bootstrap enumerates all $n^n$ possible resamples with replacement, and for even modest $n$ this is astronomical: $n = 20$ produces more resamples than there are atoms in the observable universe. In practice

we sample a manageable number $B$ of resamples uniformly at random. This introduces a Monte Carlo error distinct from the statistical error of using a finite original sample.

The Monte Carlo error of a percentile confidence-interval endpoint is approximately $\sqrt{\alpha(1-\alpha)/B} \cdot \widehat{\mathrm{SE}}_B$. For a 95% interval with $B = 1000$, this is about $0.005 \cdot \widehat{\mathrm{SE}}_B$, an order of magnitude smaller than the statistical error for typical $n$.

Rules of thumb: $B = 200$ suffices for standard errors; $B = 1000$ to 2000 for percentile or BCa confidence intervals; $B = 10{,}000$ or more for extreme quantiles or p-values below 0.01. When in doubt, rerun with a different seed and confirm results are stable.

Check your understanding

**Q:** What can happen if you use too few bootstrap samples (for example, $B = 50$)?
**A:** Substantial Monte Carlo error. Standard-error estimates are unstable across seeds, and percentile confidence-interval endpoints are unreliable because a handful of bootstrap replicates anchor each tail quantile.

## 9.12. When the bootstrap fails

The bootstrap works well when the statistic is a smooth function of the data and the sample size is moderate to large. It breaks down in several recognisable patterns.

**Extrema.** The sample maximum of a bounded distribution cannot exceed the observed maximum in any bootstrap resample, so the bootstrap distribution has no mass above the observed max. Bootstrap CIs for extrema, modes, and other boundary statistics are unreliable.

**Non-smooth statistics.** Sample quantiles at extreme tails, modes, and change-points are not smooth functions of the empirical distribution; a small change in data can produce a large change in the estimate. The

bootstrap distribution of a sample median, for instance, is discrete even for continuous data.

**Small samples.** For $n < 20$ the empirical distribution is a poor estimate of the true distribution, especially in the tails. Parametric bootstrap (resampling from a fitted parametric model) or a normal approximation with bias correction may perform better.

**Dependent data.** An iid bootstrap on time series or clustered data destroys the dependence structure and underestimates variance. Use a **block bootstrap** for time series (resample contiguous blocks of length $\ell$) or a **cluster bootstrap** for hierarchical data (resample whole clusters, keeping within-cluster structure intact).

**Heavy tails.** When the population has infinite variance, the bootstrap distribution converges slowly or not at all. Sample means and regression coefficients from heavy-tailed data require trimming, robust statistics, or the parametric bootstrap with a heavy-tailed assumption.

Check your understanding

**Q:** Why might bootstrap methods struggle with the sample median when the sample size is small (say, $n = 5$)?
**A:** With small samples the bootstrap distribution of the median is discrete and limited to the observed values. This coarse approximation of the sampling distribution yields inaccurate standard errors and confidence intervals.

## 9.13. The boot package

For production use, the `boot` package (Canty and Ripley, based on Davison and Hinkley's book) handles the resampling, parallelism, and confidence-interval calculations. The standard workflow has three steps.

```
library(boot)

# Step 1: define the statistic as a function of data
#         and indices.
```

```r
median_stat <- function(data, indices) {
  median(data[indices])
}

# Step 2: run the bootstrap.
set.seed(47)
x <- c(10, 14, 18, 23, 27, 32, 38, 42, 52, 68)
b <- boot(data = x, statistic = median_stat, R = 2000)
print(b)

# Step 3: compute confidence intervals.
boot.ci(b, type = c('perc', 'basic', 'bca'))
```

The `statistic` function *must* accept `(data, indices)` and return the statistic computed on `data[indices]` (or `data[indices, ]` for a data frame). The `boot()` function handles the resampling via the indices, which is more memory-efficient than constructing each resample as a copy of the data. For stratified designs pass `strata = group`; for parallel execution set `parallel = 'multicore'` (Linux/macOS) or `'snow'` (portable) with `ncpus = parallel::detectCores() - 1`.

For specialised bootstraps:

- `car::Boot()` wraps `boot()` for regression models, handling case resampling, residual resampling, and wild bootstrap for heteroscedastic errors.
- `boot::tsboot()` implements block bootstrap for time series.
- `rsample::bootstraps()` integrates with the `tidymodels` ecosystem.

Check your understanding

**Q:** What is the role of the `indices` parameter in the `boot` statistic function?
**A:** `indices` carries the positions of observations selected during each resample. Rather than constructing an actual bootstrap sample and passing it to the function, `boot()` passes only the indices; the function subsets the original data using `data[indices]`. This is more memory-

efficient than copying the data for every replicate.

Check your understanding

**Q:** In what scenarios would you prefer traditional methods over the bootstrap?
**A:** When sample sizes are very small, when the statistic has a known sampling distribution under assumptions that the data plausibly satisfy, or when computational resources are limited. Traditional methods also provide analytic insights (closed-form expressions, exact distributions) that numerical methods can obscure.

## 9.14. Bootstrapping regression coefficients

Linear regression provides a productive testbed for the bootstrap machinery. Two resampling strategies are standard, and a third is worth knowing for diagnostic-failure cases.

**Case (paired) bootstrap.** Resample whole rows of the data matrix with replacement, refit the model, and record the coefficient. This makes no assumption about the conditional distribution of $y$ given $x$ and remains valid under heteroscedasticity, because joint resampling of $(x, y)$ preserves whatever variance structure the data contain.

**Residual bootstrap.** Fit the model once, extract residuals $e_i = y_i - \hat{y}_i$, resample residuals with replacement to obtain $e_i^*$, and construct synthetic responses $y_i^* = \hat{y}_i + e_i^*$. Refit the model on $(x_i, y_i^*)$. This strategy assumes residuals are exchangeable: their distribution does not depend on the predictors. When that assumption holds, residual bootstrap is more efficient than case bootstrap; when it fails, residual-bootstrap intervals are too narrow because they impose a constant-variance error structure the data lack.

**Wild bootstrap.** When residuals are clearly heteroscedastic but the conditional-mean structure is correct, set $e_i^* = e_i v_i$ with $v_i$ an independent random sign or a Mammen two-point variable. This preserves the conditional variance pattern while still generating a valid resampling distribution.

A worked case bootstrap for the slope of `wt` predicting `mpg` in `mtcars`:

```
library(boot)
set.seed(47)

slope_stat <- function(data, indices) {
  fit <- lm(mpg ~ wt, data = data[indices, ])
  coef(fit)[['wt']]
}

b_lm <- boot(mtcars, slope_stat, R = 2000)
boot.ci(b_lm, type = c('perc', 'bca'))

# Compare with the classical normal-theory interval
confint(lm(mpg ~ wt, data = mtcars))['wt', ]
```

The percentile and BCa intervals from the case bootstrap are slightly wider than the classical interval from `confint()`. The discrepancy is informative: a residuals-versus-fitted plot for this model shows variance increasing with fitted value, the mild heteroscedasticity that the classical interval ignores by assuming constant error variance. The bootstrap absorbs this through joint resampling and produces a more defensible interval.

The choice among strategies follows from the assumptions:

- Use the case bootstrap when residual diagnostics suggest heteroscedasticity, when the model may be misspecified, or as a robust default in the absence of contrary information.
- Use the residual bootstrap when residuals are clean and the conditional mean is well captured. The narrower intervals it produces are warranted only when the assumption is defensible.
- Use the wild bootstrap when heteroscedasticity is severe but the mean structure is sound, particularly in small samples where case bootstrap may resample the same influential observation multiple times.

The same machinery extends to generalised linear models (case bootstrap on `glm` objects), random-effect models (cluster bootstrap that respects the grouping structure), and robust regression. The `car::Boot()` function introduced in the previous section handles several of these cases automatically.

Check your understanding

**Q:** Suppose a residuals-versus-fitted plot for a regression model shows residual spread increasing with the fitted value. Which regression bootstrap would you choose, and why?
**A:** The case (paired) bootstrap, because it resamples whole $(x, y)$ rows and therefore preserves any heteroscedastic variance structure in the original data. The residual bootstrap imposes a single residual distribution that does not depend on $x$ and would systematically underestimate variability in this setting.

## 9.15. Collaborating with an LLM on the bootstrap

Section Section 9.4 identified the four decisions that require the statistician's judgment. This section provides practice exercises. Each prompt below is adversarial: it is designed to expose a specific LLM failure mode. The model's response should be treated as a hypothesis requiring verification, not as a result to be accepted at face value.

### 9.15.1. Correlation and the Fisher-z transformation

**Prompt.** 'Bootstrap a 95% confidence interval for the correlation between `waiting` and `eruptions` in the `faithful` dataset.'

**What to watch for.** A correlation is bounded in $[-1, 1]$, so its sampling distribution is asymmetric near the boundaries. A default nonparametric percentile bootstrap on the raw correlation is inefficient and can produce endpoints that fall outside $[-1, 1]$. The textbook remedy is to bootstrap on the Fisher-z transformation $\frac{1}{2}\log\frac{1+r}{1-r}$ and then back-transform.

**Verification.** Inspect the code. Does it wrap the correlation in `atanh()` before computing quantiles and `tanh()` after? If not, compare interval widths against `cor.test()`. A noticeable discrepancy indicates the LLM skipped the transformation; ask it a follow-up on whether the percentile endpoints are guaranteed to stay in the valid range.

### 9.15.2. The bootstrap maximum

**Prompt.** 'Write R code to compute a 95% percentile confidence interval for the maximum of a sample drawn from `runif(50, 0, 1)`.'

**What to watch for.** The bootstrap fails for extrema. The bootstrap maximum cannot exceed the observed sample maximum, so the upper CI endpoint is always the observed max itself, a hard ceiling that is *not* a property of the true sampling distribution.

**Verification.** Run the code. Check whether the upper endpoint equals the sample maximum. Then ask the LLM: 'Is this interval trustworthy? What does standard bootstrap theory say about statistics defined as extrema of the sample?' See whether it recovers the failure mode on its own or insists the interval is valid.

### 9.15.3. Time-series mean

**Prompt.** Supply a `ts` object of monthly clinical biomarker observations over three years and ask: 'Bootstrap the mean and a 95% confidence interval.'

**What to watch for.** Autocorrelated data violate the iid assumption. An iid bootstrap destroys the dependence structure and *underestimates* variance, typically producing an interval much narrower than the correct one.

**Verification.** Does the code call `boot::tsboot()` or a block-bootstrap function? Or does it call plain `boot::boot()`? If the latter, the answer is wrong. Compute `acf(x)` and confirm non-trivial autocorrelation, then redo the bootstrap with `tsboot(x, ..., l = round(length(x)^(1/3)), sim = 'fixed')`. Compare interval widths.

### 9.15.4. Principle in use

These three prompts exercise the four statistician's contributions from Section 9.4. The bootstrap maximum and the time-series mean both test decision 1 (is the bootstrap appropriate for this statistic or this data?). The time-series mean additionally tests decision 2 (dependence structure). The

correlation prompt tests decisions 3 and 4 (confidence-interval method and bootstrap-distribution shape). None of these verifications requires advanced skill; all require that you ask the question the LLM will not ask for you.

## 9.16. Exercises

1. Using only base R, write `bootstrap_ci(x, stat, R = 1000, conf = 0.95)` that returns a percentile confidence interval for an arbitrary statistic of a numeric vector `x`.
2. Apply your function to compute a 95% CI for the median of a right-skewed sample (e.g., `rexp(50, 1)`). Compare to the interval from `boot::boot.ci(type = 'perc')`.
3. Compute coverage of the percentile CI in a simulation with 2000 replicates. Does it cover at the nominal 95%? Explain any under-coverage.

## 9.17. Further reading

- (Efron & Tibshirani, 1986), the clearest short introduction to the percentile bootstrap.
- (Efron & Tibshirani, 1993) Chapters 13-15, BCa and studentised intervals.
- (Hesterberg, 2015), pedagogical pitfalls that appear in student work.

## 9.18. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 9.18.1. Question 1

What is the primary purpose of the bootstrap method?

- A) To replace traditional statistical analysis completely

- B) To assess the accuracy of parameter estimates when closed-form standard errors are difficult to derive
- C) To create larger datasets from small samples
- D) To identify outliers in statistical datasets

> **i** Answer
>
> **B.** The bootstrap provides approximate standard errors, bias, and confidence intervals by resampling.

### 9.18.2. Question 2

How does the bootstrap estimate the standard error of a statistic?

- A) By calculating the exact formula for every possible case
- B) By approximating the sampling distribution through repeatedly resampling with replacement from the observed data
- C) By assuming the data follows a normal distribution
- D) By comparing the statistic to previously established benchmarks

> **i** Answer
>
> **B.** The bootstrap treats the empirical distribution of the data as a proxy for the true population distribution and resamples from it with replacement.

### 9.18.3. Question 3

Given $B$ bootstrap replications of a statistic, how is the bootstrap standard error calculated?

- A) By taking the square root of the sample variance divided by $n$
- B) By computing the standard deviation of the $B$ bootstrap replications
- C) By using a predefined formula based on the Central Limit Theorem
- D) By applying maximum likelihood estimation to the data

Answer

**B.** The bootstrap SE is the sample SD of the $B$ bootstrap replicates.

### 9.18.4. Question 4

Why is Monte Carlo approximation typically used in bootstrap estimation rather than computing the exact bootstrap distribution?

- A) Because the exact bootstrap distribution requires evaluating all $n^n$ possible resamples, which is computationally infeasible
- B) Because the Central Limit Theorem does not apply to bootstrap estimates
- C) Because the original dataset is always too small to be reliable
- D) Because exact enumeration produces biased results

Answer

**A.** Even for small $n$, $n^n$ explodes quickly. Monte Carlo samples a practical number $B$ of resamples, controlling accuracy by choice of $B$.

### 9.18.5. Question 5

How is the bootstrap estimate of bias for a statistic $\hat{\theta}$ defined?

- A) The difference between the mean of the bootstrap replications and the original estimate $\hat{\theta}$
- B) The standard deviation of the bootstrap replications
- C) The skewness of the bootstrap distribution
- D) The difference between the median and mean of the bootstrap replications

Answer

**A.** Bootstrap bias is $\bar{\theta}^* - \hat{\theta}$, where $\bar{\theta}^*$ is the mean of bootstrap replicates.

## 9.19. Prerequisites answers

1. The bootstrap assesses the sampling distribution of a statistic (its standard error, bias, and confidence interval) when a closed-form expression is unavailable, complicated, or depends on distributional assumptions the data do not satisfy.
2. The bootstrap standard error is the sample standard deviation of the $B$ bootstrap replications: $\widehat{\mathrm{SE}}_B(\hat{\theta}) = \sqrt{\frac{1}{B-1}\sum_{b=1}^{B}(\hat{\theta}_b^* - \bar{\theta}^*)^2}$.
3. The exact bootstrap distribution requires evaluating the statistic on all $n^n$ possible resamples (with replacement from $n$ observations), which is computationally infeasible for any meaningful $n$. Monte Carlo approximation draws a manageable number $B$ of resamples at random.

# Part IV.

# Statistical Models

# 10. Linear Models in Practice

## 10.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 10.19.

1. Name three assumptions of the classical linear regression model.
2. What does $R^2$ represent, and what does an $R^2$ of 0.7 tell you about a fitted model?
3. What is the primary purpose of residual diagnostic plots in linear regression?

## 10.2. Learning objectives

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

- Fit a linear model in R with `lm()` and interpret every column of `summary()`.
- Diagnose a fitted model using the four default plots: residuals, Q-Q, scale-location, Cook's distance.
- Build a model matrix `X` for a mix of continuous and categorical predictors, including interactions and polynomial terms.
- Compute the coefficient vector by hand using the QR decomposition and reproduce the output of `lm()`, point estimates, standard errors, $R^2$, residual degrees of freedom, from scratch.
- Recognise and report violations of model assumptions: heteroscedasticity, non-linearity, non-normality, influential observations, multicollinearity.
- Compute robust (sandwich) standard errors with `sandwich::vcovHC()` when homoscedasticity fails.

- Work with `broom::tidy()`, `glance()`, and `augment()` for reproducible reporting of linear models.

## 10.3. Orientation

`lm()` is the workhorse of applied statistics and the first real test of everything we have built so far: matrix algebra, QR decomposition, vectorised R, and the grammar of data frames. This chapter treats `lm()` as a white box. We will not invoke it until we have reimplemented most of what it does.

The linear model assumes

$$\mathbf{y} = \mathbf{X}\beta + \varepsilon, \quad \varepsilon \sim N(0, \sigma^2 I).$$

From chapter 5 we know the least-squares estimator is $\hat{\beta} = (X^T X)^{-1} X^T y$. From chapter 6 we know why `lm()` uses QR rather than the normal equations. This chapter adds the inferential machinery (standard errors, t-statistics, p-values, $R^2$), the diagnostics, and the working craft of actually fitting and interpreting linear models on real data.

## 10.4. The statistician's contribution

`lm()` fits a model. The interesting questions are which model, on which variables, on which transformations, with which subset, and how to interpret the result. These are the statistician's contribution; they are not automatable.

**Which variables to include.** The naive answer ('whichever predicts') is wrong for inference. In a randomised trial, the Table-1 covariates go in not because they improve prediction but because they were balanced at randomisation; adjusting for them gives the pre-registered estimand. In an observational study, the covariate set is the set that blocks non-causal paths between exposure and outcome, determined by a causal diagram drawn before the model is fit. Stepwise selection and similar algorithmic procedures produce standard errors that understate uncertainty and coefficients that do not mean what they appear to mean.

**Which scale.** A coefficient on age is interpretable if age is in years; less so if centred at 50 but unscaled; less still if transformed to a z-score. The choice depends on the audience. For a clinical paper, a one-year increment is natural; for a methods paper, a z-score may read more cleanly. The transformation on the *outcome*, log, square root, Box-Cox, is a more consequential decision: it changes the estimand from 'additive effect on $y$' to 'multiplicative effect on $e^y$', and these are not interchangeable.

**Whether the model's claims are defensible.** A linear model's p-values and confidence intervals are valid under its assumptions. When the assumptions fail, the numbers are still computed and printed; they are just no longer meaningful. Detecting failure is the purpose of diagnostics. Choosing a remedy, a transformation, robust standard errors, a different model entirely, reporting a caveat, is the statistician's judgement.

**Statistical vs. clinical significance.** A $p < 0.05$ treatment effect of 0.2 mmHg on blood pressure is not clinically meaningful even if it is statistically certain. A $p = 0.07$ effect of 8 mmHg may be clinically very meaningful but imprecisely estimated. The statistician reports both; claiming importance because of $p$ alone, or dismissing importance for the same reason, is not defensible.

These decisions shape whether a linear-model analysis is honest or misleading. An LLM will happily fit any model you describe; it will not tell you that the model is not the right one for your question.

## 10.5. Anatomy of `lm()`

R's formula interface abstracts away the model matrix. The expression `lm(mpg ~ wt + hp, data = mtcars)` specifies that the outcome is `mpg`, the predictors are `wt` and `hp`, an intercept is included by default, and the data are in `mtcars`. Internally, R:

1. Parses the formula into a list of terms.
2. Builds the model matrix $X$ via `model.matrix()`, converting factors to indicator columns and applying any transformations mentioned in the formula.
3. Computes the QR decomposition of $X$.

4. Solves for $\hat{\beta}$ via `qr.coef(qrX, y)`.
5. Computes residuals, fitted values, residual variance $\hat{\sigma}^2 = \|y - X\hat{\beta}\|^2/(n-p)$, and the covariance matrix $\widehat{\mathrm{Var}}(\hat{\beta}) = \hat{\sigma}^2(X^TX)^{-1}$ , the last via back-substitution on $R$, not by forming $(X^TX)^{-1}$ explicitly.
6. Returns all of this as an `lm` object.

```
fit <- lm(mpg ~ wt + hp, data = mtcars)
class(fit)                  # "lm"
names(fit)                  # what's in the object
#>  [1] "coefficients"  "residuals"     "effects"       "rank"
#>  [5] "fitted.values" "assign"        "qr"            "df.residual"
#>  [9] "xlevels"       "call"          "terms"         "model"

summary(fit)
```

The `summary()` output has four parts:

1. **Residuals.** Min, 1Q, median, 3Q, max. Used for a quick sanity check: median near zero, approximately symmetric quantiles.
2. **Coefficients.** Estimate, standard error, $t$-value, and $\Pr(>|t|)$, for each coefficient.
3. **Residual standard error.** $\hat{\sigma}$ and residual degrees of freedom $n-p$.
4. **F-statistic.** Tests whether the model as a whole explains variation beyond the intercept.

The coefficients table is the most important part.

- **Estimate.** $\hat{\beta}_j$, the fitted coefficient.
- **Std. Error.** $\widehat{\mathrm{SE}}(\hat{\beta}_j) = \sqrt{\hat{\sigma}^2[(X^TX)^{-1}]_{jj}}$.
- **t value.** Estimate divided by Std. Error.
- **Pr(>|t|).** Two-sided p-value from a $t$-distribution with $n-p$ degrees of freedom.

Extract specific components via accessor functions:

```
coef(fit)              # coefficient vector
confint(fit)           # 95% CIs for coefficients
fitted(fit)            # fitted values
resid(fit)             # residuals
vcov(fit)              # coefficient covariance matrix
sigma(fit)             # residual standard error
AIC(fit); BIC(fit)     # information criteria
```

Prefer these over `fit$coefficients`; accessor functions are stable across `lm`, `glm`, and many other model classes.

## 10.6. Building the model matrix

`model.matrix()` converts a formula into the design matrix $X$ that `lm()` actually works with. Understanding this step makes interactions, contrasts, and coefficient interpretation far clearer.

```
X <- model.matrix(mpg ~ wt + hp, data = mtcars)
head(X)
#>                   (Intercept)    wt  hp
#> Mazda RX4                   1 2.620 110
#> Mazda RX4 Wag               1 2.875 110
#> Datsun 710                  1 2.320  93
```

Notice the intercept column of 1s, included by default. To suppress it: `lm(mpg ~ 0 + wt + hp)` or `lm(mpg ~ wt + hp - 1)`.

### 10.6.1. Categorical predictors and contrasts

When a factor appears in a formula, R creates $k - 1$ indicator columns for a factor with $k$ levels:

```
fit <- lm(mpg ~ wt + factor(cyl), data = mtcars)
head(model.matrix(fit))
#>                   (Intercept)    wt factor(cyl)6 factor(cyl)8
#> Mazda RX4                   1 2.620            1            0
#> Datsun 710                  1 2.320            0            0
```

The factor's first level ('4' here) is the reference; the other two levels have their own indicator columns. The intercept represents the mean of the reference level (at $wt = 0$); the factor coefficients represent differences from the reference.

To change the reference level:

```
mtcars$cyl_f <- relevel(factor(mtcars$cyl), ref = "8")
```

Or via factor levels directly: `factor(mtcars$cyl, levels = c("8", "4", "6"))`.

For a non-default contrast coding (e.g., sum-to-zero, which gives coefficients interpretable as deviations from the grand mean), set `contrasts()` on the factor:

```
contrasts(mtcars$cyl_f) <- contr.sum
```

Default treatment contrasts are clinically interpretable and almost always the right choice. Sum contrasts are useful in some ANOVA presentations but confuse readers who expect reference-level interpretation.

### 10.6.2. Interactions

Specified with `:` (interaction only) or `*` (both main effects and interaction):

```
lm(mpg ~ wt * cyl, data = mtcars)    # wt, cyl, wt:cyl
lm(mpg ~ wt + cyl + wt:cyl, data = mtcars)   # same thing
```

For an interaction between continuous `wt` and factor `cyl`, the model matrix gets an extra column for every non- reference level of `cyl`: `wt:cyl6`, `wt:cyl8`. Interpretation: the slope on `wt` for the reference level is the main-effect coefficient; the slope for non-reference levels is main-effect plus the interaction coefficient.

### 10.6.3. Polynomial and transformed terms

`I()` insulates an expression from formula parsing:

```
lm(mpg ~ wt + I(wt^2), data = mtcars)
lm(mpg ~ log(hp) + wt, data = mtcars)
```

`poly()` creates orthogonal polynomial terms, which are easier to interpret and better-conditioned than raw powers:

```
lm(mpg ~ poly(wt, 2), data = mtcars)
```

Orthogonal polynomials have the property that the coefficient on the linear term is not affected by whether you include the quadratic term.

Check your understanding: reference level choice

**Question.** You fit `lm(outcome ~ treatment, data = trial)` where `treatment` is a factor with levels 'placebo', 'low-dose', and 'high-dose'. The `Intercept` coefficient in the output estimates 18.2, and the `treatmentlow-dose` coefficient is -4.1. What are the mean outcomes in the placebo and low-dose groups?

**Answer.**

The default reference level is alphabetical: 'high-dose'. So the intercept estimates the mean outcome in the high- dose group (18.2), and the low-dose coefficient gives the difference between low-dose and high-dose (-4.1), meaning low-dose mean is 14.1. To get placebo as the reference — usually what you want in a trial, use `factor(treatment, levels = c("placebo", "low-dose", "high-dose"))` or `relevel(treatment, ref = "placebo")`. This is a classic source of misreporting: the intercept does not mean 'base-

> line' in any natural sense; it means 'the reference level that R picked'. Always set the reference level deliberately.

## 10.7. Reimplementing `lm()` in 10 lines

```
set.seed(1)
n <- 100; p <- 3
X <- cbind(1, matrix(rnorm(n * p), n, p))
y <- X %*% c(2, 1, -0.5, 0.3) + rnorm(n)

qrX <- qr(X)
beta <- qr.coef(qrX, y)
resids <- qr.resid(qrX, y)
rss <- sum(resids^2)
df_resid <- n - ncol(X)
sigma2 <- rss / df_resid

# coefficient covariance: sigma^2 * (X'X)^-1 = sigma^2 * solve(R) %*% t
R <- qr.R(qrX)
Rinv <- backsolve(R, diag(ncol(R)))
vcov_beta <- sigma2 * tcrossprod(Rinv)
se_beta <- sqrt(diag(vcov_beta))

# R-squared
tss <- sum((y - mean(y))^2)
r2 <- 1 - rss / tss

# compare to lm()
lm_fit <- lm(y ~ X - 1)
all.equal(as.vector(beta), unname(coef(lm_fit)))
#> [1] TRUE
all.equal(as.vector(se_beta),
          unname(summary(lm_fit)$coefficients[, "Std. Error"]))
#> [1] TRUE
```

Everything `lm()` reports, coefficients, SEs, $R^2$, residual SD, df, comes from the QR decomposition, the residuals, and $(X^T X)^{-1}$. No magic, no Bayesian prior, no hidden regularisation.

## 10.8. Diagnostics

`plot(fit)` produces four diagnostic plots that together cover the main assumption checks.

**Plot 1: Residuals vs. Fitted.** Checks linearity and homoscedasticity. Looks for: no systematic trend (linear relationship holds), consistent spread (variance constant). A curve → non-linearity. A funnel → heteroscedasticity.

**Plot 2: Q-Q plot of standardised residuals.** Checks approximate normality. Points should lie near the 45° line. Heavy tails show up as an 'S' shape; skewness shows up as a curved deviation from the line.

**Plot 3: Scale-Location.** The square root of the absolute standardised residual against fitted values. Another view of homoscedasticity, more sensitive than plot 1 to trends in spread. A monotonic increasing trend → variance increases with the mean, a classical pattern for count or proportion data.

**Plot 4: Residuals vs. Leverage.** Identifies influential observations. Cook's distance contours (dashed lines) mark the regions of concern. Points far to the right (high-leverage: unusual covariate values) with large residuals deserve scrutiny.

```
par(mfrow = c(2, 2))
plot(fit)
par(mfrow = c(1, 1))
```

What none of these plots can check is **independence**. For time-series or clustered data, plot the residuals against time or against the cluster ID and look for structure. Independence violations usually require a different model (mixed-effects, GEE, or an explicit correlation structure).

### 10.8.1. Leverage, influence, Cook's distance

Not every observation matters equally. **Leverage** ($h_i$, the $i$th diagonal of the hat matrix $H = X(X^TX)^{-1}X^T$) measures how unusual observation $i$'s predictor values are; high-leverage observations pull the fit toward themselves.

```
h <- hatvalues(fit)
```

**Cook's distance** combines leverage and residual size into a single measure of how much the regression would change if observation $i$ were removed:

$$D_i = \frac{e_i^2}{p\hat{\sigma}^2} \cdot \frac{h_i}{(1-h_i)^2}.$$

```
cd <- cooks.distance(fit)
```

A rule of thumb: $D_i > 4/n$ warrants investigation; $D_i > 1$ is a red flag. Do not blindly delete influential observations. Instead: check whether the observation is correctly coded, whether it is part of a class of points the model is not designed for, or whether re-fitting without it changes substantive conclusions.

**DFBETA** (or DFBETAS, its studentised version) measures the change in each coefficient when each observation is removed:

```
db <- dfbetas(fit)
```

Useful when you want to know *which* coefficients an influential point is moving.

> Check your understanding: heteroscedasticity
>
> **Question.** The residuals-vs-fitted plot shows a clear fan shape: residuals small for small fitted values and large for large fitted values. What does this violate, and what are your remedies?
> **Answer.**
> The assumption violated is constant error variance (homoscedasticity).

Consequences: point estimates are still unbiased, but standard errors and their derived quantities (CIs, p-values) are wrong. Remedies: (a) variance-stabilising transformation of the outcome (log for variance proportional to mean^2, square-root for variance proportional to mean), (b) weighted least squares with weights inversely proportional to estimated variance, or (c) robust (sandwich / HC) standard errors that stay consistent under heteroscedasticity without re-fitting the model, `sandwich::vcovHC(fit, type = "HC3")` is the default recommendation. The choice depends on whether you also care about efficient estimation (use WLS or transform) or whether you just want honest SEs with the existing fit (use sandwich).

## 10.9. Robust (sandwich) standard errors

When homoscedasticity is implausible but the model is otherwise defensible, the sandwich estimator gives heteroscedasticity-consistent SEs without changing the point estimates:

```
library(sandwich)
library(lmtest)

fit <- lm(mpg ~ wt + hp, data = mtcars)
coeftest(fit, vcov. = vcovHC(fit, type = "HC3"))
```

HC3 is the default recommendation for moderate sample sizes; HC0 through HC2 are alternatives that differ in how they weight by leverage. Do not use robust SEs as a substitute for diagnosing why homoscedasticity fails; they correct SEs but do not address a mis-specified mean function.

## 10.10. Multicollinearity

Highly correlated predictors produce unstable coefficient estimates. The **variance inflation factor** (VIF) quantifies the inflation:

$$\text{VIF}_j = \frac{1}{1 - R_j^2}$$

where $R_j^2$ is from regressing $x_j$ on all other predictors. VIF $> 5$ is a warning; VIF $> 10$ is usually a problem.

```
library(car)
vif(fit)
```

Remedies:

- Drop one of a near-redundant pair of predictors.
- Combine them (mean, principal component).
- Use ridge regression, which introduces a small bias in exchange for large variance reduction.

Multicollinearity does not bias coefficient estimates; it just makes them imprecise. If the goal is prediction, some degree of multicollinearity is tolerable. If the goal is inference on individual coefficients, it is not.

## 10.11. Model comparison

Nested models can be compared with an F-test via `anova()`:

```
fit_small <- lm(mpg ~ wt, data = mtcars)
fit_large <- lm(mpg ~ wt + hp + disp, data = mtcars)
anova(fit_small, fit_large)
```

Non-nested models can be compared by AIC or BIC:

```
AIC(fit_small, fit_large)
BIC(fit_small, fit_large)
```

Lower is better for both. AIC estimates expected prediction error; BIC approximates model posterior probability under a unit-information prior. BIC penalises complexity more heavily than AIC at large $n$, so BIC tends to prefer smaller models.

Neither is a substitute for cross-validation when the goal is genuine prediction. For inference, neither is a substitute for thinking carefully about which predictors belong in the model based on subject-matter knowledge.

## 10.12. Reproducible reporting with broom

`broom` converts model objects to tidy data frames, suitable for further manipulation and rendering in reports:

```
library(broom)

tidy(fit)         # per-coefficient: estimate, SE, t, p
glance(fit)       # per-model: R2, adj R2, sigma, AIC, BIC, df, ...
augment(fit)      # per-observation: fitted, residuals, hat values, c
```

Pair with `gt` or `kableExtra` for publication-quality tables; with `ggplot2` for coefficient forest plots, fitted- value overlays, and diagnostic plots customised beyond `plot(fit)`.

Check your understanding: extracting model output

**Question.** You want to produce a coefficient plot with 95% CIs for a linear model fit to 12 predictors. Which `broom` function gives you the right data shape with minimal further work?
**Answer.**
`broom::tidy(fit, conf.int = TRUE)` returns one row per coefficient with columns for estimate, standard error, statistic, p-value, and (with `conf.int = TRUE`) lower and upper CI bounds. That is exactly the shape `ggplot2::geom_pointrange()` expects. `glance()` gives per-model summaries; `augment()` gives per-observation data. For the coefficient plot, `tidy()` with `conf.int = TRUE` is the one-line answer.

## 10.13. Collaborating with an LLM on linear models

LLMs can produce correct linear-model code almost instantly. They are less reliable when it comes to the judgement that surrounds the code: what to include, how to interpret, when to worry about assumptions.

**Prompt 1: drafting a model specification.** Describe the study and the question, and ask: 'write a `lm()` call that addresses this question, explain why each predictor is included, and flag any assumptions that may be questionable.'

*What to watch for.* LLMs tend to put kitchen-sink models together (lots of predictors) rather than models matched to the question. Check whether the predictors are there for causal blocking, for adjustment, for prediction, or just because they were in the dataset. If the justification is 'it was in the data', push back.

*Verification.* Compare the LLM's specification to what the study's analysis plan (or a similar published paper) would have pre-specified. Discrepancies are prompts for thought, not red flags in themselves.

**Prompt 2: interpreting coefficients.** Paste the `summary(fit)` output and ask: 'interpret each coefficient in the context of the scientific question, being explicit about the reference level for categorical predictors.'

*What to watch for.* LLMs occasionally paraphrase column names as if they were interpretations ('the estimate is 0.42, the standard error is 0.12'). A real interpretation includes the direction of the effect, its magnitude on the outcome scale, and the scientific meaning of a one-unit change in the predictor. If the interpretation does not read like something a clinical collaborator would write, rewrite it.

*Verification.* Double-check any mention of the intercept and any categorical coefficient: reference-level confusion is the single most common LLM error in regression interpretation.

**Prompt 3: diagnosing a failed fit.** Describe symptoms (weird coefficients, huge SEs, non-convergence) and ask: 'what is likely going wrong, and what diagnostics would clarify?'

*What to watch for.* LLMs are reasonable at proposing diagnostics (VIF, residual plots, Cook's distance) but less reliable at prioritising them. For

an analysis deadline, start with the cheapest diagnostics (check for linear combinations among predictors, plot residuals) before the expensive ones (Cook's distance on every observation, bootstrap SEs).

*Verification.* Run the suggested diagnostics and compare results across them. If all point the same way, trust the conclusion. If they disagree, the problem is more subtle than any one diagnostic is detecting.

## 10.14. Worked example: Arthritis trial

```
data("Arthritis", package = "vcd")
Arthritis$Improved_n <- as.numeric(Arthritis$Improved)   # 1/2/3
# technically ordinal ,  see next chapter for a proper ordinal model

fit <- lm(Improved_n ~ Treatment + Age + Sex, data = Arthritis)
summary(fit)
broom::tidy(fit, conf.int = TRUE)

# diagnostics
par(mfrow = c(2, 2))
plot(fit)
par(mfrow = c(1, 1))
```

The Treatment effect (Treated vs. Placebo, adjusting for age and sex) appears in the coefficient table. Interpretation must be cautious: the outcome is ordinal and the 0-1-2 coding imposes equal spacing that may not be clinically valid. The next chapter's proportional-odds model is a better fit; this example is here to demonstrate workflow, not to claim that linear regression on an ordinal outcome is the right model.

## 10.15. Principle in use

Three habits define defensible linear-model practice:

1. **Pre-specify the model.** Predictors, transformations, reference levels chosen before looking at the data (or at least before looking at any outcome-by-predictor relationship). Ex post data-driven model choice destroys the nominal error rates.
2. **Inspect diagnostics before interpreting.** A pathological residual pattern means the p-values on the table are not what they appear to be. Fix or acknowledge the violation before writing the results paragraph.
3. **Interpret on the scale of the outcome and the audience.** A clinical reader wants 'mmHg per 10-year increase in age'; a methods reader may want a standardised coefficient. Write for the reader who will actually use the result.

## 10.16. Exercises

1. Using `palmerpenguins::penguins`, fit `body_mass_g ~ flipper_length_mm + species`. Extract the coefficient vector with `coef()` and reproduce it with `qr.coef(qr(X), y)` where `X` is the model matrix.
2. For the same fit, reproduce the residual standard error, the $R^2$, and the standard errors of the coefficients without calling `summary()`. Verify every number agrees with `summary(fit)`.
3. Find the observation with the largest Cook's distance and re-fit the model without it. Report the change in coefficients and the change in $\hat{\sigma}$.
4. Fit a model with interactions between species and flipper length. Use `emmeans::emmeans()` or a hand- computed contrast to estimate the species-specific slope on flipper length. Compare to a simple pooled slope.
5. Apply `sandwich::vcovHC()` to a linear model where the residuals-vs-fitted plot shows heteroscedasticity. Compare the robust and classical standard errors. How much do the p-values change?

## 10.17. Further reading

- (Legler & Roback, 2019) Chapter 1, OLS assumption review.

- (Gelman et al., 2020), a modern reference for applied regression in R.
- (Harrell, 2015), detailed treatment of model building, interactions, and nonlinearity. The standard reference for practising biostatisticians.
- (UCLA OARC Statistical Methods and Data Analytics, n.d.), a thorough walkthrough of residual diagnostics in R.

## 10.18. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 10.18.1. Question 1

Which of the following is an assumption of linear regression models?

- A) The predictor variables must follow a normal distribution
- B) The response variable must be categorical
- C) The error terms have constant variance (homoscedasticity)
- D) The relationship between predictors must be linear

> **i** Answer
>
> **C.** Homoscedasticity is one of the Gauss-Markov assumptions. The predictors need not be normal, the response need not be categorical, and the assumption is on the mean function, not on inter-predictor linearity.

### 10.18.2. Question 2

In a linear regression model, what does $R^2$ represent?

- A) The probability that the null hypothesis is true
- B) The proportion of variance in the response variable explained by the predictors
- C) The regression coefficient for the main predictor variable

- D) The expected change in the response for a one-unit change in a predictor

> **Answer**
>
> **B.** $R^2$ is the fraction of the total response variance accounted for by the fitted model.

### 10.18.3. Question 3

What is the primary purpose of residual diagnostic plots in linear regression?

- A) To determine the statistical significance of coefficient estimates
- B) To calculate confidence intervals for predictions
- C) To check whether model assumptions are satisfied
- D) To identify which predictors should be included in the model

> **Answer**
>
> **C.** Residual plots check linearity, homoscedasticity, normality, and independence.

### 10.18.4. Question 4

A fitted linear model has one predictor with VIF = 12 and another with VIF = 3. What does this indicate?

- A) The first predictor is a better predictor than the second.
- B) The first predictor is highly collinear with other predictors; its coefficient will have inflated standard error.
- C) The second predictor should be removed.
- D) The model is over-fit.

Answer

**B.** VIF $> 10$ is usually a problem; the first predictor's coefficient is imprecisely estimated because of collinearity with other predictors. Consider dropping it, combining it with a correlated predictor, or using ridge regression.

### 10.18.5. Question 5

Your residual plot shows a clear curve. Which diagnosis is most appropriate?

- A) Heteroscedasticity; use robust standard errors.
- B) Non-linearity in the mean function; add a polynomial or transformation, or consider a non-linear model.
- C) Non-normal residuals; transform the outcome.
- D) Influential observations; remove outliers.

Answer

**B.** A curved residual pattern indicates the linear mean function is misspecified. Fit a polynomial term, log or Box-Cox the outcome, or use a generalised additive model (`mgcv::gam`). Robust SEs do not fix a wrong mean function.

## 10.19. Prerequisites answers

1. Linearity of the mean function, independence of observations, constant error variance (homoscedasticity), and, for inference, approximate normality of the errors. Point estimates are unbiased under linearity and independence alone; CIs and p-values additionally require homoscedasticity and normality (or a large enough sample that the CLT rescues normality).
2. $R^2$ is the proportion of variance in the response explained by the predictors. $R^2 = 0.7$ means the model accounts for 70% of the total

variation in the response. It does *not* measure goodness of fit in any absolute sense: a high $R^2$ does not rule out bias, non-linearity, or other misspecifications; a low $R^2$ does not rule out a correctly specified model in a high-noise setting.

3. Residual diagnostics check whether the model's assumptions (linearity, homoscedasticity, normality, independence) are reasonable. Systematic patterns in the residuals signal model misspecification: a curve signals non-linearity, a funnel signals heteroscedasticity, heavy tails on the Q-Q plot signal non-normality, and autocorrelation in the residuals (not shown by `plot(fit)`) signals violation of independence.

# 11. Generalized Linear Models

## 11.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 11.17.

1. What distinguishes Generalized Linear Models from ordinary linear models?
2. What role does the link function play in a GLM, and what is the canonical link for a binomial (0/1) outcome?
3. Write the R code to fit a logistic regression of a binary outcome on two continuous predictors using `glm()`.

## 11.2. Learning objectives

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

- State the three components of a GLM: random (exponential- family distribution), systematic (linear predictor), and link function.
- Fit binomial, Poisson, quasi-Poisson, and negative binomial models with `glm()` / `MASS::glm.nb()` and interpret the output on both the link and response scales.
- Translate logistic regression coefficients between log-odds, odds ratios, and predicted probabilities.
- Translate Poisson coefficients into incidence-rate ratios.
- Implement iteratively re-weighted least squares (IRLS) from scratch for logistic regression, and verify that it converges to the same answer as `glm()`.
- Diagnose overdispersion and respond with a quasi-likelihood fit or a negative binomial alternative.

- Use `emmeans::emmeans()` to report adjusted means, predicted probabilities, and pairwise contrasts on the response scale.

## 11.3. Orientation

Most outcomes in biomedical research are not continuous and normal. Binary (treatment success, disease status), count (adverse events, hospital visits), proportion, and time-to-event data are the norm. GLMs extend linear models to all of these within a single unified likelihood-based framework.

The framework was introduced by Nelder and Wedderburn in 1972 and has become standard enough that `glm()` feels like a minor extension of `lm()`. The conceptual leap is real nonetheless: least squares gives way to maximum likelihood, the normal distribution gives way to the exponential family, and a *link function* intervenes between the linear predictor $\eta = X\beta$ and the mean response $\mu$.

This chapter builds the machinery by hand on logistic regression (the most common GLM in biostatistics), then hands off to `glm()` for production use. Poisson regression and overdispersion get equally careful treatment because ignoring overdispersion is among the most common GLM mistakes in applied research.

## 11.4. The statistician's contribution

GLMs turn categorical and count outcomes into regression problems. What the GLM does *not* do is turn them into normal-distribution problems. That is both the opportunity (new modelling territory) and the risk (coefficients mean new things, assumptions must be checked differently).

**Which family, which link.** Binary outcome, logit link. Count outcome with variance near mean, Poisson with log link. Count with variance much larger than mean, quasi-Poisson or negative binomial. Positive continuous (costs, times), gamma with log link. Proportions of successes out of a known denominator, binomial with the denominator as weights. The choice is not mechanical; it depends on the data-generating process. Count data with

structural excess zeros (many patients who do not use the health service at all) calls for a zero-inflated model, not a Poisson.

**Interpretation on which scale.** A logistic coefficient of 0.693 is 'a log-odds ratio of 0.693', or 'an odds ratio of 2', or 'a change in probability from 0.5 to 0.67'. All three are numerically identical; they convey very different impressions to non-statistical readers. The statistician's job is to pick the scale that matches the audience and the question. For clinical communication, predicted probabilities or risk differences at clinically meaningful covariate values are usually clearer than odds ratios.

**Overdispersion is a conditional error.** In Poisson regression, if $\mathrm{Var}(Y) > E(Y)$, the standard errors printed by `glm()` understate uncertainty. P-values and CIs are over-confident. Detecting this is the analyst's job, not the software's. The fix is quasi-Poisson (same coefficients, corrected SEs) or negative binomial (different likelihood, usually better AIC).

**Separation in logistic regression.** When a predictor perfectly separates the outcome (all $y = 1$ for $x > 0$, all $y = 0$ otherwise), the MLE does not exist; the algorithm converges to coefficients that are numerically huge and technically undefined. R will produce a warning and a 'fit' with infinite-like coefficients and standard errors. The right responses are Firth penalised likelihood (`brglm2::brglm()` or `logistf::logistf()`), informative priors (Bayesian logistic regression), or acknowledging the data simply cannot support the analysis. Ignoring the warning is not an option.

These judgement calls are what distinguishes GLM analysis from mechanical output of `glm(...)`.

## 11.5. The GLM framework

A GLM has three components:

1. **Random component.** The response $Y_i$ has a distribution in the exponential family (binomial, Poisson, Gaussian, gamma, inverse Gaussian, …). Each family is characterised by a mean $\mu_i = E(Y_i)$ and a variance function $V(\mu_i)$ that relates variance to mean.
2. **Systematic component.** A linear predictor $\eta_i = \mathbf{x}_i^T \beta$, exactly as in linear models.

3. **Link function.** A monotone function $g$ connecting the two: $g(\mu_i) = \eta_i$, or equivalently $\mu_i = g^{-1}(\eta_i)$.

Standard linear models are the special case: Gaussian family with the identity link. The key innovations are (a) the freedom to choose a distribution matching the outcome type and (b) the link function that lets the linear predictor vary over the real line even when $\mu_i$ is constrained to $[0, 1]$ or $(0, \infty)$.

The *canonical* link for a family arises naturally from its exponential-family structure:

| Family | Canonical link | Response scale |
|---|---|---|
| Gaussian | identity | real |
| binomial | logit | probability $\in [0, 1]$ |
| Poisson | log | non-negative count |
| gamma | inverse | positive continuous |
| inverse Gauss. | $1/\mu^2$ | positive continuous |

Canonical links are computationally convenient (they simplify the IRLS algorithm) and often clinically interpretable. Non- canonical links (probit for binomial, identity for Poisson) are occasionally used but rarely the default.

Parameter estimation is by maximum likelihood, not least squares. Except in the Gaussian case, no closed form exists; `glm()` uses iteratively re-weighted least squares (IRLS).

## 11.6. Logistic regression

For a binary outcome $Y_i \in \{0, 1\}$ with probability $p_i = P(Y_i = 1)$, logistic regression assumes

$$\text{logit}(p_i) = \log \frac{p_i}{1 - p_i} = \mathbf{x}_i^T \beta.$$

Equivalently, $p_i = \text{expit}(\mathbf{x}_i^T \beta) = 1/(1 + e^{-\mathbf{x}_i^T \beta})$.

In R:

```
library(MASS)                 # for birthwt data
fit <- glm(low ~ age + lwt + smoke,
           family = binomial(link = "logit"),
           data = birthwt)
summary(fit)
```

The `family = binomial` argument alone defaults to the logit link; the explicit `link = "logit"` is for clarity.

### 11.6.1. Interpreting coefficients

The coefficient $\beta_j$ on predictor $x_j$ represents the change in log-odds associated with a one-unit increase in $x_j$, holding the other predictors fixed.

Three scales to translate between:

- **Log-odds.** The coefficient itself. Convenient for the arithmetic: a sum of coefficients equals the total log- odds change. Rarely clinically interpretable.
- **Odds ratio.** $\exp(\beta_j)$. A one-unit change in $x_j$ multiplies the odds of $Y = 1$ by this factor. Odds ratio of 2: odds double. OR of 0.5: odds halve. This is the conventional reporting scale in epidemiology.
- **Probability.** $p = \text{expit}(\mathbf{x}^T\beta)$. Nonlinear: the effect of a fixed change in $x$ on $p$ depends on the starting value of $p$. For two typical patients with different baseline risks, the same OR implies different absolute risk differences.

```
broom::tidy(fit, conf.int = TRUE, exponentiate = TRUE)
```

With `exponentiate = TRUE`, `tidy()` returns odds ratios and their CIs.

For predicted probabilities:

```
new_data <- data.frame(age = 25, lwt = 120, smoke = 1)
predict(fit, newdata = new_data, type = "response")
#> [1] 0.447
```

`type = "response"` transforms from log-odds (the default `type = "link"`) to probability. This is the prediction most useful for clinical communication.

Check your understanding: odds ratio vs. risk difference

**Question.** Two patient groups have baseline risks of disease of 5% and 50%. A covariate increases the odds by a factor of 2 (OR = 2) in both groups. How do the predicted probabilities change?
**Answer.**
Group A: baseline odds = 0.05/0.95 0.053; new odds = 2 × 0.053 = 0.105; new probability = 0.105/(1 + 0.105) 0.095. Change: 0.05 → 0.095, a risk increase of 4.5 percentage points.
Group B: baseline odds = 0.50/0.50 = 1; new odds = 2; new probability = 2/3 0.667. Change: 0.50 → 0.667, a risk increase of 16.7 percentage points.
The same OR of 2 produces a risk difference of 4.5 points in group A and 16.7 points in group B. This is why reporting only the OR is misleading for clinical interpretation: the clinical impact depends on the baseline risk. For common outcomes, odds ratios substantially overstate relative risk; for rare outcomes, OR and RR converge.

### 11.6.2. Separation and the MLE not existing

When a predictor perfectly separates the outcome, maximum likelihood fails:

```
x <- c(1, 2, 3, 4, 5, 6)
y <- c(0, 0, 0, 1, 1, 1)
glm(y ~ x, family = binomial)
# Warning message: glm.fit: fitted probabilities numerically 0 or 1 occ
```

Symptoms: huge coefficient estimates (e.g., 30+ for a covariate on a reasonable scale), huge standard errors, and a warning. Remedies:

- **Firth penalised likelihood.** `brglm2::brglm()` or `logistf::logistf()` adds a small penalty that makes the MLE exist and finite. Usually the right default for small- sample logistic regression.

- **Bayesian logistic regression.** Weakly-informative prior on coefficients regularises. `rstanarm::stan_glm()` is convenient.
- **Exact conditional logistic regression.** For very small samples.
- **Admit the data cannot support the analysis.** If a covariate perfectly predicts outcome, you cannot separate its effect from the outcome itself; no statistical method will rescue this.

## 11.7. IRLS from scratch

The standard algorithm for fitting a GLM is iteratively re-weighted least squares. At each iteration, construct a working response $z$ and weights $W$ such that the MLE update is a weighted least squares step.

For logistic regression:

$$z_i = \eta_i + (y_i - p_i)/(p_i(1-p_i)), \quad w_i = p_i(1-p_i).$$

The update is $\beta^{(t+1)} = (X^T W X)^{-1} X^T W z$, which is exactly a weighted least squares fit.

```
irls_logistic <- function(X, y, tol = 1e-8, max_iter = 100) {
  beta <- rep(0, ncol(X))
  for (it in seq_len(max_iter)) {
    eta <- X %*% beta
    p   <- plogis(eta)
    w   <- as.numeric(p * (1 - p))
    z   <- eta + (y - p) / w
    beta_new <- solve(crossprod(X, w * X), crossprod(X, w * z))
    if (max(abs(beta_new - beta)) < tol) break
    beta <- beta_new
  }
  beta
}

set.seed(1)
n <- 500
X <- cbind(1, matrix(rnorm(n * 2), n, 2))
```

```
p <- plogis(X %*% c(-0.5, 0.8, -0.3))
y <- rbinom(n, 1, p)

beta_irls <- irls_logistic(X, y)
beta_glm  <- coef(glm(y ~ X[, -1], family = binomial))

cbind(irls = drop(beta_irls), glm = beta_glm)
```

The two implementations produce the same coefficients to roughly machine precision. `glm()` includes additional bookkeeping (deviance, null deviance, AIC) and numerical safeguards (step halving, convergence diagnostics), but the core algorithm is what the snippet above does.

## 11.8. Poisson regression

For count data $Y_i \in \{0, 1, 2, ...\}$, assume $Y_i \sim \text{Poisson}(\mu_i)$ with $\log \mu_i = \mathbf{x}_i^T \beta$:

```
fit <- glm(count ~ age + condition + insurance,
           family = poisson(link = "log"),
           data = visits)
summary(fit)
```

Coefficients represent changes in log expected count. Exponentiating gives **incidence rate ratios (IRRs)**: $\exp(\beta_j)$ is the multiplicative change in expected count per unit increase in $x_j$. An IRR of 1.5 means expected count increases by 50%.

Offsets: when counts are rates (events per person-year, per 1000 patients, etc.), include the log of the exposure as an offset:

```
glm(events ~ treatment + age + offset(log(person_years)),
    family = poisson, data = trial)
```

The offset forces a coefficient of 1 on `log(person_years)`, which turns the linear predictor into a log-rate rather than a log-count. Coefficients on other predictors then represent rate ratios.

### 11.8.1. Overdispersion

The Poisson distribution is defined by the single parameter $\mu$, with $\text{Var}(Y) = \mu$. Real count data often shows $\text{Var}(Y) > \mu$, overdispersion, usually because of unmeasured heterogeneity, clustering, or contagion effects.

Overdispersion does not bias the point estimates but makes standard errors too small, p-values too small, and CIs too narrow. Detecting it is the analyst's responsibility.

The Pearson dispersion statistic:

$$\hat{\phi} = \frac{1}{n-p} \sum_i \frac{(y_i - \hat{\mu}_i)^2}{\hat{\mu}_i}.$$

```
phi <- sum(residuals(fit, type = "pearson")^2) / fit$df.residual
phi
```

Under a well-specified Poisson model, $\hat{\phi} \approx 1$. Values of 1.5 or higher suggest overdispersion; 2 or higher is almost certainly a problem. (Under-dispersion, $\phi < 1$, also occurs but is much less common.)

Remedies:

**Quasi-Poisson.** Same coefficients, but standard errors scaled by $\sqrt{\hat{\phi}}$:

```
fit_q <- glm(count ~ age + condition, family = quasipoisson,
             data = visits)
```

Easy. The catch: quasi-Poisson is a quasi-likelihood model, so AIC is not defined in the usual sense, making model comparison awkward.

**Negative binomial.** A full likelihood that accommodates overdispersion through an extra dispersion parameter $\theta$:

```
library(MASS)
fit_nb <- glm.nb(count ~ age + condition, data = visits)
```

Coefficients differ slightly from Poisson / quasi-Poisson; standard errors reflect the overdispersion properly. AIC and likelihood ratio tests work. For most applications with overdispersed counts, negative binomial is the better default.

Check your understanding: diagnosing overdispersion

**Question.** You fit `glm(events ~ x, family = poisson)`. The Pearson dispersion statistic is 3.2. What does this indicate, and what would you do?
**Answer.**
The residual variance is roughly 3.2× the mean, so the Poisson's equidispersion assumption fails strongly. The coefficients from this Poisson fit are still roughly unbiased, but the standard errors, p-values, and CIs are too narrow by a factor of about $\sqrt{3.2} \approx 1.8$. Re-fit with either `quasipoisson` (keeps the coefficients, scales SEs by $\sqrt{\hat{\phi}}$) or, preferably, `MASS::glm.nb()` for a proper negative binomial likelihood that handles overdispersion and supports AIC-based model comparison. The diagnosis is non-optional: reporting the over-confident Poisson SEs without correction is a methodological error.

## 11.9. GLM diagnostics

`plot(fit)` on a `glm` object produces adapted versions of the linear-model diagnostic plots, but the interpretation is different in important ways.

**Residual types:**

- **Response residuals** $(y_i - \hat{\mu}_i)$ correspond to raw residuals in linear models but have non-constant variance for GLMs. Limited diagnostic value.
- **Pearson residuals** $(y_i - \hat{\mu}_i)/\sqrt{V(\hat{\mu}_i)}$ rescale by the estimated standard deviation. Approximately constant variance under a well-specified model.
- **Deviance residuals** are the signed square roots of per-observation contributions to the deviance. These approximate $N(0, 1)$ for a correctly-specified model and are the basis for GLM diagnostic plots.

- **Working residuals** arise from IRLS and are useful for identifying computational issues.

Use `residuals(fit, type = "deviance")` or `residuals(fit, type = "pearson")` explicitly.

**Binned residual plots** are often more informative than raw residuals for GLMs, especially logistic regression where raw residuals are heavily constrained by the binary outcome. `arm::binnedplot()` provides these.

**Hosmer-Lemeshow test** assesses calibration for logistic regression by comparing observed and predicted event counts within risk deciles. Available via `ResourceSelection::hoslem.test()`. Controversial as a test (can fail to reject for badly-calibrated models with small $n$, and reject for well-calibrated models with large $n$); better used as a diagnostic plot than a significance test.

## 11.10. Predictions, contrasts, and emmeans

Predicted probabilities, rate ratios, and marginal effects are where GLM output becomes useful for readers. The `emmeans` package handles the translation from coefficients to meaningful quantities:

```
library(emmeans)

fit <- glm(low ~ age + smoke + race, family = binomial,
           data = MASS::birthwt)

# adjusted predicted probabilities by smoke, averaging over age and
emmeans(fit, "smoke", type = "response")
#>  smoke  prob     SE  df asymp.LCL asymp.UCL
#>      0 0.275 0.0427 Inf     0.198     0.367
#>      1 0.477 0.0583 Inf     0.366     0.592

# pairwise contrast
emmeans(fit, "smoke", type = "response") |> pairs()
```

`emmeans` computes the adjusted predictions at specific covariate values (or averaged over them), on the response scale if `type = "response"`, and applies the delta method for standard errors. For categorical predictors, pairwise contrasts on the link scale (risk ratios, odds ratios) or response scale (risk differences) are one function call away.

This is substantially more useful for clinical reporting than raw coefficient tables, and more trustworthy than hand-computing contrast SEs with the delta method.

For more flexible marginal-effects computation across model classes (GLMs, GAMs, Bayesian models, machine learning models, custom likelihoods), the `marginaleffects` package by Vincent Arel-Bundock has emerged as a strong general-purpose alternative to `emmeans`. It supports a wider model-class menu and a clean API for predictions, comparisons, and slopes; consider it when `emmeans` does not directly handle your model.

## 11.11. Collaborating with an LLM on GLMs

LLMs handle GLM code well; they handle GLM interpretation variably.

**Prompt 1: fitting and interpreting.** Describe the data and question, and ask: 'fit an appropriate GLM, report coefficients on the clinically relevant scale (odds ratio / rate ratio / predicted probability), and interpret each effect in plain language.'

*What to watch for.* Reference-level confusion, in both directions: the LLM may mis-identify which factor level is the reference, or may confuse log-odds with odds with probability. Any interpretation using the word 'odds' should be read with particular care; every sentence should clarify whether it is talking about odds or odds *ratio* or probability.

*Verification.* For a sample covariate pattern, compute the predicted probability by hand using `plogis(xb)` and compare to what the LLM claims the coefficients imply. If the LLM's narrative disagrees with the arithmetic, the narrative is wrong.

**Prompt 2: diagnosing overdispersion.** Paste the Pearson residuals or the dispersion statistic and ask: 'is this overdispersed, and what should I do?'

*What to watch for.* Correct diagnosis (threshold of roughly $\phi > 1.5$). Common wrong suggestions: 'use robust standard errors' (does not address overdispersion per se), or 'add more covariates' (may help if overdispersion is due to omitted variables, but often does not). The standard answers are quasi-Poisson or negative binomial.

*Verification.* Fit both Poisson and negative binomial; if the coefficients are similar and the NB's dispersion parameter is finite, the NB is the defensible model.

**Prompt 3: handling separation.** Describe the symptom (huge coefficients, convergence warning, one predictor perfectly classifying outcome) and ask: 'what's going on and how should I fit the model?'

*What to watch for.* The correct answer is Firth penalised likelihood (`brglm2::brglm()` or `logistf::logistf()`), not 'drop the problem predictor' or 'increase the sample size'. If the LLM suggests ridge regression or adds an arbitrary prior without explaining the purpose, ask it to be more specific.

*Verification.* Fit both the plain `glm` and the Firth variant; compare coefficients. The Firth version should produce finite coefficients with reasonable SEs.

## 11.12. Worked example: birthweight study

```
library(MASS)
data(birthwt)

fit <- glm(low ~ age + lwt + smoke + ht + ui,
           family = binomial,
           data = birthwt)

# tidy output on OR scale
broom::tidy(fit, conf.int = TRUE, exponentiate = TRUE)

# adjusted predicted probability for smoker vs. non-smoker
```

```r
library(emmeans)
emmeans(fit, "smoke", type = "response")

# diagnostics
par(mfrow = c(2, 2))
plot(fit)
par(mfrow = c(1, 1))

# influence
which.max(cooks.distance(fit))
```

The interpretation should focus on odds ratios (or predicted probabilities) for the clinically relevant predictors, with CIs, and note any observations with high Cook's distance that materially affected the estimate.

## 11.13. Principle in use

Three habits define defensible GLM practice:

1. **Match the family to the outcome, and the link to the family.** Binary → binomial + logit. Count → Poisson + log, or negative binomial if overdispersed. Positive continuous → gamma + log. Deviating from these defaults requires a specific reason.
2. **Report on the clinically relevant scale.** Odds ratios, rate ratios, or predicted probabilities — whichever a clinical reader can use. Raw coefficient tables on the link scale belong in the supplement, not the paper.
3. **Always check the dispersion for Poisson fits.** Overdispersion is the commonest silent failure mode in count-data analysis.

## 11.14. Exercises

1. Using `MASS::birthwt`, fit `glm(low ~ age + lwt + smoke, family = binomial)`. Reproduce the coefficients from scratch via IRLS. Verify agreement to at least 6 decimal places.

2. Simulate Poisson count data with varying degrees of overdispersion (multiplicative noise on $\mu$). Fit Poisson and quasi-Poisson; compare the coefficients and SEs. At what dispersion level does the quasi-Poisson SE notably exceed the Poisson SE?
3. Use `emmeans::emmeans()` to compute adjusted predicted probabilities for each level of a categorical predictor, on the response scale. Reproduce the standard errors via the delta method by hand: $\mathrm{SE}(\hat{p}) \approx |\hat{p}(1-\hat{p})| \cdot \mathrm{SE}(\hat{\eta})$.
4. Construct a logistic regression where one predictor perfectly separates the outcome. Fit with `glm()` and observe the warning. Fit with `logistf::logistf()`. Compare coefficients and SEs.
5. Fit a negative binomial model with `MASS::glm.nb()` to the same data as exercise 2. Compare AIC to the Poisson and quasi-Poisson fits.

## 11.15. Further reading

- (Legler & Roback, 2019) Chapters 4–5, the most readable book-length GLM introduction.
- (Faraway, 2016), Faraway's extending-the-linear- model is a comfortable level for practising biostatisticians.
- (McCullagh & Nelder, 1989), the canonical reference. Dense but definitive.
- (University of Wisconsin-Madison Social Science Computing Cooperative, n.d.), short GLM tutorial with R code.
- (Pennsylvania State University Department of Statistics, n.d.), detailed free treatment.

## 11.16. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 11.16.1. Question 1

What distinguishes Generalized Linear Models (GLMs) from standard linear models?

- A) GLMs can only be used with categorical predictors
- B) GLMs allow for response variables with non-normal distributions
- C) GLMs require more observations than standard linear models
- D) GLMs cannot include interaction terms between predictors

> **Answer**
>
> **B.** GLMs admit any exponential-family response (binomial, Poisson, gamma, etc.).

### 11.16.2. Question 2

What role does the link function serve in a GLM?

- A) It transforms the response variable to follow a normal distribution
- B) It connects the linear predictor to the expected value of the response variable
- C) It calculates p-values for the model coefficients
- D) It measures the goodness of fit of the model

> **Answer**
>
> **B.** The link function $g$ defines the relationship $g(\mu) = \eta = X\beta$.

### 11.16.3. Question 3

In R, which of the following correctly specifies a logistic regression model?

- A) `lm(binary_outcome ~ predictors, data = mydata)`
- B) `glm(binary_outcome ~ predictors, family = normal, data = mydata)`

- C) `glm(binary_outcome ~ predictors, family = binomial(link = 'logit'), data = mydata)`
- D) `logistic(binary_outcome ~ predictors, data = mydata)`

> **Answer**
>
> **C.** Logistic regression uses `glm()` with a binomial family and logit link.

### 11.16.4. Question 4

You fit a Poisson regression and observe a Pearson dispersion statistic of 2.8. The most appropriate response is:

- A) Accept the fit; dispersion near 3 is normal for count data.
- B) Refit with `quasipoisson` or `MASS::glm.nb()` to correct the standard errors.
- C) Remove outliers until the dispersion drops to 1.
- D) Switch to a Gaussian GLM with identity link.

> **Answer**
>
> **B.** Dispersion 1 invalidates the Poisson standard errors. Quasi-Poisson corrects the SEs without changing coefficients; negative binomial provides a proper likelihood with AIC support.

### 11.16.5. Question 5

A logistic regression produces an odds ratio of 2 for a predictor. For a patient with a baseline risk of 0.50, the new risk after a one-unit increase in the predictor is:

- A) 1.00.
- B) 0.75.
- C) 0.667.
- D) 0.60.

Answer

**C.** Baseline odds = 1; new odds = 2; new probability = 2/3 0.667. The same OR applied to a baseline of 0.05 gives a new probability of only about 0.095, an illustration of why OR alone is misleading without baseline risk context.

## 11.17. Prerequisites answers

1. GLMs allow the response to follow any member of the exponential family (binomial, Poisson, gamma, inverse Gaussian, etc.), not just Gaussian. They introduce a link function relating the linear predictor to the mean response. Parameters are estimated by maximum likelihood (via iteratively re-weighted least squares) rather than least squares.
2. The link function $g$ connects the linear predictor $\eta = X\beta$ to the expected value of the response: $g(\mu) = \eta$. The canonical link for a binomial outcome is the logit, $g(p) = \log(p/(1-p))$. The logit maps probabilities in $[0, 1]$ to the real line, making linear modelling of the transformed response sensible.
3. `glm(y ~ x1 + x2, family = binomial(link = "logit"), data = d)`. The `family = binomial` argument alone defaults to the logit link; specifying it explicitly documents the choice. For 0/1 data stored as `TRUE`/`FALSE` or numeric, `glm()` handles the coding automatically.

# 12. Mixed-Effects Models

## 12.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 12.20.

1. What is the primary limitation of repeated-measures ANOVA for analysing data with multiple sources of random variation (e.g., participants *and* items)?
2. What happens to the standard errors when you fit an ordinary linear regression to clustered (non- independent) observations?
3. In a mixed-effects model, what do random effects represent, and when should you declare a factor as random rather than fixed?

## 12.2. Learning objectives

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

- Distinguish fixed, random, nested, and crossed effects.
- Fit linear and generalised linear mixed models with `lme4::lmer()` and `lme4::glmer()` and read their output.
- Specify random intercepts (`(1 | group)`), random slopes (`(x | group)`), and uncorrelated random effects (`(1 + x || group)`).
- Explain why p-values and degrees of freedom are contested in mixed models, and use `lmerTest`, `pbkrtest`, or `parametric bootstrap` to obtain approximate tests.
- Diagnose convergence warnings and take appropriate action: rescaling, simplifying random effects, switching optimiser.
- Compute and interpret the intraclass correlation coefficient (ICC).

- Recognise crossed random effects (e.g., subjects and items) and write the corresponding `lmer()` formula.

## 12.3. Orientation

Clustered and repeated-measures data are the norm in biomedicine: patients nested in hospitals, observations nested within subjects, multiple pathology slides per patient, multiple time points per individual. Ignoring the clustering leads to incorrect standard errors, usually too small, because non-independence inflates the effective sample size used by the inference. Fitting a model that treats every cluster level as a separate parameter (one intercept per subject) wastes degrees of freedom and does not generalise to new subjects.

Mixed-effects models are the principled middle path: they treat between-cluster variation as a random component drawn from a distribution, with the variance of that distribution estimated from the data. This produces correctly-calibrated standard errors, generalisable inference, and 'partial pooling' of cluster-specific estimates toward the overall mean. The `lme4` package by Douglas Bates is the standard implementation in R.

## 12.4. The statistician's contribution

Mixed models are not harder to *fit* than ordinary linear models, `lmer(y ~ x + (1 | subject), data = d)` is trivial syntax. They are harder to *specify*, *interpret*, and *defend*. The judgement calls are where the analysis lives.

**Fixed or random?** A factor is fixed when its levels are the only ones of interest (treatment vs. control: nobody cares about a hypothetical 'treatment B' that was not in the trial). It is random when the observed levels are a sample from a population, and the inferential target is the population (subjects, hospitals, sites, items in a psycholinguistic experiment). The same variable can be either depending on the question. With many levels and sparse per-level data, random tends to be more honest; with few

levels and ample data per level, fixed may be cleaner. The decision is not purely statistical; it depends on the scientific question.

**Which random effects to include.** Barr et al. (2013) made the case for 'maximal' random effects, random slopes for every fixed effect that varies within cluster. The cost is convergence; the benefit is honest standard errors. Matuschek et al. (2017) argue for parsimony when maximal models do not converge. The right answer depends on the data: if you have 8 subjects and 1000 trials each, a random slope per subject is feasible; if you have 50 subjects and 4 trials each, it is probably not.

**ICC interpretation.** The intraclass correlation $\rho = \sigma^2_{\text{between}}/(\sigma^2_{\text{between}} + \sigma^2_{\text{within}})$ measures the proportion of total variance attributable to the cluster level. ICC = 0.05 means clusters explain 5% of variance; ICC = 0.5 means half. The implications for analysis differ: low ICC, ignoring clustering may not disastrously bias inference (though it should still be modelled); high ICC, ignoring it definitely does. Knowing the ICC of your design before publication review is much better than learning it during.

**Convergence warnings are not optional.** A non-converging mixed model has not been fit. Reporting its coefficients is reporting numerical noise. The right responses, rescale predictors, simplify random effects, switch optimiser, are documented and reproducible. Ignoring the warning and reporting anyway is a serious methodological failure.

These decisions distinguish mixed-model analyses that survive peer review from ones that do not.

## 12.5. Why ordinary regression fails on clustered data

Suppose you have repeated measurements on each of $n$ subjects: $K$ trials per subject, $nK$ observations total. A naive `lm(y ~ x, data = d)` treats all $nK$ observations as independent.

Two problems result:

1. **Standard errors too small.** Effective sample size is roughly $n/(1 + (K-1)\rho)$, not $nK$, where $\rho$ is the ICC. For $K = 10$, $\rho = 0.5$, the effective sample size is $n \cdot 10/5.5 \approx 1.8n$, not $10n$. Ignoring this inflates Type-I error from the nominal 5% to 20–30% in many designs.

2. **Misallocated variance.** A subject-specific intercept that is not modelled as a random effect leaks into the residual error term, inflating $\hat{\sigma}$ and reducing the apparent precision of within-subject effects.

The 'aggregating to subject means' alternative throws away within-subject variability, often halving statistical power. The 'separate analysis per subject' alternative does not generalise to new subjects.

Mixed models fix both problems simultaneously by modelling between-cluster variation explicitly.

Check your understanding: when ordinary regression fails

**Question.** A clinical trial has 100 patients, 50 in each arm, with 10 follow-up visits per patient. You fit `lm(blood_pressure ~ treatment, data = visits)` ignoring the patient-level clustering. The reported standard error on the treatment coefficient is 0.5. Roughly what should the standard error be from a properly specified mixed model, assuming an ICC of 0.6?

**Answer.**

The naive $n$ is 1000; the effective $n$ for the treatment effect is roughly $1000/(1+9\cdot 0.6) = 1000/6.4 \approx 156$. The naive SE understates the true SE by a factor of about $\sqrt{1000/156} \approx 2.5$. So the mixed-model SE should be roughly $0.5 \cdot 2.5 = 1.25$. The factor of 2.5 between the naive and correct SE means a treatment effect that looked highly significant ($p = 0.001$) under the naive analysis may be only marginal ($p = 0.05$) under the correct one. This is the cost of ignoring clustering.

## 12.6. Fixed vs. random effects

A *fixed* effect estimates a parameter at the levels you observed. The treatment in a clinical trial is fixed: only those treatments are of interest, and replication of the trial would use the same arms.

A *random* effect treats the observed levels as a sample from a distribution, with that distribution's parameters estimated from the data. Subjects in a longitudinal study are random: this study's 100 subjects are not the inferential target; the population they were sampled from is.

Practical heuristic: if you would replicate the experiment with new units (subjects, items, sites), and the new units would not be the same physical entities as the old, that factor is random.

Mathematically, for a random intercepts model:

$$y_{ij} = \alpha + \mathbf{x}_{ij}^T\beta + u_j + \varepsilon_{ij}, \quad u_j \sim N(0, \sigma_u^2), \quad \varepsilon_{ij} \sim N(0, \sigma^2).$$

The fixed effects ($\alpha$, $\beta$) and the two variance components ($\sigma_u^2$, $\sigma^2$) are estimated. The cluster-specific deviations $u_j$ are *predicted* (BLUPs, Best Linear Unbiased Predictors) rather than estimated, and are pulled toward zero by an amount that depends on how much data each cluster has. This is partial pooling: clusters with little data borrow strength from clusters with more data.

## 12.7. `lmer()` syntax

```
library(lme4)

# random intercept by subject
lmer(y ~ x + (1 | subject), data = d)

# random intercept and random slope on x
lmer(y ~ x + (1 + x | subject), data = d)
lmer(y ~ x + (x | subject), data = d)              # equivalent

# uncorrelated random intercept and slope
lmer(y ~ x + (1 + x || subject), data = d)
lmer(y ~ x + (1 | subject) + (0 + x | subject), data = d)  # equival

# nested random effects: clinics within regions
lmer(y ~ x + (1 | region/clinic), data = d)
lmer(y ~ x + (1 | region) + (1 | region:clinic), data = d)  # equiva

# crossed random effects: subjects and items
lmer(y ~ x + (1 | subject) + (1 | item), data = d)
```

The | separator: random effects formula on the left, grouping factor on the right. The double bar || removes the correlation between random intercept and slope.

The choice of random-effects structure is consequential. Random slopes for fixed effects that vary within cluster are sometimes essential for honest inference (Barr et al., 2013). They are also computationally hard: a model with many random slopes may not converge.

## 12.8. REML vs. ML

`lmer()` defaults to REML (restricted maximum likelihood) estimation, which produces unbiased estimates of variance components. For comparing nested models that differ in *fixed* effects, refit with ML (`REML = FALSE`):

```
m1 <- lmer(y ~ x1 + (1 | g), data = d, REML = FALSE)
m2 <- lmer(y ~ x1 + x2 + (1 | g), data = d, REML = FALSE)
anova(m1, m2)
```

`anova()` automatically refits with ML when comparing fixed-effect structures. For comparing models that differ in *random* effects, REML is correct (and is what `anova()` uses if both models are REML-fitted).

## 12.9. p-values and degrees of freedom

`summary(lmer_fit)` does *not* show p-values for fixed effects, because the appropriate reference distribution is not exactly known. The denominator degrees of freedom for the t-statistic depend on the random-effects structure, the data, and the test in subtle ways.

Three approaches:

1. **lmerTest::lmer()** wraps `lmer()` and adds Satterthwaite-approximated degrees of freedom and p-values:

```
library(lmerTest)
fit <- lmer(y ~ x + (1 | subject), data = d)
summary(fit)        # now includes p-values
```

Reasonable default for moderate sample sizes.

2. **pbkrtest::KRmodcomp()** uses the Kenward-Roger approximation, generally more accurate for small samples but more expensive.
3. **Likelihood ratio tests via parametric bootstrap.** `pbkrtest::PBmodcomp()` is the gold standard for small- sample inference but slow.

For variance components, likelihood-ratio tests have a boundary issue: the null hypothesis $\sigma^2 = 0$ places the parameter on the boundary of the parameter space, and the chi-square reference distribution is wrong (it should be a 50:50 mixture with a point mass at 0). The standard fix is to halve the reported p-value.

## 12.10. Predicted random effects (BLUPs)

`ranef(fit)` returns the cluster-specific deviations from the fixed-effect estimates. These are useful for:

- Plotting cluster-specific fits.
- Identifying outlier clusters (extreme BLUPs).
- Reporting between-cluster variation.

```
library(lme4)
fit <- lmer(Reaction ~ Days + (Days | Subject),
            data = sleepstudy)

re <- ranef(fit)$Subject       # data frame of intercept and slope BL
head(re)
```

BLUPs are *shrunk* toward zero relative to the cluster- specific OLS estimates. The shrinkage is greater for clusters with less data. This shrinkage is a feature, not a bug: it stabilises estimates for sparsely-sampled clusters by borrowing information from the population.

## 12.11. ICC: how much clustering is there?

For a random-intercept model, the intraclass correlation:

$$\rho = \frac{\sigma_u^2}{\sigma_u^2 + \sigma^2}.$$

```
vc <- as.data.frame(VarCorr(fit))
icc <- vc$vcov[1] / sum(vc$vcov)
icc
```

Or use `performance::icc(fit)` for a tidier interface. The ICC quantifies how much of the residual variance is between-cluster vs. within-cluster:

- ICC = 0: clusters do not differ; ordinary regression would be fine (rare).
- ICC = 0.05–0.2: typical of randomised trials with modest cluster effects.
- ICC = 0.3–0.6: typical of biological measurements with strong subject differences.
- ICC = 0.9: nearly all variance is between clusters; you may have very few independent observations.

Reporting the ICC alongside fixed-effect estimates makes the strength of clustering transparent.

Check your understanding: ICC and design effect

**Question.** Your data have ICC = 0.4 with 20 patients in each cluster. By what factor is the effective sample size reduced compared to the nominal sample size?
**Answer.**
Design effect $= 1 + (K-1)\rho = 1 + 19 \cdot 0.4 = 8.6$. Effective sample size is the nominal divided by the design effect: if you have 100 nominal observations (5 clusters of 20), the effective sample size is roughly 12. This huge inflation factor is why ignoring clustering produces catastrophic over-confidence: the analysis behaves as though it had eight times more independent information than it actually does.

## 12.12. Convergence warnings

`lmer` warnings are common with realistic data and complex random-effects structures. Common messages:

- 'Model failed to converge with max|grad| …'
- 'Model is nearly unidentifiable: large eigenvalue ratio'
- 'singular fit: see ?isSingular'

Each indicates the optimiser stopped before finding a clearly-defined optimum. Reporting the model output without addressing the warning is unsafe.

Standard remedies, roughly in order:

1. **Rescale predictors.** Continuous predictors with wildly different units inflate condition number. Centre and scale (`scale()` or `standardize()`).
2. **Simplify random effects.** Drop random slopes for predictors that vary little within cluster, or use `||` to remove correlations between random effects.
3. **Switch optimiser.** `control = lmerControl(optimizer = "bobyqa")` often succeeds where the default fails.
4. **Increase iteration limit.** `optCtrl = list(maxfun = 1e5)`.
5. **Use the singular-fit check.** `isSingular(fit)` indicates that one or more variance components have collapsed to (near) zero. The corresponding random effect should probably be removed.

If none of these work, the data may not be rich enough to support the model you specified. Simplify the random- effects structure, starting with the slopes that have the smallest estimated variance, until you have a converged, interpretable model. Document what you tried.

## 12.13. GLMMs with `glmer()`

`glmer()` extends mixed models to non-normal outcomes:

```
glmer(y ~ x + (1 | subject),
      family = binomial, data = d)            # logistic GLMM

glmer(count ~ x + (1 | subject),
      family = poisson, data = d)              # Poisson GLMM
```

Computational cost is substantially higher than `lmer`: the random-effects integration has no closed form for non- normal outcomes. The default Laplace approximation works well for moderate ICC and reasonable cluster size; for small clusters or high ICC, increase `nAGQ` (adaptive Gauss-Hermite quadrature points): `nAGQ = 7` is often a good compromise.

Convergence problems are even more common in GLMMs than LMMs. The same remedies apply, plus the consideration that binary GLMMs with sparse data often have boundary issues where one or more random effects have estimated variance near zero, a flag for re-thinking the model.

For overdispersed counts, `MASS::glmmPQL()` or `glmmTMB::glmmTMB()` provide more flexible alternatives; the latter is faster and supports a richer family of distributions (negative binomial, beta-binomial, zero- inflated, hurdle).

## 12.14. Worked example: sleep deprivation study

```
library(lme4)
library(lmerTest)

data(sleepstudy)

# subjects measured over 10 days; reaction time as response
fit <- lmer(Reaction ~ Days + (Days | Subject),
            data = sleepstudy)
summary(fit)

# fixed effect: average reaction time increases by ~10ms/day
# random effects: subjects vary in baseline RT and in slope on Days
```

```
# ICC of the intercept
performance::icc(fit)

# subject-specific predicted slopes
ranef(fit)$Subject
```

The fixed effects table shows the population-average effect of sleep deprivation. The random-effects variance components show how much subjects vary in baseline RT and in their day-by-day slopes. The correlation between the two random effects (often substantial in such data) indicates whether subjects with high baselines also have steep slopes.

## 12.15. Collaborating with an LLM on mixed models

LLMs handle the syntax of `lmer` reasonably well; they handle the conceptual judgements much less reliably.

**Prompt 1: writing the formula.** Describe the data structure (subjects nested within sites; items crossed with subjects; etc.) and ask: 'write the `lmer` formula for the correct random-effects structure.'

*What to watch for.* Confusion between nested and crossed random effects is the most common LLM error. For 'subjects within sites', the formula is `(1 | site/subject)` if subject IDs are reused across sites, or `(1 | site) + (1 | subject)` if subject IDs are unique. For 'subjects answering items', subjects and items are crossed, so `(1 | subject) + (1 | item)`.

*Verification.* Ask the LLM to expand its proposed formula into the explicit form (e.g., `(1 | site/subject)` → `(1 | site) + (1 | site:subject)`). If the expansion is correct, the formula is correct.

**Prompt 2: handling convergence warnings.** Paste the warning message and ask: 'what is happening, and what should I try?'

*What to watch for.* The standard suite of remedies. If the LLM suggests 'use lmerTest', that is not a fix for convergence; it is just a way to get p-values once the model converges. If it suggests 'just use the model anyway', push back: a non-converging model is not a fit.

*Verification.* Apply each suggested remedy in turn. Track which one resolves the warning. Often the answer is to rescale predictors and to drop a random slope; rarely is it any single one of the more exotic options.

**Prompt 3: interpreting random-effect variance.** Paste the `summary(fit)` output and ask: 'how should I interpret the random-effects variance components?'

*What to watch for.* The variance components are reported as standard deviations and variances. The interpretation is on the response scale (or, for GLMMs, on the link scale). Make sure the LLM does not confuse the random- effect variance ($\sigma_u^2$) with the residual variance ($\sigma^2$) or the fixed-effect coefficients.

*Verification.* The ICC is a sanity check: compute it from the variance components and confirm it falls in a plausible range for your design.

## 12.16. Principle in use

Three habits define defensible mixed-model practice:

1. **Specify random effects deliberately.** Identify which factors are sampled (random) and which are fixed. Include random slopes for within-cluster fixed effects when the data support them; drop them when convergence demands it.
2. **Report the ICC.** The strength of clustering is part of the result, not a nuisance. Readers need to know whether you are reporting a tightly-clustered design with little independent information or a loosely- clustered one.
3. **Take convergence warnings seriously.** A non-converging mixed model has not been fit. Document what you did to address the warning; ignoring it is not a defensible choice.

## 12.17. Exercises

1. Using `lme4::sleepstudy`, fit `lmer(Reaction ~ Days + (Days | Subject))`. Extract the fixed effects, the variance components, and

the BLUPs. Plot the subject-specific fitted lines on top of the data.
2. Compare the fixed-effect slope from exercise 1 to the slope from `lm(Reaction ~ Days)` ignoring subject. Explain why they differ.
3. Simulate data from a random-intercept logistic GLMM with a known ICC. Fit the model with `glmer()` and check whether the fitted ICC is close to the truth.
4. Refit the sleepstudy model with uncorrelated random intercepts and slopes (`||`). Compare AIC. Does the correlation matter?
5. Take the sleepstudy data and induce a non-convergence warning by including a random slope for a constructed covariate that has no within-subject variation. Apply the standard remedies in order and document which one resolves the warning.

## 12.18. Further reading

- (Bates et al., 2015), the authoritative JSS paper on `lme4`.
- (Brown, 2021), recent applied LMM introduction in R.
- (McCulloch et al., 2008), the standard textbook on GLMs and GLMMs.
- (Bolker, n.d.), Ben Bolker's GLMM FAQ; indispensable for diagnosing convergence and specification issues.
- (Barr et al., 2013), the influential 'keep it maximal' paper.
- (Matuschek et al., 2017), the more cautious response on parsimony.

## 12.19. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 12.19.1. Question 1

What is the primary limitation of repeated-measures ANOVA when analysing data where participants respond to multiple trials?

- A) They require balanced data designs

- B) They cannot model both participant-level and item-level variability simultaneously
- C) They are computationally too expensive
- D) They require normally distributed data

> **Answer**
>
> **B.** Repeated-measures ANOVA handles a single random- factor structure. Mixed-effects models generalise to multiple crossed or nested random factors.

### 12.19.2. Question 2

What happens when observations are not independent in standard regression analysis?

- A) The model will automatically correct for this violation
- B) The results will be exactly the same as with independent observations
- C) The independence assumption is violated, leading to incorrect standard errors
- D) The model will fail to converge

> **Answer**
>
> **C.** Non-independence inflates effective sample size, producing incorrectly narrow standard errors.

### 12.19.3. Question 3

In the context of mixed-effects models, what are 'random effects'?

- A) Effects that are randomly selected by the researcher
- B) Statistical noise that should be eliminated
- C) Effects that model random variation in grouping factors (like participants)
- D) Unpredictable variables that cannot be modeled

Answer

**C.** Random effects allow inference to generalise to the population of groups from which the observed levels were drawn.

### 12.19.4. Question 4

`lmer` returns a 'singular fit' warning. The most likely interpretation is:

- A) A clear bug in `lme4`; report it.
- B) One or more variance components have collapsed to near zero, indicating a random effect that the data cannot support.
- C) The fixed effects are not identifiable.
- D) Ignore it; singular fits are still valid.

Answer

**B.** Drop the corresponding random effect (or simplify the structure) and refit. A singular fit is not a model that can be reported as-is.

### 12.19.5. Question 5

Your design has 20 clusters of size 30, with ICC = 0.3. Approximately how does the effective sample size compare to the nominal sample size of 600?

- A) The same, 600.
- B) About 65 (effective).
- C) About 300 (effective).
- D) About 60 (effective).

Answer

**B.** Design effect $= 1 + 29 \cdot 0.3 = 9.7$. Effective $n = 600/9.7 \approx 62$. This dramatic reduction is the standard motivation for mixed models: the naive sample size enormously overstates the information content of clustered data.

## 12.20. Prerequisites answers

1. Repeated-measures ANOVA cannot simultaneously model multiple crossed or nested sources of random variation (e.g., participants *and* items). It assumes a single random-factor structure and balanced data, both of which are commonly violated in modern designs.
2. Non-independence leaves the point estimates approximately unbiased but produces incorrect standard errors, typically too small, so confidence intervals are too narrow and p-values too optimistic. The inflation factor is roughly $\sqrt{1 + (K-1)\rho}$ for clusters of size $K$ with ICC $\rho$. For a typical longitudinal study with ICC $= 0.5$ and 10 visits per subject, the naive standard errors are about $\sqrt{5.5} \approx 2.3$ times too small.
3. Random effects model variation in grouping factors whose levels are a sample from a larger population of interest (e.g., subjects, clinics, items). Declare a factor random when you wish to generalise inference to the population of levels, not just the specific levels observed; declare it fixed when the observed levels are themselves the inferential target. The same variable can be either, depending on the question.

# 13. Survival Analysis

## 13.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 13.18.

1. What is 'right censoring', and how does it differ from a missing outcome?
2. What does the Kaplan-Meier estimator estimate, and how does it handle censored observations?
3. In the Cox proportional hazards model $\lambda(t \mid x) = \lambda_0(t) \exp(x'\beta)$, what is the 'proportional hazards' assumption, and how do you check it in R?

## 13.2. Learning objectives

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

- Construct a `Surv()` object from an event time and an event indicator, including for time-varying covariates.
- Fit and interpret a Kaplan-Meier curve with `survfit()`, and compare survival curves with the log-rank test.
- Fit a Cox proportional hazards model with `coxph()` and interpret hazard ratios on the response scale.
- Check the proportional-hazards assumption with `cox.zph()` and diagnose violations.
- Handle time-varying covariates via the `(start, stop, event)` counting-process format.
- Recognise competing risks and use cumulative incidence functions or Fine-Gray models when they apply.

## 13.3. Orientation

Time-to-event outcomes are the defining feature of most clinical trials, most epidemiological cohort studies, and most survival analyses in cancer, cardiovascular, and infectious-disease research. The outcome is not 'did the event happen' (that is a binary outcome and would be analysed as a GLM) but 'when did it happen, if at all'.

The distinguishing challenge is *censoring*: for many participants the event is not observed before the study ends, before they withdraw, or before some other time- bounded reason for follow-up to stop. A naive analysis that drops censored observations is biased; a naive analysis that imputes their event times is more biased still. Survival methods are the principled response.

R's `survival` package, written by Terry Therneau, is the canonical implementation. It is one of the oldest and best-tested packages in R; the API has been stable for decades.

## 13.4. The statistician's contribution

Survival analysis is a domain where the mechanics (`survfit`, `coxph`, `cox.zph`) are well-supported by software, but the assumptions and the interpretation are where most analyses go wrong.

**Independent censoring is an assumption, not a default.** The Kaplan-Meier estimator and the Cox model both assume that censoring is independent of the event: participants who drop out are similar in event risk to participants who remain. This often fails. Patients who withdraw because of side effects may be at different risk of the event than those who tolerate the treatment. Hospital-administrative censoring (the patient's chart was lost) is usually independent; clinical censoring (the patient stopped attending follow-up because their disease progressed) is usually not. Detecting violations of independent censoring is rarely possible from the data alone; it is a substantive judgement.

**The proportional hazards assumption matters.** Cox regression's coefficient $\exp(\beta)$ is interpretable as a hazard ratio *only if* that ratio is

constant over time. When PH fails, and it often does for treatments whose effect wanes over time, the reported HR is an average across the follow-up period whose interpretation depends on follow-up length. Reporting a single HR for a non-PH effect is misleading.

**Hazard ratio versus risk difference.** Hazard ratios are mathematically convenient but clinically opaque. A reader who sees 'HR = 0.7, p < 0.001' wants to know: how many more patients survived to 5 years on treatment vs. control? That is a *survival difference at* $t = 5$, easily extracted from the model. Reporting only the HR misses the substantive answer.

**Time-varying covariates need the counting-process format.** A baseline covariate that the patient acquires during follow-up (e.g., a complication, a dose change) is not a baseline covariate. Treating it as one, using its final value or its baseline value with a single row per patient, biases the estimated effect. The counting- process format with one row per follow-up interval is the correct representation.

**Competing risks change everything.** If a patient can die from the cause of interest, from another cause, or remain alive at study end, naive Kaplan-Meier on cause- specific death overstates the cumulative incidence. The correct estimator is the cumulative incidence function; the correct regression model is Fine-Gray's subdistribution hazards. Forgetting this is a classic error in cardiovascular and oncology analyses.

These judgements are what distinguishes survival analysis from a mechanical execution of `coxph()`.

## 13.5. Censoring

A right-censored observation has its event time bounded *below* by the censoring time $C$: the event happens at some unknown $T > C$. The participant was followed to time $C$ without an event being observed. Common reasons:

- The study ended before the event occurred (administrative censoring).
- The participant withdrew or was lost to follow-up.
- A non-event 'absorbing' state was reached (e.g., received a transplant, was deemed ineligible).

Right censoring is the dominant pattern in survival analysis. Left censoring (event known to occur before some time) and interval censoring (event known to occur in a time interval) appear in screening and disease-progression studies; the `survival` package supports them via `Surv(..., type = ...)`.

**Independent censoring** is the standard assumption: a participant censored at time $C$ has the same event distribution beyond $C$ as a participant who is still event-free at $C$. Violations are *informative* censoring, which biases the estimator. Diagnostics are limited; the defence is mostly substantive.

## 13.6. The `Surv()` object

A `Surv()` object packages an event time and an event indicator (1 = event, 0 = censored) into a single object that survival functions can use:

```
library(survival)

# right-censored
s <- Surv(time = lung$time, event = lung$status == 2)
head(s)
#> [1] 306    455    1010+   210    883    1022+

# the '+' marks censoring
```

Conventions vary in coding the indicator. Many R datasets use `event = 1, censored = 0`. Many CDISC datasets use the opposite (`CNSR = 1` for censored, `0` for event). Read the data dictionary; the wrong coding produces an analysis on the wrong cohort.

For time-varying covariates, the `(start, stop, event)` form represents one interval of constant covariate values:

```
Surv(time = start, time2 = stop, event = event)
```

For interval-censored data:

```
Surv(time = lower, time2 = upper, type = "interval2")
```

## 13.7. Kaplan-Meier estimator

The Kaplan-Meier (product-limit) estimator estimates the survivor function $S(t) = P(T > t)$ nonparametrically, allowing for right censoring:

$$\hat{S}(t) = \prod_{t_j \le t} \frac{n_j - d_j}{n_j}$$

where the product is over distinct observed event times $t_j$, $n_j$ is the number at risk just before $t_j$, and $d_j$ is the number of events at $t_j$. Censored observations contribute to the risk set up to (but not beyond) their censoring time.

```
library(survival)
data(lung)

# overall KM
fit <- survfit(Surv(time, status == 2) ~ 1, data = lung)
plot(fit, mark.time = TRUE,
     xlab = "Days", ylab = "Survival probability")

# by sex (1 = male, 2 = female)
fit_sex <- survfit(Surv(time, status == 2) ~ sex, data = lung)
plot(fit_sex, col = c("blue", "red"))
legend("topright", legend = c("Male", "Female"),
       col = c("blue", "red"), lty = 1)
```

A more polished plot:

```
library(survminer)
ggsurvplot(fit_sex, data = lung,
           conf.int = TRUE, risk.table = TRUE,
           pval = TRUE, surv.median.line = "hv")
```

The median survival time is the time at which $\hat{S}(t)$ crosses 0.5. With CIs:

```
summary(fit_sex)$table
```

The log-rank test compares survival across groups:

```
survdiff(Surv(time, status == 2) ~ sex, data = lung)
```

It tests the null hypothesis of equal hazard at every time point, against the (unspecified) alternative that they differ. Like all rank tests, it is most powerful when the true hazard ratio is roughly constant over time, the same setting in which the Cox model is the right tool.

## 13.8. Cox proportional hazards model

The Cox model:

$$\lambda(t \mid \mathbf{x}) = \lambda_0(t) \exp(\mathbf{x}^T \beta).$$

The hazard at time $t$ is the baseline hazard $\lambda_0(t)$ multiplied by an exponential function of the covariates. Importantly, $\lambda_0(t)$ is left *unspecified* and is not estimated, only the relative hazards (the $\beta$) are.

```
fit <- coxph(Surv(time, status == 2) ~ age + sex + ph.ecog,
             data = lung)
summary(fit)
```

The output shows coefficients $\hat{\beta}_j$ on the log- hazard scale and $\exp(\hat{\beta}_j)$ as **hazard ratios**. A hazard ratio of 1.5 means the instantaneous risk of the event is 1.5× higher per unit increase in $x_j$, holding other covariates fixed.

`exp(coef(fit))` gives the HRs; `confint()` gives 95% CIs on the log-hazard scale, which exponentiate to CIs on the HR scale.

`broom::tidy()` produces a tidy data frame:

```
broom::tidy(fit, exponentiate = TRUE, conf.int = TRUE)
```

### 13.8.1. Tied event times

When multiple events occur at the same recorded time, the partial likelihood is ambiguous. `coxph()` defaults to the Efron approximation, which is accurate and fast. The Breslow alternative is faster but less accurate when ties are common. For data with extensive tying (e.g., daily follow-up on slowly-progressing disease), use exact methods (`method = "exact"`), which are the most accurate but slowest. The Efron default is the right choice for almost all clinical data.

Check your understanding: hazard ratio vs. survival difference

**Question.** A Cox model reports HR = 0.7 (95% CI 0.55, 0.89) for treatment vs. control. The 5-year survival is 65% in control and 75% in treatment. Which number should you report to a clinical reader, and why?

**Answer.**

Report both. The HR is the standard way to summarise the treatment effect for the methods-literate, and it is the quantity the regression actually estimates. The 5-year survival difference (10 percentage points) is what clinicians and patients understand. Reporting only the HR loses the absolute scale of the benefit; reporting only the difference loses the time-summary character of the HR. Modern reporting practice (e.g., CONSORT for trials) recommends both. The HR speaks to whether the treatment helps; the absolute difference speaks to how much.

## 13.9. Checking proportional hazards

`cox.zph()` computes a test based on scaled Schoenfeld residuals: under the null of proportional hazards, the scaled residuals should not trend with time.

```
zph <- cox.zph(fit)
zph
plot(zph)            # residuals vs. time, one panel per covariate
```

A small p-value for a particular covariate flags that its hazard ratio changes with time. Visual inspection of the plot is essential: a small p-value driven by one outlying time point is different from a clear monotonic trend.

When PH fails, options:

1. **Stratify** by the offending covariate. The Cox model with `+ strata(x)` allows a separate baseline hazard for each level of `x` but does not estimate its main effect. Use when the variable's effect is not of direct interest (e.g., centre in a multicentre trial).
2. **Time-varying coefficient.** Allow the coefficient on the variable to vary with time, e.g., via the `tt = function(x, t, ...)` argument in `coxph()`. This is the right move when you want to report the time pattern of the effect.
3. **Switch to AFT.** Accelerated failure time models do not assume PH; they assume the covariate accelerates or decelerates the time scale.
4. **Time-period-specific HRs.** Split follow-up at a landmark time (e.g., 1 year) and report HRs for each period.

## 13.10. Time-varying covariates

A patient's covariate may change during follow-up: a laboratory value updates, a treatment changes, a side effect develops. Treating only the baseline value as the covariate is biased.

The counting-process format expands the dataset so that each row represents an interval over which the covariate is constant:

```
id  start  stop  event  drug  age  sex
 1    0     30      0    A    65    M
 1   30     60      0    B    65    M
 1   60    150      1    B    65    M
```

Patient 1 had drug A from day 0 to 30, drug B from 30 to 150, and the event at 150. Each row is one interval.

```
fit_tv <- coxph(Surv(start, stop, event) ~ drug + age + sex,
                data = expanded)
```

`survival::tmerge()` is the canonical tool for converting wide-format longitudinal data into the counting-process format. The package vignette (`vignette("timedep", package = "survival")`) is the authoritative reference.

## 13.11. Competing risks

When a patient can die from the cause of interest, from another cause, or remain alive (or in a similar three-way state), naive Kaplan-Meier on the cause of interest, by treating death from other causes as censoring, *over*- states the cumulative incidence. The reason: the estimator implicitly assumes that the patient who died of another cause might still have died of the cause of interest later, which is impossible.

The correct nonparametric estimator is the **cumulative incidence function** (CIF):

```
library(cmprsk)
cif <- cuminc(ftime = data$time, fstatus = data$status,
              group = data$treatment)
plot(cif)
```

For regression on the *subdistribution* hazard (the hazard of the cause of interest among those who have not yet experienced any of the competing events), use the Fine-Gray model:

```
library(cmprsk)
fg <- crr(ftime = data$time, fstatus = data$status, cov1 = X,
          failcode = 1, cencode = 0)
```

The `tidycmprsk` package provides a tidier interface. Cause-specific Cox models (`coxph` with the competing event treated as censoring) are also valid but answer a different question: the rate among the still-at-risk, rather than the cumulative incidence.

Check your understanding: competing risks

**Question.** In a study of cardiovascular mortality, you treat death from any non-cardiovascular cause as censoring and use Kaplan-Meier. What is wrong with this analysis?

**Answer.**

KM with non-CV death treated as censoring assumes that patients who die of non-CV causes might still have died of CV causes later, that censoring is independent of the event of interest. With death as the censoring mechanism, this is mechanically false: a patient who has died cannot subsequently experience the event. The KM estimator therefore overstates the cumulative incidence of CV death. The correct estimator is the cumulative incidence function (CIF), which acknowledges that non-CV death removes the patient from being at risk for CV death. Use `cmprsk::cuminc` for the nonparametric estimate and `cmprsk::crr` for Fine-Gray regression. This error is widespread in older cardiovascular and oncology literature.

## 13.12. Worked example: NCCTG lung cancer trial

```
library(survival)
library(survminer)
data(lung)

# KM by sex
km <- survfit(Surv(time, status == 2) ~ sex, data = lung)
ggsurvplot(km, data = lung, conf.int = TRUE,
           risk.table = TRUE, pval = TRUE,
           legend.labs = c("Male", "Female"))
```

```r
# Cox model
fit <- coxph(Surv(time, status == 2) ~ age + sex + ph.ecog,
             data = lung)
summary(fit)
broom::tidy(fit, exponentiate = TRUE, conf.int = TRUE)

# PH check
zph <- cox.zph(fit)
zph
plot(zph)

# adjusted survival curves at typical ages, by sex
new_data <- expand.grid(age = c(50, 70), sex = c(1, 2),
                        ph.ecog = 1)
sf <- survfit(fit, newdata = new_data)
plot(sf)
```

The fixed effects table shows that being female and lower ECOG score are associated with longer survival; age has weaker effect. The PH check should be inspected before any HR is reported.

## 13.13. Collaborating with an LLM on survival analysis

Survival analysis has more silent failure modes than most modelling areas. LLMs are competent at the standard operations and prone to the standard mistakes.

**Prompt 1: writing the analysis.** Describe the data (time variable, event indicator including its coding, covariates) and ask: 'fit Kaplan-Meier curves and a Cox model, check proportional hazards, and report hazard ratios with CIs.'

*What to watch for.* The single most common error is event indicator coding: the LLM may use `status == 2` when the data have `status == 1` for events, or vice versa. Always verify by looking at the table of events and how the model counts them. Reading `summary(fit)$nevent` against the table of `status` values catches this.

*Verification.* Run the analysis and check three things: the number of events in the model output matches the data, the direction of the HRs makes clinical sense, and the log-rank p-value (if reported) is consistent with the visual KM curves.

**Prompt 2: handling time-varying covariates.** Describe a covariate that updates during follow-up and ask: 'set up the counting-process data and fit a time-varying Cox model.'

*What to watch for.* The expansion to counting-process format is error-prone; LLMs sometimes produce expansions that double-count events or have incorrect risk sets. The canonical tool is `survival::tmerge()`. If the LLM does not use it, push back.

*Verification.* After expansion, check that the total number of events is correct (one event per patient who experienced one), that no row has `start >= stop`, and that the per-patient sum of follow-up matches the original. These three checks catch most expansion bugs.

**Prompt 3: competing risks.** Describe the outcome and the competing event. Ask: 'should I use Cox or Fine-Gray, and why?'

*What to watch for.* Both are defensible answers depending on the question. Cause-specific Cox models the rate among those still at risk; Fine-Gray models the cumulative incidence. For prognosis ('what is the patient's probability of CV death?'), Fine-Gray is usually more appropriate. For aetiology ('does the treatment affect the biological pathway to CV death?'), cause-specific Cox is. The LLM should distinguish these; if it does not, ask.

## 13.14. Principle in use

Three habits define defensible survival analyses:

1. **Verify the censoring coding before any analysis.** The single most common error in applied survival work is reversed event/censoring indicators.

2. **Always check proportional hazards.** Reporting a Cox coefficient without checking PH is reporting a number whose interpretation depends on follow-up length.
3. **Report on the survival scale, not just the hazard scale.** Hazard ratios are for methods sections; absolute differences in survival probability are for results sections.

## 13.15. Exercises

1. Using `survival::lung`, fit a Cox model for overall survival on age and sex. Produce hazard-ratio estimates with 95% CIs. Check the proportional-hazards assumption and report the conclusion.
2. Simulate a two-arm trial with exponential survival, true hazard ratio of 0.7, median follow-up 18 months, and 20% loss to follow-up. Estimate the observed power at $n = 300$ total from 1000 replicates.
3. Construct a dataset with one baseline covariate that is a strong predictor of *censoring* (not the event). Show by simulation that naive KM overstates survival because censoring is informative; discuss remedies.
4. Use `survival::tmerge()` to expand a wide-format longitudinal dataset to the counting-process format. Verify the expansion did not change the total number of events or total follow-up.
5. Simulate competing risks with a 30% rate of competing death. Compare KM (treating competing as censoring), the cumulative incidence function, and a Fine-Gray regression. Quantify the bias of KM relative to CIF.

## 13.16. Further reading

- Therneau and Grambsch (2000), *Modeling Survival Data: Extending the Cox Model*, Springer, the standard extended-Cox reference; the `survival` package is its companion implementation.
- Kleinbaum and Klein (2012), *Survival Analysis: A Self-Learning Text*, 3rd ed., Springer, the most readable introduction at MS level.

- The `survival` package vignettes on CRAN, run `vignette(package = "survival")` for the full list. The 'timedep' vignette is essential for time-varying covariates.
- Andersen and Geskus (2010), 'Cumulative incidence in competing risks data and competing risks regression analysis', *Clinical Cancer Research*, a clear motivation for Fine-Gray.

## 13.17. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 13.17.1. Question 1

Right censoring of an observation means:

- A) The participant's outcome value is missing.
- B) The participant's event time is known to exceed some observed time, but is otherwise unknown.
- C) The participant is dropped from the analysis.
- D) The data were collected at the wrong time.

> **i** Answer
>
> **B.** Censoring preserves partial information ('event did not occur before time $C$') that a missing-outcome analysis would discard.

### 13.17.2. Question 2

In the Cox proportional hazards model, the coefficient $\beta$ is interpreted on the:

- A) Probability scale (a one-unit change in $x$ changes the probability of the event by $\beta$).
- B) Hazard scale (a one-unit change in $x$ multiplies the hazard by $\exp(\beta)$).

- C) Time scale (a one-unit change in $x$ changes median survival by $\beta$ time units).
- D) Odds scale.

> **Answer**
>
> **B.** $\exp(\beta)$ is the hazard ratio. The interpretation is multiplicative on the instantaneous hazard, conditional on having survived to that time.

### 13.17.3. Question 3

`cox.zph()` produces a small p-value for one of your covariates. Which is the most appropriate response?

- A) Drop the covariate from the model.
- B) Acknowledge that the proportional-hazards assumption is violated for that covariate, inspect the scaled-Schoenfeld plot, and consider stratification or a time-varying coefficient.
- C) Add an interaction with another covariate.
- D) Switch to a logistic regression.

> **Answer**
>
> **B.** A PH violation does not mean the covariate is unimportant; it means its effect changes with time. Remedies preserve the variable while modelling the time dependence honestly.

### 13.17.4. Question 4

In a study of cardiovascular mortality, you treat death from non-cardiovascular causes as censoring and apply Kaplan-Meier. The estimated cumulative incidence of CV death will be:

- A) Unbiased.
- B) Biased downward (underestimated).
- C) Biased upward (overstated).
- D) Unrelated to the competing-risks issue.

> **Answer**
>
> **C.** KM treating competing death as censoring overstates the cumulative incidence because it implicitly assumes patients who died of other causes might have later died of CV causes. Use the cumulative incidence function or Fine-Gray regression.

### 13.17.5. Question 5

Your dataset has a covariate that changes during follow-up. The correct way to fit a Cox model with this covariate is:

- A) Use only the baseline value.
- B) Use only the final value.
- C) Use the time-average value.
- D) Expand the data to the counting-process format with one row per interval of constant covariate value and fit `coxph(Surv(start, stop, event) ~ ...)`.

> **Answer**
>
> **D.** Counting-process format is the correct representation. Options A–C all introduce bias.

## 13.18. Prerequisites answers

1. Right censoring occurs when a participant's event time is known to exceed some observed time $C$ (for example, they are still alive at the end of the study, or they withdrew at time $C$). The event *did* occur for some participants, and for those the event time is observed; for censored participants, the event time is partially observed (bounded below by $C$). This is different from a missing outcome because we retain information (the event did not occur before $C$); a missing outcome provides no such information and must be handled as missing data (see the Missing Data chapter of the companion *Practicum* volume).

2. The Kaplan-Meier estimator estimates the survivor function $S(t) = P(T > t)$, where $T$ is the event time. It multiplies the conditional probabilities of surviving each observed event time, where the conditional probability at time $t_j$ is $(n_j - d_j)/n_j$ with $n_j$ the risk set and $d_j$ the number of events. Censored participants contribute to the risk set until the moment of their censoring and are dropped afterward. The estimator is nonparametric and unbiased under independent censoring.
3. The proportional-hazards assumption is that the hazard ratio $\exp(x'\beta)$ is *constant over time*. Equivalently, the log hazard functions of any two groups defined by the covariates are parallel. You check it in R with `survival::cox.zph(fit)`, which tests whether the scaled Schoenfeld residuals have a non-zero slope against time, and graphically with `plot(cox.zph(fit))`. A significant global test or a visibly trending residual indicates violation; common remedies include stratification on the offending covariate or modelling time-varying coefficients.

# 14. Bayesian Computation

## 14.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 14.20.

1. State Bayes' theorem for posterior inference on a parameter $\theta$ given data $y$, identifying the prior, likelihood, posterior, and normalising constant.
2. What is the difference between a Metropolis-Hastings step and a Gibbs step? Under what conditions is Gibbs sampling possible?
3. After running an MCMC chain, what diagnostics would you use to decide whether the chain has converged, and what does the Gelman-Rubin $\hat{R}$ statistic measure?

## 14.2. Learning objectives

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

- State Bayes' theorem and apply it to a simple conjugate example by hand (beta-binomial, normal-normal).
- Implement a Metropolis-Hastings sampler from scratch for a one-parameter problem and verify convergence.
- Fit a Bayesian GLM with `rstanarm::stan_glm()` and interpret its output: posterior means, credible intervals, posterior predictive checks.
- Fit a Bayesian hierarchical model with `brms::brm()` and recognise the equivalence to `lme4::lmer()` under weakly informative priors.
- Run `posterior::summarise_draws()` and read the ESS, $\hat{R}$, and MCSE columns.

- Diagnose a chain that has not converged: trace plots, divergences, $\hat{R} > 1.01$, low ESS, low E-BFMI.
- Conduct a prior sensitivity analysis when priors might matter.

## 14.3. Orientation

Bayesian inference answers: *given this data and what I believed before I saw it, what should I believe now?* Frequentist inference answers: *under repeated sampling, how often would my procedure produce the observed result if the null were true?* Neither is universally correct; both are useful, and a biostatistician encounters both.

The computational tools to fit Bayesian models became accessible to non-specialists with the Stan ecosystem (2012-) and the tidy-Bayes R packages: `rstanarm`, `brms`, `bayestestR`, `posterior`. For most common models — linear regression, GLMs, mixed-effects, survival, a Bayesian fit is now a one-liner that takes a minute or two to run and produces output as easy to read as `summary(lm)`.

This chapter teaches computation, not philosophy. For the philosophy, Gelman et al.'s *Bayesian Data Analysis* and McElreath's *Statistical Rethinking* are the canonical references. For the computation, Stan is the lingua franca and `rstanarm`/`brms` are the R wrappers that turn most common models into one-liners.

A note on Stan backends. `rstan` is the original R interface and the default for `rstanarm` and (for many years) `brms`. `cmdstanr` is the contemporary alternative: it talks directly to the Stan command-line `cmdstan` and exposes new Stan features faster than `rstan` does. Switch to `cmdstanr` (via `brms::brm(..., backend = "cmdstanr")`) when you need a recent Stan feature or when shorter build times matter. For this chapter's coverage, the default `rstan` backend is fine.

## 14.4. The statistician's contribution

Bayesian computation is well-supported by software. The judgements that matter, which prior, which model, whether the chain converged, what the

posterior actually says, are not.

**Priors are not nuisance parameters.** A weakly informative default prior (`rstanarm`'s default for regression coefficients is `normal(0, 2.5)` on standardised scale) is fine for many applications and gives results numerically similar to a maximum-likelihood fit. But priors are not neutral: they encode what you believed before the data. For small samples, or for parameters with little information in the data, the prior dominates the posterior. The statistician's job is to choose priors deliberately, usually weakly informative, occasionally genuinely informative, and to do a sensitivity analysis when priors might matter.

**Convergence is a diagnostic, not a default.** $\hat{R}$ near 1.00 and adequate ESS are necessary, not sufficient. Trace plots, divergences (HMC), E-BFMI, and posterior predictive checks together build the case that the chain has converged to the right distribution. Any one of these failing is a problem; ignoring them and reporting the posterior mean as the answer is irresponsible.

**Credible intervals are not confidence intervals.** A 95% credible interval is a probability statement about the parameter: 'given the data and prior, the parameter has 0.95 probability of being in this range'. A 95% confidence interval is a probability statement about the procedure: 'in repeated sampling, this method captures the parameter 95% of the time'. They have different interpretations and sometimes different numerical values. Reporting a credible interval as a 'CI' without qualification is a common error.

**Posterior predictive checks are not optional.** A model that fits its training data perfectly may still generate data that look nothing like the observed data on substantively important features (zero counts, extreme values, autocorrelation). `pp_check(fit)` overlays observed data on simulated draws from the posterior predictive distribution; failures here are model- misspecification flags that no diagnostic of the parameters alone will catch.

These judgements are what distinguishes Bayesian analysis from running `stan_glm()` and reporting the result.

## 14.5. Bayes' theorem

For a parameter $\theta$ and data $y$:

$$p(\theta \mid y) = \frac{p(y \mid \theta)\, p(\theta)}{p(y)} \quad \propto \quad p(y \mid \theta)\, p(\theta).$$

The pieces:

- $p(\theta)$: the **prior**, what you believe about $\theta$ before seeing the data.
- $p(y \mid \theta)$: the **likelihood**, the probability of the observed data given the parameter.
- $p(\theta \mid y)$: the **posterior**, what you believe about $\theta$ after seeing the data.
- $p(y) = \int p(y \mid \theta) p(\theta)\, d\theta$: the **normalising constant** (marginal likelihood). MCMC samples from the unnormalised posterior, so this integral is rarely computed in practice.

The proportionality is the workable form. In closed form, when prior and likelihood are conjugate, the posterior has a known distributional form. In general, posterior sampling is the route.

### 14.5.1. A conjugate example: beta-binomial

For $y \sim \text{Binomial}(n, \theta)$ with prior $\theta \sim \text{Beta}(\alpha, \beta)$:

$$\theta \mid y \sim \text{Beta}(\alpha + y, \beta + n - y).$$

The posterior mean: $(\alpha + y)/(\alpha + \beta + n)$. With a uniform prior $\text{Beta}(1, 1)$ and data $y = 60$ successes out of $n = 100$, the posterior is $\text{Beta}(61, 41)$ with mean $61/102 \approx 0.598$, just slightly shrunk from the MLE 0.6 by the prior.

```r
n <- 100
y <- 60
alpha <- beta <- 1                  # uniform prior
post <- function(theta) dbeta(theta, alpha + y, beta + n - y)
curve(post, from = 0, to = 1)
abline(v = 0.6, col = "red")        # MLE
qbeta(c(0.025, 0.975), alpha + y, beta + n - y)
#> [1] 0.501 0.689
```

The 95% credible interval for $\theta$ comes directly from the beta quantile function. Compare to the Wald 95% CI for the same data: $0.6 \pm 1.96\sqrt{0.6 \cdot 0.4/100} \approx (0.504, 0.696)$, nearly identical. With informative priors or smaller samples, the agreement weakens.

## 14.6. A Metropolis-Hastings sampler from scratch

For a one-dimensional posterior $\pi(\theta) \propto p(y \mid \theta)p(\theta)$:

```r
mh_sampler <- function(log_posterior, theta0, n_iter = 10000,
                       proposal_sd = 1) {
  theta <- numeric(n_iter)
  theta[1] <- theta0
  accept <- 0
  for (i in 2:n_iter) {
    theta_prop <- rnorm(1, theta[i - 1], proposal_sd)
    log_a <- log_posterior(theta_prop) - log_posterior(theta[i - 1])
    if (log(runif(1)) < log_a) {
      theta[i] <- theta_prop
      accept <- accept + 1
    } else {
      theta[i] <- theta[i - 1]
    }
  }
  list(samples = theta, accept_rate = accept / (n_iter - 1))
}
```

```r
# example: beta-binomial posterior
log_post <- function(theta) {
  if (theta <= 0 || theta >= 1) return(-Inf)
  dbinom(60, 100, theta, log = TRUE) + dbeta(theta, 1, 1, log = TRUE)
}

set.seed(1)
res <- mh_sampler(log_post, theta0 = 0.5, n_iter = 10000,
                  proposal_sd = 0.1)
res$accept_rate
#> [1] 0.31

# discard burn-in, summarise
samples <- res$samples[1001:10000]
mean(samples)
quantile(samples, c(0.025, 0.975))
```

Three things matter for getting MH right:

1. **Proposal distribution.** Symmetric (e.g., normal) is the simplest case; the acceptance probability becomes $\min(1, \pi(\theta_{\text{prop}})/\pi(\theta_{\text{current}}))$. Asymmetric proposals require the Metropolis-*Hastings* correction.
2. **Tuning the proposal scale.** Too small → high acceptance, slow exploration. Too large → low acceptance, lots of rejections. The classic target is acceptance around 0.234 in high dimensions, ~0.4–0.5 for one-dimensional problems.
3. **Burn-in and thinning.** The first samples depend on the starting value; discard them. Thinning (keeping every $k$th sample) is rarely useful with modern diagnostics, ESS captures the relevant autocorrelation.

For high-dimensional posteriors, raw MH scales poorly: acceptance rates collapse, and chains move slowly through the posterior. Hamiltonian Monte Carlo (the engine in Stan) addresses this with momentum variables and gradient information, achieving much better mixing in high dimensions.

## 14.7. Hamiltonian Monte Carlo, in one paragraph

HMC augments the parameter space with momentum variables and uses the gradient of the log-posterior to propose correlated steps that follow approximate Hamiltonian dynamics. The result: proposals are far from the current state but still in regions of high posterior density, so acceptance is high and effective sample size per iteration is much higher than MH. Stan implements an adaptive HMC variant called the No-U-Turn Sampler (NUTS) that tunes the trajectory length automatically. The cost per iteration is high (each requires multiple gradient evaluations), but the cost per effective sample is far lower than MH for problems with more than a handful of parameters. This is why `rstanarm` and `brms` work as well as they do on realistic models: NUTS can fit a hierarchical model in seconds when MH would take hours.

You will not implement HMC by hand for this course; understanding why it dominates MH is enough.

## 14.8. Fitting a Bayesian GLM with `rstanarm`

`rstanarm` mirrors base R's modelling functions: `stan_lm` for linear regression, `stan_glm` for GLMs, `stan_lmer` and `stan_glmer` for mixed-effects, `stan_polr` for ordinal regression, `stan_betareg` for beta regression, and several others. Default priors are weakly informative on the standardised scale, which works for most regression problems out of the box.

```
library(rstanarm)
data(penguins, package = "palmerpenguins")
penguins <- na.omit(penguins)

fit <- stan_glm(
  body_mass_g ~ flipper_length_mm + species,
  data = penguins,
  family = gaussian(),
  prior = normal(0, 100, autoscale = TRUE),
  prior_intercept = normal(4000, 500),
  chains = 4,
```

```
  cores = 4,
  seed = 47,
  refresh = 0       # silence sampling progress
)

summary(fit)
broom.mixed::tidy(fit, conf.int = TRUE)
```

`prior` controls the prior on slope coefficients; `prior_intercept` the prior on the intercept; `prior_aux` the prior on the residual standard deviation. The `autoscale = TRUE` flag rescales the prior to the standard deviation of each predictor, making default priors sensible across covariate scales.

The output:

- **Median** posterior estimate per coefficient (the default; mean is also available).
- **MAD_SD**: the median absolute deviation of the posterior, scaled to be comparable to a frequentist SE.
- **Sample sizes**: ESS for each parameter; should be several hundred minimum for reliable quantile-based summaries.
- $\hat{R}$: should be $< 1.01$ for every parameter.

Comparison to `lm()`:

```
fit_lm <- lm(body_mass_g ~ flipper_length_mm + species, data = penguins
cbind(
  bayes = coef(fit),
  freq  = coef(fit_lm)
)
```

Under weakly informative priors, the posterior medians should match the OLS coefficients to several decimal places, and the MAD_SDs should match the SEs. The match becomes worse as priors become more informative, sample sizes shrink, or the model becomes more hierarchical.

Check your understanding: Bayes vs. frequentist agreement

**Question.** Under what conditions should you expect a Bayesian posterior mean to differ substantially from the maximum likelihood estimate?

**Answer.**

Three conditions, alone or in combination, drive the gap: (1) informative priors that pull the posterior away from the data; (2) small sample sizes, where prior weight relative to likelihood weight is large; (3) hierarchical structure, where the Bayesian model imposes shrinkage of group-level parameters toward the population mean while a frequentist mixed-model does the same but typically less strongly. With a weakly informative or uniform prior, a moderate sample size, and a non-hierarchical model, the two approaches will agree to about three significant digits, and reporting the Bayesian fit produces no substantive change to the conclusions.

## 14.9. Fitting a Bayesian hierarchical model with brms

`brms` translates a `lme4`-style formula into a Stan model and fits it. The interface is more flexible than `rstanarm` (custom likelihoods, censored outcomes, splines, mixture models) at the cost of slower compile times.

```
library(brms)
data(sleepstudy, package = "lme4")

fit <- brm(
  Reaction ~ Days + (Days | Subject),
  data = sleepstudy,
  family = gaussian(),
  chains = 4,
  cores = 4,
  seed = 47,
  refresh = 0
)
```

```
summary(fit)
plot(fit)
posterior::summarise_draws(fit, default_summary_measures())
```

Compare to `lme4::lmer(Reaction ~ Days + (Days | Subject), data = sleepstudy)`: the fixed-effect estimates should agree closely (weakly informative priors); the random-effect variance components should agree closely; the BLUPs from `lmer` and the posterior means of the subject-specific deviations from `brm` should also agree, with `brm` producing slightly more shrinkage when the data are sparse for a subject.

The Bayesian view of random effects is straightforward: they are parameters with a hierarchical prior. The prior's variance (the random-effects SD) is itself estimated. There is no philosophical distinction between fixed and random effects in a Bayesian model; both are parameters, and the hierarchical prior on the latter is what produces the shrinkage.

## 14.10. `cmdstanr` for advanced Stan use

`cmdstanr` is the recommended interface to Stan for advanced users. It talks directly to the cmdstan executable, supports the latest Stan features fastest, and is the basis on which `brms` builds. Use `cmdstanr::cmdstan_install()` once; subsequent `brms::brm(... backend = "cmdstanr")` calls will use it. For most users, `brms` with the default `rstan` backend is fine; switch to `cmdstanr` when you need a Stan feature that `rstan` does not yet expose, or when build-time matters.

## 14.11. Posterior summaries

Once the chain has converged, the posterior is a sample. Use it.

```
library(posterior)

draws <- as_draws_df(fit)
summarise_draws(draws,
```

```
                mean,
                ~ quantile(.x, probs = c(0.025, 0.5, 0.975)),
                rhat,
                ess_bulk,
                ess_tail)
```

The columns:

- `mean`, `2.5%`, `50%`, `97.5%`: posterior summaries. The 95% credible interval is the 2.5th to 97.5th percentile of the posterior; the median is often more robust than the mean for skewed posteriors.
- `rhat`: $\hat{R}$. Want $< 1.01$.
- `ess_bulk`, `ess_tail`: bulk and tail effective sample sizes. Want at least a few hundred each for reliable posterior quantile summaries.

`tidybayes` provides similar functionality with a more ggplot-friendly API. `bayesplot` produces standardised diagnostic plots: trace plots, autocorrelation, posterior interval plots.

## 14.12. Posterior predictive checks

`pp_check(fit)` overlays the observed data on draws from the posterior predictive distribution. Misfit is visible:

```
pp_check(fit, ndraws = 100)
```

The default plot shows the observed outcome density vs. posterior predictive draws. A model whose posterior predictive draws differ systematically from the observed data, different mode, different variance, different range, different proportion of zeros, is misspecified on something the data are sensitive to.

PP checks are essential and routinely overlooked. A posterior with $\hat{R} = 1.000$ and tight credible intervals can still come from a model that does not describe the data; PP checks are how you find out.

## 14.13. Convergence diagnostics

Required:

- **Trace plots.** Visual check for stationarity (no drift) and mixing (chains overlap, look like white noise).
- $\hat{R}$**.** Want $< 1.01$ for every parameter. $\hat{R}$ compares within-chain variance to between-chain variance; values near 1 mean the chains have all converged to the same distribution.
- **Effective sample size.** Want at least 400 in `ess_bulk` and `ess_tail` per parameter. Lower ESS means the chain is autocorrelated; quantile-based summaries are noisy.

For HMC (Stan-based fits, including `rstanarm` and `brms`):

- **Divergences.** Should be zero (or nearly). A divergent transition is HMC failing to integrate the dynamics faithfully. Even one divergence flags a problem; many divergences typically mean the model is not as identifiable as you thought, or the parameterisation is bad. Standard fix: non-centred parameterisation (especially for hierarchical scale parameters) and `adapt_delta = 0.99` (the default is 0.8).
- **E-BFMI.** Energy-based fraction of missing information. Should be $> 0.3$. Low E-BFMI suggests the posterior has funnel-like shape that HMC struggles with; usually a reparameterisation problem.

```
library(bayesplot)
mcmc_trace(fit)
mcmc_acf(fit, regex_pars = "b_")
```

Check your understanding: $\hat{R}$ and ESS

**Question.** Your model has $\hat{R} = 1.005$ for every parameter (good) but `ess_bulk = 80` for one of the random-effects variance parameters. Should you trust the posterior summary for that parameter?
**Answer.**
No. ESS = 80 means the effective number of independent samples for that parameter is only 80, so quantile-based summaries (the credible interval bounds especially) have substantial Monte Carlo error. The

posterior median is probably accurate; the 2.5th and 97.5th percentiles might shift by 5–10% on a re-run. Standard fix: run more iterations (`iter = 4000` instead of the default 2000), or improve mixing via reparameterisation. The threshold of 400 ESS for reliable quantile summaries is the published guideline (Vehtari et al. 2021); below that, treat the credible-interval bounds as approximate.

## 14.14. Priors

`rstanarm` and `brms` use weakly informative defaults. For most regression problems with reasonable sample sizes, these are fine. When are they not?

- **Very small samples.** With $n = 20$, the prior may shape the posterior more than the data do. Be deliberate.
- **Strong prior knowledge.** When external evidence genuinely informs the parameter (e.g., a meta-analysis has pinned the treatment effect to a narrow range), using that information via the prior is appropriate.
- **Identifiability problems.** A weakly identified parameter (e.g., the scale of a latent variable in a factor model) needs a prior that the data cannot fully overrule.

For sensitivity analysis, refit with two alternative priors and report whether conclusions change. If they do, disclose; if they do not, the analysis is robust to the prior choice.

```
fit_default <- stan_glm(y ~ x, data = d)
fit_skeptical <- stan_glm(y ~ x, data = d, prior = normal(0, 0.5))
fit_optimistic <- stan_glm(y ~ x, data = d, prior = normal(2, 1))

# compare posterior means and CIs across priors
```

## 14.15. Collaborating with an LLM on Bayesian computation

LLMs handle the syntax of `stan_glm` and `brm` reasonably well; they handle the diagnostics and prior choices much less reliably.

**Prompt 1: translating a frequentist fit.** Paste the output of `glm(...)` and ask: 'fit the equivalent Bayesian model with `rstanarm`, using weakly informative priors. Compare the coefficients to the frequentist estimates and explain any differences.'

*What to watch for.* The LLM should produce a fit with similar coefficients (under weakly informative priors, they will agree to several decimals on most data). If the LLM produces wildly different estimates, suspect that the priors are too informative or the chain has not converged.

*Verification.* Run both fits. Print the coefficients side by side. Check $\hat{R}$ and ESS for the Bayesian fit. If either is bad, the comparison is meaningless.

**Prompt 2: diagnosing a non-converged chain.** Paste the warning ('There were N divergent transitions…') and the relevant trace plots, and ask: 'what is happening, and what should I try?'

*What to watch for.* Standard remedies for divergences: non-centred parameterisation for hierarchical scales, `adapt_delta = 0.99`, more informative priors on problematic parameters. The LLM should know these. If it suggests 'just ignore the warning', push back.

*Verification.* Implement the suggested fix and verify the divergences disappear. If they do not, escalate (a reparameterisation that LLMs do not always identify is needed).

**Prompt 3: posterior predictive check interpretation.** Paste the `pp_check` plot description and ask: 'what does this say about the model?'

*What to watch for.* LLMs can usually identify obvious mismatches (mode in wrong place, missing tail behaviour, zero-inflation). They are weaker on subtle mismatches that require subject-matter knowledge. Trust the LLM's pattern- matching but verify the substantive judgement against your understanding of the data.

## 14.16. Principle in use

Three habits define defensible Bayesian computation:

1. **Always check convergence.** $\hat{R}$, ESS, divergences, E-BFMI. A non-converged chain is not a fit.
2. **Always do a posterior predictive check.** A converged chain on a misspecified model produces a precise wrong answer. PP checks are how you find out.
3. **Be deliberate about priors.** Defaults are fine for many applications; they are not fine for all. Sensitivity analysis disclosing prior dependence is the polite minimum.

## 14.17. Exercises

1. Using `palmerpenguins::penguins`, fit a Bayesian linear regression of `body_mass_g` on `flipper_length_mm` with `rstanarm::stan_glm()` under weakly informative priors. Extract the 95% credible interval for the slope. Compare it to the 95% CI from `lm()`.
2. Implement a Metropolis-Hastings sampler from scratch for the posterior of a binomial proportion with a `beta(1, 1)` prior. Run 10,000 iterations and compare the sample mean and SD to the analytic posterior mean and SD. What proposal SD gives an acceptance rate near 0.4?
3. Fit the `sleepstudy` random-slope model with both `lme4::lmer()` and `brms::brm()`. Compare the fixed effects, the variance components, and the subject- specific intercepts and slopes (BLUPs from lmer vs. posterior means from brm). Under what conditions would you expect them to disagree?
4. Run a `stan_glm()` fit with three different priors on the slope: `normal(0, 1000)` (weak), `normal(0, 1)` (moderate), and `normal(2, 0.1)` (strongly informative). Document how the posterior shifts.
5. Fit a model that diverges (a hierarchical model with a tight, identical prior on every random-effect group often does the trick). Diagnose

using `bayesplot::mcmc_pairs()` and apply a non-centred parameterisation. Verify the divergences vanish.

## 14.18. Further reading

- Gelman et al. (2013), *Bayesian Data Analysis*, 3rd ed., Chapman and Hall/CRC, the canonical reference.
- McElreath (2020), *Statistical Rethinking*, 2nd ed. , accessible introduction paired with `brms` code.
- Stan Development Team documentation at `mc-stan.org/docs`, the ecosystem reference.
- Vehtari et al. (2021), 'Rank-normalisation, folding, and localisation: an improved $\hat{R}$ for assessing convergence of MCMC', *Bayesian Analysis*, the modern guideline for $\hat{R}$ and ESS thresholds.
- Bürkner (2017), 'brms: An R package for Bayesian multilevel models using Stan', *JSS*, the brms paper.

## 14.19. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 14.19.1. Question 1

Bayes' theorem expresses the posterior $p(\theta \mid y)$ as proportional to:

- A) The likelihood $p(y \mid \theta)$.
- B) The prior $p(\theta)$.
- C) The product $p(y \mid \theta)\, p(\theta)$.
- D) The marginal likelihood $p(y)$.

Answer

**C.** The proportionality avoids computing the normalising constant $p(y)$, which is the high-dimensional integral that motivates MCMC in the first place.

### 14.19.2. Question 2

A 95% credible interval for a parameter $\theta$ is:

- A) The set of values for which a frequentist test would not reject the null at $\alpha = 0.05$.
- B) The set of values such that, given the prior and data, the posterior probability that $\theta$ is in the set is 0.95.
- C) The set of $\theta$ values containing the MLE.
- D) Equivalent in interpretation to a frequentist confidence interval.

Answer

**B.** Credible intervals are probability statements about the parameter; confidence intervals are probability statements about the procedure. Numerically, they often agree under weakly informative priors with moderate samples.

### 14.19.3. Question 3

In the Metropolis-Hastings algorithm, the acceptance probability for a symmetric proposal is:

- A) $\min(1, p(\theta_{\text{prop}})/p(\theta_{\text{current}}))$, where $p$ is the posterior density (or any function proportional to it).
- B) $\min(1, p(\theta_{\text{current}})/p(\theta_{\text{prop}}))$.
- C) Always 1.
- D) The acceptance rate of the chain.

Answer

**A.** A proposal that is more likely than the current value is always accepted; one that is less likely is accepted with probability equal to the ratio.

### 14.19.4. Question 4

You run an HMC chain in Stan and see 12 divergent transitions reported. The most appropriate response is:

- A) Ignore them; small numbers of divergences are cosmetic.
- B) Increase `adapt_delta` (e.g., to 0.99), consider a non-centred parameterisation, and possibly tighten priors.
- C) Switch to Metropolis-Hastings.
- D) Take the posterior at face value if $\hat{R}$ is near 1.

Answer

**B.** Even a few divergences flag a region of the posterior the sampler cannot traverse, biasing the posterior toward easier-to-sample regions. The standard remedies are reparameterisation and `adapt_delta`.

### 14.19.5. Question 5

Posterior predictive checks (`pp_check(fit)`) primarily detect:

- A) Convergence failures of the chain.
- B) Insufficient ESS.
- C) Model misspecification: features of the data the model fails to reproduce in posterior simulations.
- D) Choice of prior.

Answer

**C.** PP checks compare observed data to posterior predictive draws and are essential for detecting misspecification that no diagnostic of the parameters alone catches.

## 14.20. Prerequisites answers

1. Bayes' theorem: $p(\theta \mid y) = p(y \mid \theta)p(\theta)/p(y)$, where $p(\theta)$ is the **prior**, $p(y \mid \theta)$ is the **likelihood**, $p(\theta \mid y)$ is the **posterior**, and $p(y) = \int p(y \mid \theta)p(\theta)\, d\theta$ is the **normalising constant** (marginal likelihood). In practice, MCMC samples from the unnormalised posterior $p(y \mid \theta)p(\theta)$, so the normalising constant need not be computed explicitly.
2. A Metropolis-Hastings step proposes a new value from a proposal distribution and accepts it with a probability based on the ratio of posterior densities. It requires only the ability to evaluate the posterior up to a constant. A Gibbs step samples each parameter (or block of parameters) from its full conditional distribution. Gibbs is possible only when the full conditionals are available in closed form, typically via conjugacy. MH applies more generally; Gibbs is more efficient when available.
3. Trace plots (visual check for stationarity and mixing), the Gelman-Rubin statistic $\hat{R}$ (want $< 1.01$), effective sample size (ESS; want more than several hundred per parameter for reliable quantile- based summaries), and, for HMC, the divergence count (ideally zero) and E-BFMI (ideally $> 0.3$). $\hat{R}$ compares within-chain and between-chain variance after discarding burn-in; values near 1 indicate that multiple chains have converged to the same target distribution, values substantially above 1 indicate that the chains are still exploring different regions.

# Part V.

# Visualization and Communication

# 15. Visualization and the Grammar of Graphics

## 15.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 15.20.

1. Name the seven main components of a statistical plot according to the Grammar of Graphics.
2. In `ggplot2`, what does an aesthetic mapping (`aes()`) specify?
3. What is the primary advantage of the Grammar of Graphics approach compared with traditional per-chart-type plotting functions?

## 15.2. Learning objectives

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

- State the components of the grammar of graphics: data, aesthetic mapping, geom, stat, scale, coord, facet.
- Build a plot with `ggplot2` by composing layers.
- Choose an appropriate geom for a given combination of variable types (continuous-continuous, continuous-discrete, discrete-discrete, count, time, etc.).
- Apply scales for colour, fill, size, and shape, including colour-blind-safe palettes.
- Use facets to display the same plot conditioned on a third variable.
- Recognise and avoid common visualisation errors: truncated axes, dual y-axes, double-encoding, default rainbow colour scales, pie charts.

- Distinguish 'good plots' (one message, truthful, clear encoding) from 'bad plots' that decorate rather than communicate.

## 15.3. Orientation

A good plot conveys a single message quickly and truthfully. A bad plot does neither: it either decorates without communicating or communicates a wrong impression of the data.

The Grammar of Graphics, codified by Leland Wilkinson and implemented for R by Hadley Wickham as `ggplot2`, makes the *structure* of a plot explicit. Every plot is a composition of seven kinds of element: data, aesthetic mappings, geoms, stats, scales, a coordinate system, and facets. Once you think in these pieces, the universe of statistical graphics becomes a small set of swappable parts rather than a zoo of named chart types.

This chapter teaches the grammar conceptually and applies it through `ggplot2`. The next chapter goes deeper on advanced ggplot techniques (themes, extensions, interactivity).

## 15.4. The statistician's contribution

Software can produce a plot from any data; software cannot choose what plot to make.

**Pick the right encoding.** Continuous values map well to position (x, y, length). Categorical values map well to colour, shape, or facets. Mapping a continuous variable to shape (10 different shapes for 10 levels) creates an unreadable plot. Mapping a categorical variable to a sequential colour scale implies an ordering that may not exist. The grammar makes these mistakes structurally visible because each aesthetic has a natural variable type.

**One plot, one message.** A figure that crams in three relationships is harder to read than three figures each showing one. The temptation to combine is strong; the clarity gained by separating is usually worth more than the space saved.

**Truthful axes.** Truncating a y-axis to make a small difference look large is dishonest. Logarithmic axes without a clear visual cue mislead readers who do not notice. Axes should encode what they appear to encode.

**Colour matters.** Roughly 8% of men and 0.5% of women have some form of colour vision deficiency. Default `ggplot2` palettes (and most journal defaults) are not colour-blind- safe. The `viridis`, `RColorBrewer`, and `ggthemes` palettes include accessible options. Picking one is a thirty-second fix that keeps the plot legible to a substantially larger audience.

These judgements are what separate plots that communicate from plots that fill space. They are not automatable; the grammar makes them easier to think about by structuring the plot into separable decisions.

## 15.5. The grammar of graphics: seven components

A plot is the composition of:

1. **Data.** A data frame (or tibble). The plot is a view of these data.
2. **Aesthetic mappings.** Which columns of the data map to which visual properties (x-position, y-position, colour, shape, size, fill, alpha).
3. **Geoms (geometric objects).** What is drawn at each data point: a point, a line, a bar, a polygon, a tile.
4. **Stats (statistical transformations).** Whether the data are drawn as-is or first summarised: smoothed, binned into histograms, summarised by mean and SE.
5. **Scales.** How aesthetic values are translated to pixels: which range of x-values maps to which pixels; which colours encode which categories.
6. **Coordinate system.** Cartesian (default), polar, geographic, transformed (log, sqrt).
7. **Facets.** Whether the plot is split into a grid of subplots conditioned on a categorical variable.

Every `ggplot()` call is some combination of these. Most calls touch only a few; the others have sensible defaults.

## 15.6. A minimum-viable example

```
library(ggplot2)
library(palmerpenguins)

p <- penguins |>
  na.omit() |>
  ggplot(aes(x = flipper_length_mm,
             y = body_mass_g,
             colour = species)) +
    geom_point(alpha = 0.7) +
    scale_colour_brewer(palette = "Dark2") +
    labs(x = "Flipper length (mm)",
         y = "Body mass (g)",
         colour = "Species") +
    theme_minimal()

p
```

What each piece does:

- `penguins |> na.omit()` is the **data**.
- `aes(x = ..., y = ..., colour = ...)` is the **aesthetic mapping**: three data columns mapped to three visual properties.
- `geom_point(alpha = 0.7)` is the **geom**: each row gets a semi-transparent point.
- `scale_colour_brewer(...)` is the **scale**: how species values map to colours (Dark2 is colour-blind-safe).
- `labs(...)` and `theme_minimal()` provide labels and a visual style.
- No explicit **stat** because `geom_point` defaults to `stat_identity` (no transformation).
- No **coord** because `coord_cartesian` is the default.
- No **facet** because we are showing all data in one panel.

The plot is built by adding layers with `+`. Each layer can override the inherited mapping, change the geom, add a stat, or modify the appearance.

## 15.7. Geoms by variable types

The choice of geom is largely determined by the types of the x and y variables.

| x type | y type | Common geoms |
|---|---|---|
| continuous | continuous | `geom_point`, `geom_smooth`, `geom_density_2d` |
| continuous | discrete | `geom_boxplot` (with `coord_flip`), `geom_violin` |
| discrete | continuous | `geom_boxplot`, `geom_violin`, `geom_point` (jittered), `geom_col` |
| discrete | discrete | `geom_count`, `geom_tile` (heatmap) |
| time | continuous | `geom_line`, `geom_step`, `geom_area` |
| (1 var) | (1 var) | `geom_histogram`, `geom_density`, `geom_bar` |

For two-variable scatterplots with thousands of points, overlap obscures the pattern. Solutions:

```
geom_point(alpha = 0.05)                # transparency
geom_hex(bins = 50)                     # hexagonal binning
geom_density_2d_filled()                 # 2D density contours
```

For violin and boxplot combinations:

```
ggplot(penguins, aes(x = species, y = body_mass_g)) +
  geom_violin(fill = "lightgrey") +
  geom_boxplot(width = 0.1) +
  geom_jitter(width = 0.1, alpha = 0.4)
```

Layer geoms freely; the order matters (later layers paint over earlier ones).

Check your understanding: choosing a geom

**Question.** You have a dataset with 5,000 observations of two continuous variables, and you want to show the relationship. The default `geom_point()` produces a black blob in the middle of the plot. What are your options?

**Answer.**

The blob is overplotting. Standard remedies, roughly in order: (1) `alpha = 0.05` makes individual points semi-transparent, so density translates to visual darkness; (2) `geom_hex()` bins observations into hexagonal cells coloured by count; (3) `geom_density_2d()` or `geom_density_2d_filled()` shows the joint density as contours rather than points. For pure scatterplots with very large $n$, `geom_hex` is usually clearest. For showing *both* the density and any individual outliers, combine `geom_hex` with `geom_point` for outliers only.

## 15.8. Statistical transformations

Geoms can summarise the data before drawing. The summary is controlled by the `stat` argument or by the geom itself.

```
# raw points (stat_identity, the default)
ggplot(penguins, aes(flipper_length_mm, body_mass_g)) +
  geom_point()

# linear smooth with 95% CI
ggplot(penguins, aes(flipper_length_mm, body_mass_g)) +
  geom_point() +
  geom_smooth(method = "lm", formula = y ~ x)

# loess smooth (default for n < 1000)
ggplot(penguins, aes(flipper_length_mm, body_mass_g)) +
  geom_point() +
  geom_smooth()

# group means with SE
```

```
ggplot(penguins, aes(species, body_mass_g)) +
  stat_summary(fun.data = mean_se)

# histogram (stat_bin)
ggplot(penguins, aes(body_mass_g)) +
  geom_histogram(bins = 30)
```

`geom_smooth(method = "lm")` fits a linear model per group (if the aesthetic includes `colour` or `group`) and plots the fitted line with a confidence band. For regression visualisation, this single line of code does what would take many in base R.

## 15.9. Scales

Scales control how aesthetic values map to visual properties. The defaults are reasonable but rarely optimal.

**Continuous axes:**

```
scale_x_continuous(breaks = seq(0, 100, 25),
                   labels = scales::number_format(suffix = "%"),
                   limits = c(0, 100))

scale_y_log10()                 # log-transformed y axis
scale_x_sqrt()                  # square-root transformed
```

**Discrete axes:**

```
scale_x_discrete(limits = c("placebo", "low", "high"))   # custom or
```

**Colour:**

```
scale_colour_brewer(palette = "Dark2")    # colour-blind safe, quali
scale_colour_viridis_d()                  # colour-blind safe, ordere
scale_colour_manual(values = c("a" = "red", "b" = "blue"))
scale_colour_gradient(low = "white", high = "darkblue")  # continuou
```

`viridis` and ColorBrewer's qualitative palettes (Dark2, Set1, Set2) are the standard colour-blind-safe defaults. Avoid the rainbow scale (jet, `rainbow()`) for continuous data: it is not perceptually uniform and is unreadable for many readers.

For discrete fills with no natural order, qualitative. For ordered discrete or continuous, viridis or sequential ColorBrewer (Blues, Reds, etc.). For diverging data (positive vs. negative), diverging palettes (RdBu, PuOr).

## 15.10. Faceting

Facets split the plot into a grid of small multiples by one or two variables:

```
# rows = species, columns = sex
ggplot(penguins, aes(flipper_length_mm, body_mass_g)) +
  geom_point() +
  facet_grid(species ~ sex)

# wrap into a single dimension
ggplot(penguins, aes(flipper_length_mm, body_mass_g)) +
  geom_point() +
  facet_wrap(~ species, ncol = 1)
```

Faceting is one of the strongest tools in the grammar: it lets you condition on a third (or fourth) variable without losing the underlying plot's design. Particularly useful for displaying group-specific patterns when groups are too many or too small for colour-coding to be readable.

## 15.11. Common mistakes

**Truncated axes.** Setting `ylim()` to exclude zero on a bar chart exaggerates differences. Setting it to exclude the data range entirely is misleading. The axis should encode the data faithfully.

**Dual y-axes.** Two y-axes on different scales nearly always mislead. Readers cannot tell which trace corresponds to which axis at a glance. Better: split into two facets.

**Pie charts beyond two categories.** Humans read angles poorly. A bar chart of the same data is universally clearer.

**Default colour for ordered categories.** `ggplot2`'s default `scale_colour_discrete` is *unordered*. If your categories are ordered (e.g., low/medium/high), use `scale_colour_viridis_d()` or a sequential ColorBrewer palette to encode the ordering visually.

**3D plots.** Almost always worse than 2D plus a colour or size encoding.

**Over-the-top legends.** A legend that explains 10 categories with 10 colours and 10 shapes triple-encodes the same information. Pick one.

Check your understanding: a misleading plot

**Question.** A bar chart shows a treatment group with 78% response and a control group with 75% response. The y-axis runs from 70% to 80%, making the treatment bar appear roughly four times taller than the control bar. What is the visual lie?

**Answer.**

The truncated y-axis. The actual difference is 3 percentage points; the visual difference suggests roughly fourfold. For bar charts, the y-axis should start at 0 (or, for negative values, span both directions symmetrically). For showing small differences honestly, either show them as a line on a true-to-scale y-axis, or report them as differences directly (a bar chart of the 3-point gap with its CI). Truncated y-axes are among the most common ways scientific papers visually mislead, and the grammar of graphics makes the issue visible: the scale is encoding pixels-per-unit, and a truncated scale encodes pixels-per-relative-difference, which is not what the reader expects.

## 15.12. Themes and labels

Themes control non-data appearance: backgrounds, grids, fonts. Built-in themes include `theme_grey()` (the default), `theme_minimal()`,

`theme_bw()`, `theme_classic()`. Pick one and use it consistently across a paper or report.

```
ggplot(penguins, aes(flipper_length_mm, body_mass_g)) +
  geom_point() +
  labs(title = "Body mass vs. flipper length",
       subtitle = "Palmer Archipelago penguins, 2007--2009",
       x = "Flipper length (mm)",
       y = "Body mass (g)",
       caption = "Source: palmerpenguins package") +
  theme_minimal()
```

Always label axes with units. 'Body mass (g)' tells the reader more than 'body_mass_g'. Captions citing data sources earn trust.

## 15.13. Saving plots

```
ggsave("flipper_mass.png", plot = p, width = 6, height = 4,
       dpi = 300, units = "in")

# vector format for publication
ggsave("flipper_mass.pdf", plot = p, width = 6, height = 4)
```

For papers, use vector formats (PDF, SVG) when possible; they remain crisp at any zoom. For web, PNG at moderate DPI is a good default. Avoid JPG for plots, the compression artefacts are visible on lines and text.

## 15.14. Collaborating with an LLM on visualisation

LLMs handle ggplot2 code well; they handle visualisation *judgement* less reliably.

**Prompt 1: drafting a plot.** Describe the data and the question. Ask: 'write a ggplot2 plot that addresses the question. Use a colour-blind-safe palette and label axes with units.'

*What to watch for.* The default LLM plot tends toward busy: too many aesthetics encoded, default ggplot theme, sometimes no axis labels. Push for clarity over decoration. Multiple iterations of 'simpler' tend to improve the result.

*Verification.* Render the plot and ask whether a reader who has never seen the data could state the message in one sentence. If not, simplify.

**Prompt 2: critiquing a plot.** Paste a `ggplot()` call and the rendered plot description, ask: 'critique this plot, what would be unclear or misleading to a reader?'

*What to watch for.* LLMs are reasonable at spotting truncated axes, dual y-axes, missing labels, and palette issues. They are weaker on subject-matter problems (wrong choice of summary, inappropriate scale for the units). Bring substantive judgement.

*Verification.* Apply the suggested fixes. Run the plot. Compare before and after.

**Prompt 3: replicating a published plot.** Describe a target figure (or paste the published plot if available) and ask the LLM to reproduce its design.

*What to watch for.* LLMs can reproduce the visual style of well-known plots (Tufte, Minard); they cannot read your data and decide whether the same design fits. Treat the LLM output as a starting point and adapt for your data.

*Verification.* Compare with the original. Are the encodings the same? The aspect ratio? The colours? Differences may be improvements; they may be regressions.

## 15.15. Principle in use

Three habits define defensible visualisation:

1. **One plot, one message.** Resist the urge to combine three relationships into one figure. Three figures with one message each are clearer.

2. **Truthful axes, accessible colours.** No truncated axes that exaggerate differences; no colour palettes that exclude colour-blind readers; no dual y-axes that mislead by their scale.
3. **Compositional thinking.** Build plots by combining grammar pieces, not by reaching for chart types. The right plot for an unusual question is rarely a built-in chart; it is usually a small variation on something familiar.

## 15.16. Version notes (ggplot2 3.5+)

A few legacy patterns no longer work in current `ggplot2`. `aes_string()` is soft-deprecated in `ggplot2` 3.5+; use the `.data` and `.env` pronouns from `rlang` for programmatic aesthetics, e.g., `aes(x = .data[[varname]])`. `qplot()` has been fully removed; use `ggplot()` directly. When older code or LLM output uses these constructs, translate to the current forms before relying on the result.

## 15.17. Exercises

1. Reproduce the penguin body-mass-vs-flipper-length scatter plot from `palmerpenguins`, coloured by species, faceted by island. Add a regression line and a 95% confidence band per species.
2. Take a bad plot from a recent newspaper article (or social media). Identify three things wrong with it. Rewrite it in `ggplot2` to fix them.
3. Use `ggplot2::aes()` to map the same variable to two different aesthetics (e.g., colour and shape). Argue for or against this as a design choice with reference to the encoding principles in this chapter.
4. Take a dataset with 100,000+ rows. Make a scatter plot that does not suffer from overplotting. Compare three approaches: alpha, hex binning, and 2D density contours. Which works best for the data?
5. Make a `facet_wrap` plot with too many panels (say, 30 for 30 patients in a longitudinal study). Identify what goes wrong, and propose two alternatives: `facet_wrap` with a different ordering, or a different plot entirely.

## 15.18. Further reading

- (Wickham, 2016), canonical `ggplot2` reference; the 3rd edition is online at https://ggplot2-book.org/.
- (Wilke, 2019), principles of effective visualisation, with every figure built in R.
- (Tufte, 2001), the foundational text on information design.
- (Healy, 2018), shorter, applied, with R code; a good complement to Wickham's reference.

## 15.19. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 15.19.1. Question 1

According to the Grammar of Graphics framework, which of the following is NOT one of the seven main components of a plot?

- A) Data
- B) Aesthetics
- C) Colors
- D) Layers

> **Answer**
>
> **C.** Colour is a specific aesthetic, not a component of the grammar.

### 15.19.2. Question 2

In `ggplot2`, what does the aesthetic mapping specify?

- A) The colour scheme to use for the plot
- B) How variables in the data are mapped to visual properties

- C) The size of the plot window
- D) The theme and style of the plot

> **Answer**
>
> **B.** `aes()` maps data columns to visual properties such as x, y, colour, shape, size.

### 15.19.3. Question 3

What is the main advantage of the Grammar of Graphics approach compared to traditional plotting approaches?

- A) It produces more colourful plots
- B) It renders plots faster
- C) It provides a systematic way to describe and create visualisations
- D) It requires less code to create simple plots

> **Answer**
>
> **C.** The grammar is compositional: new plots are assembled from a small set of reusable pieces.

### 15.19.4. Question 4

A bar chart shows treatment success rates of 75% (control) and 78% (treatment). The y-axis runs from 70% to 80%. The visual impression is misleading because:

- A) The colours are unreadable.
- B) The y-axis is truncated, exaggerating the difference.
- C) Bar charts are inherently misleading.
- D) The font size is too small.

Answer

**B.** For bar charts, the y-axis should start at 0 (or span symmetrically across zero for negative values). Truncation makes a 3-point difference look much larger.

### 15.19.5. Question 5

You are encoding species (3 categories) on a scatter plot. Which scale is most appropriate?

- A) `scale_colour_continuous()` with a viridis ramp
- B) `scale_colour_brewer(palette = "Dark2")`
- C) `scale_colour_gradient(low, high)`
- D) Default ggplot2 colours

Answer

**B.** Three unordered categories want a qualitative colour-blind-safe palette. Dark2 (or Set2 / viridis_d) fit the role; continuous scales imply an ordering that species do not have; the ggplot2 defaults are not colour-blind-safe.

## 15.20. Prerequisites answers

1. Data, aesthetic mappings, geometries (geoms), statistical transformations (stats), scales, coordinate system, and faceting. Note that *colours* is a specific aesthetic, not one of the seven components.
2. An aesthetic mapping specifies how a variable in the data is mapped to a visual property of the plot (x-position, y-position, colour, shape, size, etc.). The grammar enforces a strict separation between data (what) and visual encoding (how it appears).
3. The grammar provides a systematic, compositional way to describe and create visualisations. New plots are built by swapping or layering grammar components, rather than by learning a different function for

every chart type. The benefit is that the same vocab covers everything from a histogram to a Sankey diagram to a custom map.

# 16. Advanced ggplot2

## 16.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 16.18.

1. When choosing a colour scale for a visualisation, what property of the data should guide your choice among sequential, diverging, and qualitative scales?
2. Name two appropriate ways to display uncertainty in a visualisation.
3. When designing multi-panel figures (for example, with `facet_wrap()` or `patchwork`), what is the key consideration that supports comparison across panels?

## 16.2. Learning objectives

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

- Compose multi-panel figures with `patchwork` and `cowplot`, with shared legends and panel labels.
- Annotate plots with `ggrepel`, `geom_text`, mathematical expressions, and `annotate()`.
- Build custom scales, themes, and (briefly) custom geoms.
- Produce publication-quality figures with consistent fonts, sizes, and export formats appropriate to the destination (paper, slide deck, web).
- Animate a plot with `gganimate` for teaching and exploratory uses.
- Display uncertainty with error bars, confidence bands, density plots, ribbon plots, and bootstrap distributions.

## 16.3. Orientation

Once the grammar of graphics is in hand, the remaining work in making a finished figure is largely aesthetic: composition, typography, colour, and layout. `ggplot2` and its ecosystem give you fine-grained control over each. This chapter covers the tools used in every real paper or report, the ones that turn 'a plot that conveys the analysis' into 'a plot a journal will accept'.

The pieces fall into four groups: composition (combining multiple plots), annotation (adding labels and highlights), styling (themes and colour palettes), and output (exporting at the right resolution and format).

## 16.4. The statistician's contribution

`ggplot2` defaults are reasonable for exploration; they are not the right choice for a published figure. The adjustments needed to make a figure publication-ready are small in code but consequential in clarity.

**Match the polish to the audience.** A figure for an internal slide deck does not need 600-DPI vector output. A figure for a clinical journal does. Spending an hour fine-tuning a figure for an audience of three is misuse of time; not spending the hour on a figure for a peer-reviewed paper is misuse of opportunity.

**Display uncertainty.** A point estimate without a confidence interval invites the reader to over-trust the estimate. A regression line without a confidence band is a similar invitation. The remedy is small: `geom_errorbar`, `geom_ribbon`, the `se = TRUE` argument to `geom_smooth`. Refusing to show uncertainty because the band looks 'noisy' is wrong; if the data are noisy, the figure should show that.

**Compose deliberately.** Multi-panel figures should make comparisons easy. Shared axes, shared legends, consistent scales, panel labels in the same position. Inconsistent panels create work for the reader; that work is what your figure should be doing for them.

**Choose typography that respects the medium.** Default `ggplot2` text is fine on a slide; on a printed page it often looks small and grey.

Customising the theme is a ten-line investment that pays off across an entire paper.

These judgements determine whether figures get accepted on the first review or send the manuscript back for revisions.

## 16.5. Composing multi-panel figures with patchwork

`patchwork` provides an arithmetic-style operator interface for combining ggplots:

```
library(patchwork)
library(ggplot2)
library(palmerpenguins)

p1 <- ggplot(penguins, aes(flipper_length_mm, body_mass_g, colour = 
        geom_point() +
        labs(title = "A. Body mass vs. flipper length")

p2 <- ggplot(penguins, aes(species, body_mass_g, fill = species)) +
        geom_boxplot() +
        labs(title = "B. Body mass distribution")

p3 <- ggplot(penguins, aes(bill_length_mm, bill_depth_mm, colour = sp
        geom_point() +
        labs(title = "C. Bill morphology")

# horizontal: side by side
p1 + p2

# vertical: stacked
p1 / p2

# 2x2 grid with shared legend
(p1 + p2) / (p3 + p2) +
  plot_layout(guides = "collect") &
  theme(legend.position = "bottom")
```

Operators:

- `+` places plots side by side.
- `/` stacks plots vertically.
- `|` is equivalent to `+` (horizontal).
- Parentheses group plots into sub-arrangements.
- `&` applies a theme or scale to all plots.
- `plot_layout(guides = "collect")` collects shared legends into one location.

For more control, `cowplot::plot_grid()` provides similar functionality with finer-grained alignment options. For inset plots (a small panel inside a larger one), use `patchwork::inset_element()`.

Panel labels (A, B, C) are best added in the `labs()` call of each individual plot, prepended to the title, rather than via patchwork's `plot_annotation(tag_levels = "A")`. The latter places labels in the corners but does not guarantee they appear in the order you want for a non-rectangular layout.

## 16.6. Annotation: text, arrows, and highlights

```
library(ggrepel)

# label specific points (one per row)
ggplot(mtcars, aes(wt, mpg, label = rownames(mtcars))) +
  geom_point() +
  geom_text_repel(size = 3, max.overlaps = 10)

# annotate a single text string at fixed coordinates
ggplot(penguins, aes(flipper_length_mm, body_mass_g)) +
  geom_point() +
  annotate("text", x = 200, y = 6000, label = "Larger species",
           hjust = 0)

# math expressions
ggplot(data.frame(x = 1:10, y = (1:10)^2), aes(x, y)) +
```

```
  geom_line() +
  labs(title = expression(paste("Quadratic: ", y == x^2)),
       y = expression(y ~ "(units)"))

# arrows pointing at features
ggplot(penguins, aes(flipper_length_mm, body_mass_g, colour = specie
  geom_point() +
  annotate("segment", x = 175, xend = 180, y = 5500, yend = 5000,
           arrow = arrow(length = unit(0.3, "cm"))) +
  annotate("text", x = 175, y = 5600, label = "Outlier",
           hjust = 0)
```

`ggrepel` produces non-overlapping text labels by optimising their positions. `max.overlaps` controls how many labels are allowed to overlap before some are dropped. For dense plots, label only a few key points.

## 16.7. Custom themes

A custom theme keeps figures consistent across a paper or a package. Build it once, apply everywhere.

```
theme_phb228 <- function(base_size = 11) {
  theme_minimal(base_size = base_size) +
    theme(
      plot.title       = element_text(face = "bold", size = base_siz
      plot.subtitle    = element_text(colour = "grey40"),
      axis.title       = element_text(face = "bold"),
      axis.text        = element_text(colour = "grey20"),
      panel.grid.minor = element_blank(),
      panel.grid.major.x = element_line(linewidth = 0.2, colour = "g
      panel.grid.major.y = element_line(linewidth = 0.2, colour = "g
      legend.position  = "bottom",
      strip.background = element_rect(fill = "grey95", colour = NA),
      strip.text       = element_text(face = "bold")
    )
}
```

```
ggplot(penguins, aes(flipper_length_mm, body_mass_g, colour = species)) +
  geom_point() +
  scale_colour_brewer(palette = "Dark2") +
  theme_phb228()
```

Set defaults at the top of an analysis script:

```
theme_set(theme_phb228())
```

`update_geom_defaults()` changes default geom parameters (default `geom_point` size, default `geom_line` width):

```
update_geom_defaults("point", list(size = 1.5, alpha = 0.7))
update_geom_defaults("line",  list(linewidth = 0.7))
```

For a project's typography, declare a font family and use it consistently:

```
library(showtext)
font_add_google("Source Sans 3", "source")
showtext_auto()

theme_set(theme_phb228() + theme(text = element_text(family = "source")))
```

`showtext` makes Google Fonts available to ggplot2 and ensures they render correctly when exporting.

## 16.8. Custom colour palettes

For project-wide colour consistency, define your palette once:

```
phb228_palette <- c(
  "Adelie"    = "#1f4e79",
  "Chinstrap" = "#9d2235",
  "Gentoo"    = "#2e8b57"
```

```
)

scale_colour_phb228 <- function(...)
  scale_colour_manual(values = phb228_palette, ...)
scale_fill_phb228 <- function(...)
  scale_fill_manual(values = phb228_palette, ...)

ggplot(penguins, aes(flipper_length_mm, body_mass_g, colour = specie
  geom_point() +
  scale_colour_phb228()
```

For sequential or diverging continuous palettes, `viridis::scale_colour_viridis_c` and `scico::scale_colour_scico(palette = "vik")` are perceptually uniform and colour-blind-safe.

Check your understanding: matching scale to data

**Question.** You are visualising a heatmap of correlations between predictors. Correlations range from -0.9 to +0.9. Which colour scale is most appropriate?
**Answer.**
A diverging palette centred on zero. Correlations have a meaningful midpoint (zero correlation), and positive vs. negative correlations should be visually distinguishable. Standard choices: `RdBu` (red-to-blue, ColorBrewer), `PuOr` (purple-orange), `BrBG` (brown-bluegreen). Set the midpoint of the scale to zero (`midpoint = 0` in `scale_fill_gradient2`) and the limits symmetric around it. A sequential palette would map zero to a particular colour without visual reflection of the sign change. A qualitative palette would imply discrete categories, losing the continuous correlation magnitude.

## 16.9. Displaying uncertainty

Multiple geoms exist for showing uncertainty alongside an estimate.

```
# error bars on a categorical x
ggplot(summarised, aes(group, mean)) +
  geom_point() +
  geom_errorbar(aes(ymin = mean - 2 * se, ymax = mean + 2 * se),
                width = 0.2)

# confidence band on a regression line
ggplot(d, aes(x, y)) +
  geom_point() +
  geom_smooth(method = "lm", se = TRUE)

# ribbon for a manual uncertainty range
ggplot(d, aes(x)) +
  geom_ribbon(aes(ymin = lower, ymax = upper), alpha = 0.3) +
  geom_line(aes(y = mean))

# half-eye plot from posterior or bootstrap samples
library(ggdist)
ggplot(samples, aes(x = group, y = posterior)) +
  stat_halfeye()
```

`ggdist` is particularly worth knowing for Bayesian work or any analysis with full uncertainty distributions: `stat_halfeye`, `stat_dotsinterval`, `stat_lineribbon` make rich uncertainty representations one line each.

For point estimates with CIs in a forest plot:

```
forest_data |>
  ggplot(aes(estimate, term)) +
    geom_pointrange(aes(xmin = conf.low, xmax = conf.high)) +
    geom_vline(xintercept = 0, linetype = "dashed") +
    labs(x = "Effect estimate (95% CI)", y = NULL)
```

For posterior distributions from MCMC, `bayesplot::mcmc_areas` produces a similar interval plot directly from posterior samples.

## 16.10. Exporting for publication

```
# raster: PNG at high DPI for slides, web
ggsave("figure.png", plot = p, width = 6, height = 4,
       dpi = 300, units = "in")

# vector: PDF for LaTeX submissions
ggsave("figure.pdf", plot = p, width = 6, height = 4,
       device = cairo_pdf)

# vector: SVG for the web
ggsave("figure.svg", plot = p, width = 6, height = 4)

# specific journal sizing (single column, double column)
ggsave("figure_singlecol.pdf", plot = p,
       width = 90 / 25.4, height = 60 / 25.4, units = "in",
       device = cairo_pdf)
```

Three things to know about export:

1. **Use vector formats** (PDF, SVG, EPS) for line art and typography. Raster formats (PNG, JPG) lose quality on zoom. JPG additionally adds compression artefacts; never use JPG for plots.
2. **DPI matters for raster.** 300 DPI is the standard for print; 600 for high-quality scientific figures. 72 DPI is web display only.
3. **Embed fonts in PDFs.** `device = cairo_pdf` ensures non-default fonts are embedded so the figure looks the same on systems that lack the font. Without this, the recipient may see a fallback font that looks wrong.

For journal submissions, check the figure size requirements (typically single-column 85–90 mm, double-column 170–180 mm) and produce figures at exactly the target size, not larger. Plots designed at 6×4 inches and submitted at 3 inches wide have illegible labels.

## 16.11. Animation with gganimate

```
library(gganimate)

# bootstrap convergence: posterior of mean as n grows
boot_data <- expand_grid(rep = 1:100, n = c(10, 50, 100, 500)) |>
  mutate(mean_est = map2_dbl(rep, n, \(r, k) mean(rnorm(k))))

p <- ggplot(boot_data, aes(mean_est)) +
  geom_histogram(bins = 30) +
  labs(title = "Sampling distribution of the mean, n = {closest_state}"
       x = "Sample mean") +
  transition_states(n, transition_length = 2, state_length = 1)

animate(p, nframes = 100, fps = 10)
anim_save("convergence.gif")
```

`gganimate` supports several transitions: `transition_states` (discrete steps), `transition_time` (continuous), `transition_reveal` (incrementally reveal a line), and others. For teaching and exploratory work, animation is often the clearest way to show how a distribution evolves with sample size, iterations, or parameter changes.

For papers, animation is rarely useful: figures are static. For slide decks, blog posts, and supplementary materials, animation can convey what static figures cannot.

## 16.12. Worked example: regression diagnostics in three panels

```
library(ggplot2)
library(patchwork)
library(broom)
```

```
fit <- lm(body_mass_g ~ flipper_length_mm + species, data = na.omit(
diag_data <- augment(fit)

p_resid <- ggplot(diag_data, aes(.fitted, .resid)) +
  geom_point(alpha = 0.5) +
  geom_smooth(method = "loess", se = FALSE, colour = "red") +
  labs(title = "A. Residuals vs. fitted",
       x = "Fitted values", y = "Residuals")

p_qq <- ggplot(diag_data, aes(sample = .std.resid)) +
  geom_qq(alpha = 0.5) +
  geom_qq_line(colour = "red") +
  labs(title = "B. Normal Q-Q",
       x = "Theoretical quantiles",
       y = "Standardised residuals")

p_cook <- ggplot(diag_data, aes(seq_len(nrow(diag_data)), .cooksd)) +
  geom_col(width = 0.5) +
  geom_hline(yintercept = 4 / nrow(diag_data),
             linetype = "dashed", colour = "red") +
  labs(title = "C. Cook's distance",
       x = "Observation index",
       y = "Cook's distance")

(p_resid + p_qq) / p_cook + plot_layout(heights = c(1, 0.7))
```

This composition reads naturally: top row two related diagnostics (residual structure and Q-Q), bottom row a single observation-level diagnostic. Panel labels carry the reader through.

## 16.13. Collaborating with an LLM on advanced ggplot

LLMs handle ggplot composition reasonably well; they handle typography and journal conventions less reliably.

**Prompt 1: matching a journal style.** Paste the journal's figure guidelines (or the URL) and ask: 'write a `theme_journal()` function that matches.'

*What to watch for.* The output theme is a starting point. Specific journal requirements (font family, font size at print, panel border vs. axis lines) often need to be verified by hand against the actual published figures.

*Verification.* Generate a sample figure in your theme and a sample figure from a recent published paper. Compare side by side. Adjust until indistinguishable.

**Prompt 2: combining plots.** Describe four plots and ask: 'combine these into a 2x2 grid with shared legend and panel labels A, B, C, D.'

*What to watch for.* Most LLM solutions use `patchwork`, which is correct. Watch for legend handling: the LLM may forget `plot_layout(guides = "collect")` or place legends outside the plot area in unexpected ways.

*Verification.* Render the combined plot at the target size. Are the legends in a single shared location? Are the axes consistent? Are the panel labels in the right position?

**Prompt 3: animation.** Describe what you want to animate (e.g., 'how does the bootstrap distribution converge as $n$ grows?') and ask the LLM to produce `gganimate` code.

*What to watch for.* The animation may run too fast (no time to read each frame) or too slow (boring). Adjust `nframes` and `fps`.

*Verification.* Watch the animation. Does it tell the story you wanted? If the message gets lost in the movement, an animated plot is not the right medium.

## 16.14. Principle in use

Three habits define defensible advanced visualisation:

1. **Customise once, apply everywhere.** A `theme_phb228()` function and a project palette make every figure consistent for free.

2. **Show uncertainty.** Confidence bands, error bars, posterior intervals, whatever the analysis produces. Never report point estimates without their uncertainty.
3. **Export for the destination.** PDF for LaTeX, PNG for slides, SVG for web. Embed fonts. Match journal sizing.

## 16.15. Exercises

1. Build a three-panel figure: (a) raw data scatter; (b) residuals-vs-fitted; (c) QQ plot. Combine with `patchwork` and add panel labels 'A', 'B', 'C'.
2. Write a custom `theme_phb228()` function with serif body text, sans-serif axis titles, and a colour-blind- safe default palette. Apply it to three plots.
3. Export a figure at 300 DPI as both PDF (for LaTeX) and PNG (for Word). Open both and verify the fonts are embedded correctly.
4. Make a forest plot of effect estimates with 95% CIs for ten coefficients from a regression. Use `geom_pointrange` and a vertical reference line at zero.
5. Animate a sampling distribution converging as $n$ grows. Use `gganimate::transition_states`. Save as GIF and verify the animation tells the story.

## 16.16. Further reading

- (Wickham, 2016), chapters on scales, themes, and extending `ggplot2`. The 3rd edition is online at `ggplot2-book.org`.
- (Wilke, 2019), the source of many of the design principles this chapter invokes.
- (Healy, 2018), shorter, applied, with R code.
- The `gganimate` and `patchwork` package vignettes are excellent and concise.

## 16.17. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 16.17.1. Question 1

Which of the following is MOST important when choosing a colour scale for a data visualisation?

- A) Using the widest possible range of different colours
- B) Matching the colour scale to the type of data being visualised (sequential, diverging, or qualitative)
- C) Always using the same colour scheme across all visualisations in a project
- D) Prioritising aesthetically pleasing colour combinations over all other considerations

> **i** Answer
>
> **B.** Sequential, diverging, and qualitative scales each encode a different kind of variable structure; mismatching distorts interpretation.

### 16.17.2. Question 2

Which approach is recommended when displaying uncertainty in your data?

- A) Omit uncertainty information to avoid confusing the audience
- B) Only show the mean or median values as single points
- C) Always show exact numerical values for uncertainty in a caption rather than visualising it
- D) Use visual elements like error bars, confidence bands, or density plots to represent uncertainty

Answer

**D.** Uncertainty should be visualised alongside the estimate rather than omitted or relegated to captions.

### 16.17.3. Question 3

When designing multi-panel figures, which is the most important design consideration?

- A) Always arrange panels in a perfect grid with equal dimensions regardless of the data
- B) Use as many panels as possible to show every possible data combination
- C) Maintain consistent scale and layout across panels to facilitate comparisons
- D) Avoid panels altogether and instead create a single complex figure

Answer

**C.** Consistent scales and layouts mean that visual differences across panels reflect data differences, not display differences.

### 16.17.4. Question 4

You want to combine four ggplots into a 2x2 grid with a shared legend. Which approach uses the canonical R tool?

- A) `gridExtra::grid.arrange(plot1, plot2, ...)` then manually add the legend.
- B) `patchwork::wrap_plots(p1, p2, p3, p4) + plot_layout(guides = "collect")`.
- C) Save each plot, open in Photoshop, manually combine.
- D) `cowplot::plot_grid()` with default arguments.

Answer

**B.** `patchwork` is the canonical modern composition tool; `guides = "collect"` is what shares the legend. `cowplot::plot_grid` works similarly but with a different syntax.

### 16.17.5. Question 5

For a journal submission, you should export figures as:

- A) JPG at 72 DPI.
- B) PNG at 96 DPI.
- C) PDF (vector) with embedded fonts, sized to the journal's column width.
- D) PowerPoint (.pptx) with editable layers.

Answer

**C.** Vector PDFs scale to any size without quality loss. Embedding fonts (`device = cairo_pdf`) ensures the figure renders correctly on any system. Sizing at the column width avoids the journal scaling your figure down and making labels illegible.

## 16.18. Prerequisites answers

1. Match the colour scale to the type of data: sequential for ordered or continuous data, diverging for data with a meaningful midpoint, qualitative for unordered categorical data. Using the wrong family distorts perception. Sequential and diverging scales should be perceptually uniform (viridis, ColorBrewer) and colour-blind-safe.
2. Error bars, confidence bands, density plots, violin plots, point clouds, ribbons. Do not omit uncertainty information or replace it with a numerical value in a caption. The figure should encode estimate and uncertainty together.

3. Maintain consistent axis scales and panel layout so that apparent differences across panels reflect real differences in the data, not differences in the display. Shared scales (`facet_*` or `patchwork::plot_layout(axis_titles = "collect")`) are the standard mechanism.

# 17. Interactive Visualization with Shiny

## 17.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 17.20.

1. What are the two top-level components of every Shiny application?
2. What does `reactive()` do, and how does its behaviour differ from that of a plain R function?
3. How are outputs wired between the UI and the server in a Shiny application?

## 17.2. Learning objectives

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

- Explain Shiny's reactive programming model: inputs, outputs, reactive expressions, observers.
- Build a single-file Shiny app with `ui`, `server`, and `shinyApp()`.
- Distinguish `reactive()` (cached, recomputed on dependency change, used to *produce a value*) from `observeEvent()` (run for side effects, no cached return value).
- Use `bindEvent()` (a more general decorator added in Shiny 1.6 alongside `eventReactive()`) to control when an expensive reactive recomputes.
- Modularise an app using Shiny modules with namespace isolation.
- Deploy an app to `shinyapps.io` or a Posit Connect server.
- Debug reactivity with `reactlog` and trace through a reactivity graph.

## 17.3. Orientation

An interactive application lets collaborators explore a model's behaviour without running any code. For a biostatistician, this means writing less one-off code in response to every new question a collaborator asks: 'what does the predicted survival look like at age 70?' becomes a slider on a deployed app rather than a request for a new PDF. Shiny is the dominant tool for building such applications in R.

The framework was created by Joe Cheng at RStudio (now Posit) and has matured over a decade into the standard for analytic dashboards in R. The Mastering Shiny book by Hadley Wickham (mastering-shiny.org) is the canonical modern reference.

## 17.4. The statistician's contribution

Shiny apps look easy until you maintain one. The pitfalls are predictable; the judgements that prevent them are not in the framework.

**Decide what should be reactive.** Every input change can trigger every downstream computation. Sometimes that is right (a small dataset filter); sometimes it is disastrous (a 30-second model fit running every time the user moves a slider by 1). The right reactive granularity, often involving `bindEvent()` to delay expensive computations until a 'Run' button is pressed, is the analyst's judgement, not the framework's default.

**Validate inputs.** A Shiny app is a public surface, even when 'public' means 'visible to your three collaborators'. Inputs that crash the server because the user uploaded a CSV with extra columns, or set a slider to zero, are bugs. `validate()` and `req()` catch them gracefully. Defensive validation is the cost of a robust app.

**Modularise.** A single-file app of 200 lines is fine. A single file of 2000 lines is unmaintainable. Shiny modules let you encapsulate UI/server pairs that handle one component (a dataset selector, a plot, a table) and combine them with namespace isolation. Splitting the monolith is the difference between an app that survives a year of feature requests and one that gets rewritten.

**Be deliberate about deployment.** A `shinyapps.io` free tier app times out after idle minutes. A self-hosted Shiny Server has different scaling and security characteristics. A Posit Connect deployment integrates with internal authentication. Pick the destination that matches the audience; do not assume.

These judgements are what distinguishes a deployed, shared analysis from a local-only exploration toy.

## 17.5. Reactive programming in one page

Shiny's reactive model has three building blocks:

- **Reactive sources.** Inputs (`input$x`) and external state. They produce values that change over time.
- **Reactive expressions.** Computations that depend on reactive sources. They are *lazy* (only run when their output is needed) and *cached* (rerun only when a dependency changes).
- **Observers.** Computations that run for side effects (writing files, updating an external database). Run whenever a dependency changes; do not return a cached value.

The dependency graph is built automatically. When you write `reactive({ filter(data, year == input$year) })`, Shiny notes that the expression depends on `input$year`. When `input$year` changes, the expression's cached value is invalidated; the next time anything asks for it, the expression recomputes.

This is a different mental model from imperative R. You do not call functions in order; you declare relationships and let the framework figure out execution. The payoff is that the UI updates automatically when inputs change. The cost is that bugs are about *when* things happen, which is harder to debug than *what* they compute.

## 17.6. A minimum Shiny app

```
library(shiny)

ui <- fluidPage(
  titlePanel("Penguin scatter"),
  sidebarLayout(
    sidebarPanel(
      selectInput("xvar", "X variable",
                  choices = c("flipper_length_mm",
                              "bill_length_mm",
                              "body_mass_g")),
      selectInput("yvar", "Y variable",
                  choices = c("body_mass_g",
                              "bill_depth_mm"))
    ),
    mainPanel(
      plotOutput("scatter")
    )
  )
)

server <- function(input, output, session) {
  output$scatter <- renderPlot({
    data <- na.omit(palmerpenguins::penguins)
    ggplot(data,
           aes(.data[[input$xvar]],
               .data[[input$yvar]],
               colour = species)) +
      geom_point() +
      theme_minimal()
  })
}

shinyApp(ui, server)
```

Five things to notice:

1. **`ui` is HTML in disguise.** `fluidPage`, `sidebarLayout`, and friends are R functions that emit HTML.
2. **Each input has an id and a server-side counterpart.** `selectInput("xvar", ...)` in the UI corresponds to `input$xvar` in the server.
3. **Each output has a UI placeholder and a server `render*` function paired by id.** `plotOutput("scatter")` in the UI; `output$scatter <- renderPlot({ ... })` in the server.
4. **`renderPlot({ ... })` is reactive.** It re-runs every time any reactive value referenced in the body changes , here, `input$xvar` and `input$yvar`.
5. **`shinyApp(ui, server)` returns an app object** that can be run interactively (in the console) or deployed.

The `.data[[input$xvar]]` syntax inside `aes()` lets us use a string-valued input as a data column reference; this is the modern tidy-eval idiom for column selection by character.

## 17.7. Inputs, outputs, reactives

Inputs are widgets the user controls:

- `numericInput`, `sliderInput`, `selectInput`, `checkboxInput`, `radioButtons`.
- `textInput`, `passwordInput`, `dateInput`, `dateRangeInput`.
- `fileInput` for uploads, `actionButton` for triggering events.

Outputs are placeholders for content the server fills in:

- `plotOutput`, `tableOutput`, `dataTableOutput`, `verbatimTextOutput`, `textOutput`, `htmlOutput`, `uiOutput`.

`render*` functions on the server side populate them:

- `renderPlot`, `renderTable`, `renderDT`, `renderPrint`, `renderText`, `renderUI`.

Reactives are intermediate computations:

```
server <- function(input, output, session) {
  # cached intermediate
  filtered <- reactive({
    data |> filter(year == input$year)
  })

  # plot uses filtered() (note the parentheses!)
  output$plot <- renderPlot({
    ggplot(filtered(), aes(x, y)) + geom_point()
  })

  # table uses the same filtered() value, not recomputed
  output$table <- renderTable({
    summarise(filtered(), n = n(), mean = mean(y))
  })
}
```

The reactive `filtered()` is shared by both outputs. Without it, `filter(data, year == input$year)` would run twice. With it, it runs once and the cached value is shared.

## 17.8. `reactive()` vs `observeEvent()`

A `reactive()` produces a value (lazy, cached). An `observeEvent()` runs for side effects (eager, no cached return value):

```
# computes a value, used by downstream code
filtered <- reactive({ filter(data, year == input$year) })

# performs an action, no value
observeEvent(input$save_button, {
  saveRDS(filtered(), "results.rds")
  showNotification("Saved")
})
```

Use `reactive()` when you want a cached intermediate value to be referenced by multiple downstream reactives or outputs. Use `observeEvent()` when you want to do something *because* an input changed (or a button was clicked) but do not need a return value.

`observe()` (without `Event`) runs whenever any of its dependencies changes. `observeEvent(trigger, expr)` runs only when `trigger` changes; `expr` cannot reference reactives directly without isolation.

## 17.9. Controlling expensive recomputation

The default reactive model recomputes whenever a dependency changes. For expensive computations (a long model fit, a database query, an external API call), this is unacceptable: the user wants the model to update only when they have finished entering parameters.

`bindEvent()` (added in Shiny 1.6 alongside the older `eventReactive()`) does this:

```
# expensive fit, only re-runs when run_button is clicked
fit <- bindEvent(reactive({
  Sys.sleep(5)            # pretend this is expensive
  lm(y ~ x, data = filtered())
}), input$run_button)

# fit() returns the most recent fit (or NULL initially)
output$summary <- renderPrint({ summary(fit()) })
```

For an `observeEvent`, the syntax is similar:

```
observeEvent(input$run_button, {
  fit <- lm(y ~ x, data = filtered())
  saveRDS(fit, "fit.rds")
})
```

The Mastering Shiny book ('Reactivity in depth' chapter) is essential reading for getting reactive timing right.

Check your understanding: reactive vs. observer

**Question.** You want your Shiny app to fit a regression when the user clicks a 'Fit' button, then update both a plot and a table with the result. Should the fit be a `reactive()`, an `observeEvent()`, or both?

**Answer.**

A `reactive()` triggered by `input$run_button`:

```
fit <- bindEvent(reactive({
  lm(y ~ x, data = filtered())
}), input$run_button)
```

The plot and table both reference `fit()`, so the fit is computed once and the cached result is shared. Using an `observeEvent` would not give you a cached return value to share. Using a plain `reactive()` would re-run on every input change, defeating the purpose of the button. The combination of `reactive()` and `bindEvent()` is the right pattern.

## 17.10. Validating inputs

`validate()` and `req()` halt reactive evaluation when inputs are not in valid states:

```
output$result <- renderPlot({
  req(input$file)                    # need a file
  data <- read.csv(input$file$datapath)
  validate(
    need(nrow(data) > 0,             "File is empty"),
    need(ncol(data) >= 2,            "Need at least 2 columns"),
    need(input$xvar %in% names(data), "Selected x variable not present"
  )
  ggplot(data, aes(.data[[input$xvar]], .data[[input$yvar]])) +
    geom_point()
})
```

`req(input$file)` halts evaluation silently if the user has not yet uploaded a file, nothing renders, no error shown.

`validate(need(...))` halts evaluation and displays a human-readable message explaining why. Useful when the user has done something almost-but-not-quite right.

Without these, an app with a missing file or invalid input throws server-side errors that may render as red text in the UI or stack traces in the log. Validate deliberately.

## 17.11. Shiny modules

A Shiny module is a UI/server pair that handles one component, with its own namespace. Modules are how single-file apps become maintainable multi-file applications.

```
# module: a dataset selector with a preview table
dataset_ui <- function(id) {
  ns <- NS(id)
  tagList(
    selectInput(ns("dataset"), "Dataset",
                choices = c("mtcars", "iris", "penguins")),
    tableOutput(ns("preview"))
  )
}

dataset_server <- function(id) {
  moduleServer(id, function(input, output, session) {
    data <- reactive({
      switch(input$dataset,
             mtcars   = mtcars,
             iris     = iris,
             penguins = na.omit(palmerpenguins::penguins))
    })

    output$preview <- renderTable({ head(data()) })
```

```
    return(data)            # return the reactive for caller to use
  })
}

# top-level app uses the module
ui <- fluidPage(
  dataset_ui("data1"),
  plotOutput("scatter")
)

server <- function(input, output, session) {
  data <- dataset_server("data1")
  output$scatter <- renderPlot({
    ggplot(data(), aes(.data[[names(data())[1]]],
                       .data[[names(data())[2]]])) +
      geom_point()
  })
}
```

Three things modules give you:

1. **Namespace isolation.** `NS(id)` ensures that two instances of the same module do not collide on input ids.
2. **Reusability.** Define once, instantiate many.
3. **Testability.** Each module can be tested in isolation.

For an app larger than a few hundred lines of code, modules are the canonical way to keep the codebase organised.

## 17.12. Deployment

The three common destinations:

**`shinyapps.io`** is Posit's hosted service. Free tier allows a few apps with limited active hours per month. `rsconnect::deployApp()` deploys with a few clicks. Public URLs; basic authentication available on paid plans.

**Posit Connect** is the enterprise self-hosted platform. Integrates with internal authentication, scales across multiple servers, supports private deployments. Used by most large organisations running R in production.

**Self-hosted Shiny Server (open source).** Free, but you manage the infrastructure: server provisioning, scaling, authentication, monitoring. The 'just run a Shiny app on my Linux box' option.

For deployment to shinyapps.io:

```
library(rsconnect)
deployApp(appDir = ".", appName = "penguin-explorer",
          forceUpdate = TRUE)
```

For each: read the deployment documentation. The `{rsconnect}` package handles the basics; production deployments often require attention to memory limits, session timeouts, and authentication.

## 17.13. Debugging reactivity

`reactlog::reactlog_enable()` records every reactive event. Run the app, interact with it, then call `reactlog::reactlog_show()` to see the dependency graph and event history:

```
options(shiny.reactlog = TRUE)
runApp("my_app")
# (interact)
reactlog::reactlog_show()
```

The interactive viewer shows which reactives recomputed when, which dependencies triggered which updates, and which observers ran in response to which events. For debugging an app where things update at the wrong time or not at all, `reactlog` is the canonical tool.

For server-side errors, `shiny::runApp(launch.browser = TRUE)` in development gives you a console that traces messages. For production, the deployment platform's logs (Posit Connect, shinyapps.io) capture stack traces.

## 17.14. Worked example: regression explorer

```
library(shiny)
library(palmerpenguins)
library(ggplot2)
library(broom)

penguins_clean <- na.omit(penguins)

ui <- fluidPage(
  titlePanel("Penguin regression explorer"),
  sidebarLayout(
    sidebarPanel(
      selectInput("response", "Response",
                  choices = c("body_mass_g",
                              "bill_length_mm")),
      checkboxGroupInput("predictors", "Predictors",
                         choices = c("flipper_length_mm",
                                     "bill_depth_mm",
                                     "species",
                                     "sex"),
                         selected = c("flipper_length_mm",
                                      "species")),
      actionButton("fit", "Fit")
    ),
    mainPanel(
      plotOutput("scatter"),
      tableOutput("coefficients")
    )
  )
)

server <- function(input, output, session) {
  formula <- reactive({
    req(input$predictors)
    as.formula(paste(input$response, "~",
                     paste(input$predictors, collapse = " + ")))
```

```
  })

  fit <- bindEvent(reactive({
    lm(formula(), data = penguins_clean)
  }), input$fit)

  output$scatter <- renderPlot({
    req(fit())
    diag <- augment(fit())
    ggplot(diag, aes(.fitted, .resid)) +
      geom_point() +
      geom_smooth(method = "loess", se = FALSE) +
      labs(title = "Residuals vs. fitted")
  })

  output$coefficients <- renderTable({
    req(fit())
    broom::tidy(fit(), conf.int = TRUE)
  })
}

shinyApp(ui, server)
```

The user picks a response, predictors, and clicks 'Fit'. The fit recomputes only when the button is clicked, not on every checkbox change. The plot and table both share the cached `fit()`. `req()` ensures graceful handling before the first fit.

This is a small but realistic pattern: an interactive exploration of a model where the expensive step (the fit) is gated behind a button.

## 17.15. Collaborating with an LLM on Shiny

Shiny apps are an area where LLMs can produce a lot of working code quickly, and where reactivity bugs hide in plain sight.

**Prompt 1: drafting an app.** Describe the app: what should the user see, what inputs should they have, what output should appear. Ask: 'write a single-file Shiny app implementing this. Use `bindEvent()` for any expensive computation that should be gated behind a button.'

*What to watch for.* Default LLM apps tend to be too reactive: every input change triggers everything. For expensive computations, the gating with `bindEvent()` / `actionButton()` is essential.

*Verification.* Run the app. Try moving sliders fast and see whether the app keeps up. If it stutters, the computation is too aggressive; refactor with `bindEvent`.

**Prompt 2: diagnosing a reactivity bug.** Paste the relevant chunk of `server` code and describe the symptom: 'when I change input X, output Y does not update' or 'when I change input X, output Y updates twice'. Ask the LLM to diagnose.

*What to watch for.* The LLM should recognise common patterns: missing parentheses on a reactive call (`fit` vs. `fit()`), `observeEvent` where `reactive` was needed, or vice versa. If the diagnosis is generic, push for specifics.

*Verification.* Apply the fix and observe whether the bug goes away. Use `reactlog` if the bug is subtle.

**Prompt 3: refactoring to modules.** Paste the single-file app and ask: 'refactor this into Shiny modules. Identify reasonable component boundaries.'

*What to watch for.* The LLM should respect namespace boundaries (using `NS(id)` consistently). Common errors include forgetting `ns()` on input ids in the module UI, or using global state that breaks with multiple module instances.

*Verification.* Run the refactored app. If you have used the module's input ids correctly, instantiating two copies of the module should not cause id collisions.

## 17.16. Principle in use

Three habits define defensible Shiny development:

1. **Match reactive granularity to cost.** Cheap computations can run on every input change; expensive ones need `bindEvent()` and an explicit trigger.
2. **Validate inputs.** `req()` and `validate()` are a small investment that produces graceful failure modes.
3. **Modularise as the app grows.** Single-file apps above a few hundred lines should be split into modules. Modules also enable unit testing.

## 17.17. Exercises

1. Build a Shiny app that lets the user pick a dataset from a drop-down, choose x and y variables, and view a scatter plot. Add a slider for the `loess` smoothing span.
2. Deploy the app from exercise 1 to shinyapps.io (free tier). Share the URL with a classmate and have them try to break it. Fix whatever they break.
3. Convert the app into a two-module design: a `dataset_ui/server` module for data selection and a `plot_ui/server` module for plotting. Explain why modular code is easier to test.
4. Add an upload-CSV input that lets the user supply their own data. Add `req()` and `validate()` calls so the app handles bad uploads gracefully (empty file, non-CSV file, file with no numeric columns).
5. Use `reactlog` to inspect the dependency graph of your app from exercise 4. Are any reactives being re-evaluated more than necessary? Optimise.

## 17.18. Further reading

- (Wickham, 2021), *Mastering Shiny*, the canonical modern reference, free at mastering-shiny.org.

- (Sievert, 2020), interactive graphics with plotly, which pairs naturally with Shiny for dashboards.
- The `shiny` package vignettes, especially 'Reactivity: An overview'.

## 17.19. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 17.19.1. Question 1

What are the two main components that make up every Shiny application?

- A) HTML and JavaScript functions
- B) User Interface (UI) and Server components
- C) Input widgets and reactive functions
- D) Plotly objects and ggplot2 visualisations

> **Answer**
>
> **B.** Every Shiny app is defined by a UI (inputs and output placeholders) and a server (logic computing outputs from inputs).

### 17.19.2. Question 2

What is the primary purpose of the `reactive()` function?

- A) To render graphics that update automatically
- B) To create user interface elements like sliders or dropdown menus
- C) To define computations that automatically update when their inputs change, with caching
- D) To establish a connection between the server and the user's web browser

Answer

**C.** `reactive()` caches its value and re-runs only when a reactive dependency changes.

### 17.19.3. Question 3

How are outputs created in a Shiny application?

- A) Outputs are created using HTML tags in the UI and do not require server-side processing
- B) Each output must be created with its own separate `app.R` file
- C) Output functions in the UI must have corresponding render functions in the server, paired by id
- D) Shiny outputs can only display static content

Answer

**C.** UI output functions (e.g., `plotOutput("p")`) are paired by id with server render functions (e.g., `output$p <- renderPlot({...})`).

### 17.19.4. Question 4

You want a long-running model fit to recompute only when the user clicks a 'Run' button. Which Shiny construct implements this?

- A) `reactive()` alone.
- B) `observeEvent(input$run_button, ...)` returning a value.
- C) `reactive()` combined with `bindEvent()` (or equivalently `eventReactive()`).
- D) `isolate()`.

Answer

**C.** `bindEvent()` (or its older synonym `eventReactive`) wraps a reactive so it recomputes only when the named trigger changes.

### 17.19.5. Question 5

The recommended approach to debugging Shiny reactivity is:

- A) Add `print()` statements throughout the server.
- B) Use `reactlog::reactlog_show()` to inspect the dependency graph and event history.
- C) Run the app in a separate R process and read the log files.
- D) Avoid reactivity altogether by hardcoding values.

> **Answer**
>
> **B.** `reactlog` is the canonical tool for tracing which reactives ran when, in response to which events.

## 17.20. Prerequisites answers

1. A user-interface (UI) function and a server function. The UI defines inputs and output placeholders; the server defines how outputs are computed from inputs. `shinyApp(ui, server)` ties them into an app object.
2. `reactive()` wraps a computation so that its value is cached and recomputed only when a reactive dependency (typically `input$*` or another reactive) changes. A plain R function recomputes every time it is called, regardless of whether its inputs have changed. Reactives are referenced with parentheses (`fit()`) to retrieve their current value.
3. Each output placeholder in the UI has a companion `render*()` function in the server, paired by a shared id. For example, `plotOutput("p")` in the UI is populated by `output$p <- renderPlot({ ... })` in the server. Shiny tracks the dependencies of the `renderPlot` body and re-runs it whenever a referenced reactive value changes.

# Part VI.

# Scaling and Software Engineering

# 18. Parallel Computing in R

## 18.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 18.17.

1. What is the primary advantage of parallel computing for statistical problems?
2. What kind of statistical computation is most suitable for parallel processing, and why?
3. Why is using the maximum number of available cores not always the optimal choice?

## 18.2. Learning objectives

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

- Distinguish between embarrassingly parallel, data-parallel, and inherently sequential problems.
- Parallelise a simulation or bootstrap with `future.apply` or `furrr`.
- Pick an appropriate backend (`multisession`, `multicore`, `cluster`) for your platform.
- Generate reproducible random numbers in parallel via `future.seed = TRUE` (which uses the L'Ecuyer-CMRG generator).
- Report speedup and efficiency, accounting for overhead and Amdahl's law.
- Avoid common pitfalls: incorrect RNG streams, global state, large object serialisation, BLAS thread oversubscription.

## 18.3. Orientation

Most simulations and bootstraps are *embarrassingly parallel*: each iteration is independent of the others. Running them in parallel often produces near-linear speedup with a trivial code change. This chapter shows how to do that correctly, and how to recognise the cases where parallelism does not help.

The modern R parallel ecosystem centres on the `future` package by Henrik Bengtsson. `future`, `future.apply`, and `furrr` provide a unified API that abstracts over backend choice (cores on a single machine, multiple machines, HPC clusters). Older packages (`parallel`, `foreach`, `doParallel`) still work and are widely deployed, but `future` is the recommended modern interface for most applications.

## 18.4. The statistician's contribution

Parallelism is a tool for trading developer time for machine time. The judgements are about when the trade is worth it.

**Profile before parallelising.** A simulation that takes 30 seconds in serial does not need parallelism; the time spent setting up `future.apply` exceeds the saving. A simulation that takes 30 minutes is worth parallelising; 30 hours is essential. Knowing which case you are in requires profiling first, not parallelising preemptively.

**Reproducibility is non-trivial.** Naive parallel code with `set.seed()` does not produce reproducible results across workers. Each worker can draw from the same RNG stream, producing correlated (or identical) results, or from independent streams set up with the L'Ecuyer-CMRG generator. Detecting the difference is the analyst's responsibility; software defaults vary across packages.

**Mind the data.** Parallel workers receive copies of the global state. A 5 GB dataset in your R session becomes 5 GB on each worker; with 8 workers, that is 40 GB of memory used. For small data this is fine; for large data, you need to load data inside the worker function or use shared-memory techniques.

**Mind the BLAS.** R's matrix operations dispatch to BLAS, which may itself be multi-threaded (OpenBLAS, MKL). Running 8 R workers each with 8 BLAS threads is 64 threads competing on probably 8 physical cores. The typical fix: `RhpcBLASctl::blas_set_num_threads(1)` inside the worker so BLAS does not spawn additional threads. This pitfall is invisible until you measure and the 'parallel' code is somehow *slower* than serial.

These judgements are what distinguishes parallel code that delivers a speedup from parallel code that delivers a speedup *and* correct, reproducible results.

## 18.5. When does parallelism help?

Three categories of problem:

**Embarrassingly parallel.** Each task is independent of the others. Bootstrap resamples, Monte Carlo simulation replicates, fitting the same model to many strata. Speedup scales nearly linearly with the number of cores, up to overhead. The dominant case in statistical computing.

**Data parallel.** A single computation can be split across cores by partitioning the data: matrix multiplication, large-scale sum, distributed gradient descent. Useful but harder to implement; modern BLAS handles small-scale data parallelism automatically. For massive data, frameworks like Spark or `arrow` are typically more appropriate.

**Inherently sequential.** Each step depends on the previous: a single MCMC chain, a stochastic gradient descent trajectory, a forward simulation of a dynamical system. No amount of parallel hardware can speed up a serial dependency.

For sequential problems, you can sometimes parallelise *at a higher level*: run several MCMC chains in parallel (common). Or split the data into independent batches and fit separate models to each. The trick is identifying the boundary where independence holds.

## 18.6. The `future` framework

`future` provides a unified interface across parallel backends:

```
library(future)

# choose a backend
plan(sequential)        # no parallelism (default)
plan(multisession)      # multiple R sessions on this machine
plan(multicore)         # forking, Linux/macOS only
plan(cluster, workers = 4)
plan(future.batchtools::batchtools_slurm)   # HPC cluster
```

`multisession` is the cross-platform default. It starts $k$ fresh R processes; each receives a copy of the workspace as needed.

`multicore` (Linux/macOS only) uses operating-system forking. Faster startup than `multisession` and shared memory until a worker writes (copy-on-write). Does not work in RStudio because of fork-safety issues.

`cluster` lets you specify worker machines explicitly, useful for HPC.

The 'plan' is set globally for the session. Code using the `future` API (or `future.apply`, or `furrr`) runs according to the current plan without code changes.

## 18.7. `future.apply` and `furrr`

These packages re-implement the `apply` and `purrr` families on top of `future`:

```
library(future)
library(future.apply)

plan(multisession, workers = 4)

# parallel sapply / lapply / mapply / vapply
```

```
results <- future_sapply(1:1000,
                         FUN = function(i) {
                           # one bootstrap replicate
                           x <- sample(data, length(data), replace =
                           mean(x)
                         },
                         future.seed = TRUE)
```

`future.seed = TRUE` activates the L'Ecuyer-CMRG generator internally, ensuring each worker uses an independent non-overlapping RNG stream. Without it, you may get correlated (or identical) sequences across workers.

`furrr` does the same for the `purrr` map functions:

```
library(furrr)
plan(multisession, workers = 4)

results <- future_map_dbl(1:1000,
                          ~ {
                            x <- sample(data, length(data), replace =
                            mean(x)
                          },
                          .options = furrr_options(seed = TRUE))
```

For most modern code, `furrr` is the recommended choice because the API matches `purrr`; existing serial code using `map_dbl` becomes parallel by replacing it with `future_map_dbl`.

Check your understanding: parallel RNG

**Question.** You parallelise a bootstrap with `future_map_dbl(1:1000, my_boot_fn)` without `furrr_options(seed = TRUE)`. The results across runs are not reproducible, and worse, two workers sometimes produce identical sequences of bootstrap statistics. What is going on?

**Answer.**

Without explicit parallel-safe RNG handling, each worker inherits the parent process's RNG state at the moment of forking (or starts with an

undefined state, depending on backend). Two workers can end up using identical seeds and produce correlated or identical sequences. The solution is `furrr_options(seed = TRUE)` (or `future.seed = TRUE` for `future_*apply`), which activates the L'Ecuyer-CMRG generator and assigns each worker an independent stream. The behaviour is deterministic given the parent seed, so re-running with the same parent seed produces identical results across workers and across re-runs. Forgetting this option is among the most common bugs in parallel R code, partly because it produces results that look reasonable but are silently miscalibrated.

## 18.8. Measuring speedup

Three quantities matter:

- **Speedup**: $T_1/T_n$, the ratio of serial to parallel time.
- **Efficiency**: speedup divided by the number of workers. 100% means linear scaling; below 100% means overhead is reducing returns.
- **Amdahl's law**: the maximum speedup is bounded by $1/(s+(1-s)/n)$, where $s$ is the serial fraction and $n$ is the number of workers. If 10% of the computation is inherently serial, maximum speedup is $10\times$ no matter how many workers you have.

```
library(microbenchmark)

# serial
t_serial <- system.time({
  results_serial <- purrr::map_dbl(1:1000, my_boot_fn)
})

# parallel
plan(multisession, workers = 4)
t_par <- system.time({
  results_par <- furrr::future_map_dbl(1:1000, my_boot_fn,
                                       .options = furrr_options(seed =
})
```

```
speedup <- t_serial[3] / t_par[3]
efficiency <- speedup / 4
c(speedup = speedup, efficiency = efficiency)
```

Typical patterns for embarrassingly parallel code:

- Tiny per-iteration work (microseconds): parallelism is net slower because of serialisation overhead.
- Moderate per-iteration work (milliseconds to seconds): near-linear speedup up to about $n_{\text{cores}}$.
- Heavy per-iteration work (minutes): linear speedup up to saturation; possibly diminishing returns beyond.

For the modest per-iteration work typical of bootstrap or simulation, expect 70–90% efficiency on 4–8 cores; lower on 16+ cores due to overhead and contention.

## 18.9. Common pitfalls

**RNG.** Already covered; `future.seed = TRUE` / `furrr_options(seed = TRUE)`.

**Global state.** Workers receive a copy of the workspace needed to run the function. Variables not mentioned in the function body are not copied (saving time and memory), but variables hidden in environments may be missed. The modern `future` API tries to detect dependencies automatically; for unusual cases, use `furrr_options(globals = ...)`.

**BLAS oversubscription.** `RhpcBLASctl::blas_set_num_threads(1)` inside the worker prevents BLAS from spawning additional threads:

```
plan(multisession, workers = 4)

future_map_dbl(1:1000, function(i) {
  RhpcBLASctl::blas_set_num_threads(1)
  RhpcBLASctl::omp_set_num_threads(1)
```

```
  # ... heavy linear algebra computation ...
})
```

**Large data copies.** A 5 GB dataset replicated across 8 workers consumes 40 GB. Three remedies:

1. Load data inside the worker function rather than from the global environment.
2. Use shared-memory backends (e.g., `bigmemory`, `arrow`, `parquet` files read on demand).
3. Reduce the data: workers operate on a summary or a subset.

**Stack overflows from over-parallelisation.** Spawning more processes than cores is rarely useful and often counter-productive. `parallelly::availableCores()` reports the canonical 'how many cores can I use' value, respecting HPC scheduler limits.

## 18.10. Worked example: parallel bootstrap

```
library(future)
library(furrr)
library(palmerpenguins)

plan(multisession, workers = 4)

penguins_clean <- na.omit(penguins)

boot_one <- function(i, data) {
  resampled <- data[sample(nrow(data), replace = TRUE), ]
  mean(resampled$body_mass_g)
}

set.seed(1)
boot_means <- future_map_dbl(
  1:10000,
```

```
  ~ boot_one(.x, penguins_clean),
  .options = furrr_options(seed = TRUE)
)

quantile(boot_means, c(0.025, 0.5, 0.975))
```

For 10,000 bootstrap replicates each taking ~1 ms, this parallelises cleanly across 4 cores: about 4× speedup compared to a serial version.

For the same data and a larger per-iteration cost (e.g., fitting a model on each resample), the speedup approaches $n_{\text{cores}}$ exactly because the per-iteration cost dominates the per-iteration overhead.

## 18.11. When *not* to parallelise

- **Computation already fast.** Code that runs in 5 seconds is not worth the engineering effort.
- **Sequential dependencies dominate.** A single MCMC chain. A reduce/scan that mathematically requires sequential evaluation.
- **Memory pressure.** If the data are large and the per-iteration work is small, copying the data to workers exceeds the work saved.
- **Many small tasks with serialisation overhead.** `future_map_dbl(1:10000, function(i) i^2)` is slower than the serial version because the overhead per task exceeds the work per task.

The rule: parallelise when the per-iteration work is dominant, and the iterations are independent.

## 18.12. Collaborating with an LLM on parallel R

LLMs handle the basic parallel APIs reasonably; the subtle correctness issues (RNG, BLAS, memory) are where they often fall short.

**Prompt 1: parallelising a serial loop.** Paste the serial loop and ask: 'parallelise this with `furrr` and ensure reproducible random numbers.'

*What to watch for.* The LLM should produce code with `furrr_options(seed = TRUE)`. If it does not, push back explicitly. The default behaviour (no seed handling) is a silent correctness bug.

*Verification.* Run twice with the same parent seed and verify identical results. If they differ, the seed handling is wrong.

**Prompt 2: diagnosing poor speedup.** Describe the benchmark (e.g., '4 cores, expected 4× speedup, observed 1.5×') and ask: 'what could be limiting parallel efficiency?'

*What to watch for.* Standard suspects: BLAS oversubscription, large data copies, small per-iteration work, hyperthreading limits. The LLM should mention several. If it fixates on one, push for the full list.

*Verification.* Apply each fix in turn and remeasure. Sometimes the fix is to give up and accept that this particular problem does not parallelise well.

**Prompt 3: choosing a backend.** Describe the platform (macOS laptop, Linux HPC, Windows desktop) and ask: 'which `future` plan should I use?'

*What to watch for.* `multisession` is the safe cross-platform default. `multicore` is faster on Linux/macOS but does not work in RStudio. `cluster` backends require more setup. The LLM should know these trade-offs.

*Verification.* Try the recommended plan and verify it runs on your platform. If it errors with 'not supported on Windows' or similar, switch to `multisession`.

## 18.13. Principle in use

Three habits define defensible parallel computation:

1. **Profile before parallelising.** Don't add parallel overhead to code that is already fast enough.
2. **Always use seed-aware parallel APIs.** `future.seed = TRUE` or `furrr_options(seed = TRUE)`. Without them, results are silently miscalibrated.

3. **Mind the BLAS.** For matrix-heavy parallel code, set per-worker BLAS threads to 1 to prevent oversubscription.

## 18.14. Exercises

1. Run a 10,000-replicate simulation sequentially with `purrr::map()`. Parallelise it with `furrr::future_map()` and a `multisession` plan. Time both and compute the speedup on your machine.
2. Repeat exercise 1 using a `multicore` plan on macOS or Linux. Compare to `multisession`. Explain any difference in speed.
3. Construct an example where parallelism *slows* the computation down (hint: small per-iteration work, large data copying). Document why.
4. Fit a logistic regression on each bootstrap resample in parallel. Use `furrr_options(seed = TRUE)` and verify reproducibility. Confirm BLAS is not oversubscribing your cores.
5. Run a parallel benchmark with 1, 2, 4, 8, and 16 workers (or as many as you have). Plot the speedup vs. workers. Where does efficiency drop sharply?

## 18.15. Further reading

- The `future` ecosystem documentation at `futureverse.org`, the modern parallel-R reference.
- (Bengtsson, 2021), the JSS paper on `future`.
- (Rizzo, 2019) Chapter 16, `parallel` and `foreach` in the context of simulation studies.
- Eddelbuettel (2020), *Extending R*, Chapman and Hall/CRC, Chapter 4, parallel R.

## 18.16. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 18.16.1. Question 1

What is the primary advantage of using parallel computing in R for statistical applications?

- A) It always makes code run faster regardless of the problem size
- B) It allows multiple independent computations to be performed simultaneously, reducing total execution time for suitable problems
- C) It automatically optimises R code without requiring any programming changes
- D) It eliminates the need for writing loops in R

**Answer**

**B.** Parallel computing speeds up problems that decompose into independent tasks; it is not a universal speedup.

### 18.16.2. Question 2

Which type of statistical computation is most suitable for parallel processing?

- A) Sequential iterative algorithms where each step depends on the previous result
- B) Simple arithmetic on small datasets
- C) Embarrassingly parallel problems such as bootstrap resampling where computations are completely independent
- D) Interactive data exploration

**Answer**

**C.** Embarrassingly parallel problems have no serial dependence between tasks and scale almost linearly with the number of workers.

### 18.16.3. Question 3

What is the most important consideration when deciding how many processor cores to use?

- A) Always use the maximum number of cores available
- B) The number of cores should equal the number of observations
- C) Balance overhead against gains; using fewer than maximum is often optimal for system responsiveness and avoids BLAS oversubscription
- D) The number of cores has no effect on performance

> **i** Answer
>
> **C.** Saturating every core incurs overhead and degrades system responsiveness; the optimal worker count balances costs against gains.

### 18.16.4. Question 4

You parallelise a bootstrap with `future_map` but forget `furrr_options(seed = TRUE)`. What is the most likely consequence?

- A) The code will not run.
- B) Workers may use correlated or identical RNG streams, biasing the bootstrap distribution.
- C) The parallel speedup is reduced.
- D) Memory usage increases.

> **i** Answer
>
> **B.** Without explicit parallel-safe RNG, the seed state across workers is undefined or correlated. Results are silently miscalibrated.

### 18.16.5. Question 5

You see only 1.5× speedup on 4 cores for a parallel linear-algebra-heavy computation. The most likely cause is:

- A) RNG misconfiguration.
- B) BLAS oversubscription: each worker spawns multiple BLAS threads, so 4 workers × 4 BLAS threads contend for 4 physical cores.
- C) Insufficient workers.
- D) The serial code is faster than parallel.

**Answer**

**B.** Standard fix: `RhpcBLASctl::blas_set_num_threads(1)` inside each worker. The contention from competing BLAS threads often appears as 'parallel is barely faster than serial' on linear-algebra-heavy code.

## 18.17. Prerequisites answers

1. Parallel computing allows multiple independent computations to run simultaneously on multiple cores or machines, reducing total wall-clock time for suitable problems. It is not a universal speedup; problems with strong serial dependencies cannot benefit.
2. *Embarrassingly parallel* problems, whose work decomposes into fully independent tasks, are most suitable. Bootstrap resampling and simulation replicates are canonical examples. Problems with serial dependence (e.g., a single MCMC chain) are not. Multi-chain MCMC, however, is embarrassingly parallel at the chain level.
3. Each parallel task incurs overhead (worker startup, data serialisation, result collection), so using every core can slow a small computation rather than speed it up. System responsiveness also degrades when every core is saturated. For matrix-heavy code, BLAS oversubscription compounds the problem: $k$ workers each spawning $m$ BLAS threads is $km$ threads competing on (typically) $k$ physical cores.

# 19. R Package Development

## 19.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 19.19.

1. What are the two absolutely essential files or directories required for a minimal functional R package?
2. What is the primary function of the `DESCRIPTION` file?
3. What is the key distinction between packages listed under `Imports` and those listed under `Suggests`?

## 19.2. Learning objectives

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

- Create a skeletal R package with `usethis::create_package()` and add the canonical development tooling (`use_git`, `use_testthat`, `use_readme_rmd`, `use_mit_license`).
- Structure code in `R/`, data in `data/`, and tests in `tests/testthat/`.
- Document a function with `roxygen2` tags and generate `.Rd` files with `devtools::document()`.
- Manage dependencies in `DESCRIPTION` using `Imports`, `Suggests`, and `Depends`.
- Manage exports through the auto-generated `NAMESPACE`, with `@export` and `@importFrom` roxygen tags.
- Install a package from source, from a local directory, and from GitHub.
- Run `devtools::check()` and resolve common warnings.

## 19.3. Orientation

Everything you have written so far has been a script. A script is a set of instructions that runs top-to-bottom and ends. A *package* is a reusable library of functions with documentation, tests, and a dependency declaration. Packages are how analytical work scales beyond a single analyst and a single project.

The Wickham and Bryan book *R Packages* (2nd ed.) and the `usethis` and `devtools` packages it documents are the modern foundation for R package development. The workflow they describe has become the standard, and this chapter follows it.

This chapter covers the workflow up to a buildable, installable, documented package. The next chapter (testing) covers the testing infrastructure that makes packages maintainable.

## 19.4. The statistician's contribution

Package development looks like software engineering, and it is. The judgements specific to statistical packages are about audience and scope.

**Who will use this?** A package for your own analysis needs little more than working code and a `DESCRIPTION`. A package for your research group needs documentation and tests. A package for the public CRAN repository needs clean dependencies, vignettes, and continuous integration. Pick the audience first; the engineering follows.

**What goes in the package, and what stays in the analysis?** A function reused across two projects belongs in a package; one used once does not. A regression diagnostic that comes up in every analysis is a candidate; a one-off plot for one paper is not. The boundary is fuzzy; the heuristic is that anything you would otherwise copy-paste between projects is a packaging candidate.

**Documentation as a contract.** The roxygen comment above a function is a promise about its behaviour: what arguments it takes, what it returns, what conditions it expects. Vague documentation produces confused users and bug reports about expected behaviour. Specific documentation —

including the edge cases the function does or does not handle, saves time over the package's life.

**Dependencies are a tax.** Each package you `Import` becomes a thing your users must install. Each version restriction (`dplyr (>= 1.0.0)`) constrains the environments your package will work in. The minimum-viable dependency list saves friction; the maximalist 'use whatever is convenient' approach produces packages that are hard to install five years later.

These judgements are what distinguishes packages people use from packages that exist. Software craft does the rest.

## 19.5. Why package code?

Concrete benefits:

- **Reuse.** A function in a package is loaded with `library(yourpkg)`; a function in a script is sourced by hand or copy-pasted.
- **Documentation.** Roxygen comments produce help pages accessible via `?function_name`, indistinguishable from base R or CRAN package help.
- **Tests.** Packages have a canonical home for tests (`tests/testthat/`); scripts do not.
- **Dependency management.** `DESCRIPTION` declares what the package needs; users install dependencies automatically.
- **Distribution.** A `.tar.gz` build can be installed by others. CRAN, GitHub, and internal package servers all use this format.

The cost: more files, more conventions, more tooling. The threshold for going from script to package is roughly 'this code will be reused by me or someone else more than once'.

## 19.6. `usethis::create_package()` and first commit

```
library(usethis)

# create the skeleton in a new directory
create_package("~/research/phb228utils")
```

This creates:

- `DESCRIPTION` (metadata)
- `NAMESPACE` (export declarations; auto-generated)
- `R/` (source code directory; empty)
- `phb228utils.Rproj` (RStudio project file)
- `.Rbuildignore` (files to exclude from package build)
- `.gitignore` (files to exclude from version control)

Then add the standard scaffolding:

```
use_git()                    # initialise git repository
use_github()                 # create a GitHub repository
use_mit_license()            # add an MIT licence
use_readme_rmd()             # README that knits to README.md
use_testthat()               # set up the testing framework
use_news_md()                # NEWS.md for changelog
```

Each of these adds files in canonical locations and updates `DESCRIPTION` and `.Rbuildignore` as needed. Doing all of this manually is tedious and error-prone; `usethis` encodes the conventions.

## 19.7. Package structure

```
phb228utils/
├── DESCRIPTION         # metadata
├── NAMESPACE           # auto-generated by roxygen2
├── R/                  # source code
│   ├── summarise.R
│   ├── plot.R
│   └── package.R       # package-level documentation
```

```
├── man/                # auto-generated by roxygen2
│   └── *.Rd
├── tests/
│   ├── testthat.R
│   └── testthat/
│       └── test-summarise.R
├── vignettes/
│   └── intro.Rmd       # long-form documentation (optional)
├── data/               # binary R data files (.rda)
├── data-raw/           # scripts that create data/*.rda (not shippe
├── inst/               # other files shipped with the package
├── LICENSE
├── README.md
├── NEWS.md
└── phb228utils.Rproj
```

What goes where:

- **`R/`**: source code. One function per file is a common convention (`R/summarise.R` for `summarise()`). Files are loaded in alphabetical order; if some functions depend on others, use `@include` roxygen tags or merge into one file.
- **`man/`**: auto-generated `.Rd` documentation files. Never edit these by hand; edit the roxygen comments and re-run `devtools::document()`.
- **`tests/testthat/`**: test files (covered in Chapter 20).
- **`data/`**: example datasets shipped with the package, loaded with `data(my_data)`.
- **`vignettes/`**: long-form articles built with the package and accessible via `vignette("intro", "phb228utils")`.

## 19.8. roxygen2 documentation

Documentation lives in comments above each function:

```
#' Summarise a numeric vector
#'
#' Produces a tibble with the mean, standard deviation, and
```

```
#' quartiles of a numeric vector, ignoring missing values.
#'
#' @param x A numeric vector.
#' @param probs Quantile probabilities to report. Defaults
#'   to `c(0.25, 0.5, 0.75)`.
#' @return A tibble with one row and columns `mean`, `sd`,
#'   and one column per requested quantile.
#' @export
#' @examples
#' summarise_numeric(rnorm(100))
#' summarise_numeric(rnorm(100), probs = c(0.05, 0.5, 0.95))
summarise_numeric <- function(x, probs = c(0.25, 0.5, 0.75)) {
  stopifnot(is.numeric(x))
  q <- quantile(x, probs = probs, na.rm = TRUE)
  tibble::tibble(
    mean = mean(x, na.rm = TRUE),
    sd   = sd(x, na.rm = TRUE),
    !!!setNames(as.list(q), paste0("q", probs * 100))
  )
}
```

Tags:

- `@param name description`: each argument.
- `@return description`: what the function returns.
- `@export`: the function is part of the package's public API.
- `@examples`: runnable examples; checked by `R CMD check`.
- `@importFrom pkg fn`: import a specific function from another package.
- `@seealso`: links to related functions.
- `@inheritParams other_function`: copy parameter docs from another function.

After editing roxygen comments, regenerate the `.Rd` files and `NAMESPACE`:

```
devtools::document()
```

This is one of the most common commands in package development; bind it to a keystroke.

Check your understanding: `@export` vs not

**Question.** A function in `R/` does not have `@export` in its roxygen header. What does this mean for users of the package?
**Answer.**
The function is *internal*: not part of the public API. Users who load the package with `library(yourpkg)` cannot call it directly. They can still access it via the triple- colon operator (`yourpkg:::internal_fn(...)`), but doing so is discouraged because internal functions can change without notice between versions. Internal functions are useful for helpers shared among exported functions without polluting the namespace. Reserve `@export` for the functions you want users to call; everything else remains internal.

## 19.9. DESCRIPTION: Imports, Suggests, Depends

A typical `DESCRIPTION`:

```
Package: phb228utils
Title: Helper Functions for Statistical Computing
Version: 0.1.0
Authors@R:
    person("Ronald", "Thomas", email = "rgthomas47@gmail.com",
           role = c("aut", "cre"))
Description: Reusable helpers for the PHB 228 statistical
    computing course textbook.
License: MIT + file LICENSE
Encoding: UTF-8
LazyData: true
Imports:
    dplyr (>= 1.0.0),
    tibble,
    rlang
```

```
Suggests:
    testthat (>= 3.0.0),
    knitr,
    rmarkdown
RoxygenNote: 7.3.2
```

The dependency fields:

- **`Imports`** are mandatory. Functions you call in your exported code go here. `library(yourpkg)` will fail if these are not installed. Use `pkg::fn()` to call imported functions explicitly (best practice) or `@importFrom pkg fn` to make them available unqualified.
- **`Suggests`** are optional. Used in vignettes, tests, examples, or by features that gate themselves on `requireNamespace("pkg", quietly = TRUE)`. The package must work without these.
- **`Depends`** is for packages that should be loaded whenever yours is loaded (so `library(yourpkg)` also loads them). Avoid: it pollutes the user's namespace. Use `Imports` instead.

Add a dependency with `usethis::use_package()`:

```
use_package("dplyr")          # adds to Imports
use_package("testthat", "Suggests")
```

Version requirements: use minimum versions for features you need, not exact pins. `dplyr (>= 1.0.0)` means 'any version 1.0 or later'. Avoid `dplyr (== 1.0.5)`, this constrains users to one specific version, which usually breaks within a year.

## 19.10. Installing and loading

During development:

```
devtools::load_all()         # simulate library(yourpkg) without instal
```

`load_all()` makes your package's functions available in the current R session, including non-exported ones (so you can call internal functions for testing). It is the fastest way to iterate on changes.

To install for real use:

```
devtools::install()          # build and install in user's library
devtools::build()            # build a .tar.gz for distribution
```

For a package on GitHub:

```
remotes::install_github("yourname/yourpkg")
```

Or from a local source directory:

```
install.packages("/path/to/yourpkg.tar.gz", repos = NULL,
                 type = "source")
```

The standard checks before submission to anyone (CRAN, a collaborator, your future self):

```
devtools::check()
```

This runs `R CMD check`, the gold standard for package quality. It tests:

- Documentation is complete and consistent.
- Examples run without errors.
- Tests pass.
- Dependencies are declared correctly.
- No undocumented functions.
- The package builds and loads on a clean R session.

A clean `check()` (no errors, warnings, or notes) is the goal for any shareable package.

## 19.11. Common `R CMD check` warnings

**'no visible binding for global variable'** when you use column names with non-standard evaluation (NSE) inside functions. Common in dplyr code: `dplyr::filter(data, year == 2020)` references `year` without quoting. The fix:

```
utils::globalVariables(c("year", "treatment"))
```

at the top of one of your `R/` files, or use `.data$year` and `.data$treatment` (preferred).

**'undefined exports'** when an `@export`ed function does not exist. Re-run `devtools::document()`.

**'package required but not declared'** when you use `pkg::fn()` for a package not in `Imports`. Add it.

**'examples lines wider than 100 characters'** when an example line is too long. Break it up.

The fixes are straightforward; the work is doing the fixes consistently.

## 19.12. Vignettes

A vignette is a long-form article packaged with your code:

```
use_vignette("intro")
```

This creates `vignettes/intro.Rmd` with a template. Edit it; build with `devtools::build_vignettes()`; access in R with `vignette("intro", "phb228utils")`.

Vignettes are how to teach users *why* and *how to use* your package, beyond the function-by-function reference. For an analysis package, the vignette is often a worked example.

## 19.13. Worked example: a small package

```
# 1. create the package
usethis::create_package("~/research/phb228utils")

# 2. add tooling (run from inside the new package)
usethis::use_git()
usethis::use_mit_license()
usethis::use_testthat()
usethis::use_readme_rmd()

# 3. add a function
# in R/summarise.R:
#   roxygen header above summarise_numeric()
usethis::use_package("tibble")

# 4. document and check
devtools::document()
devtools::load_all()
?summarise_numeric

# 5. add a test
usethis::use_test("summarise")
# write the test, run:
devtools::test()

# 6. install
devtools::install()
```

This sequence creates a working, documented, tested, installable package in about thirty minutes of focused work.

## 19.14. Collaborating with an LLM on package development

Package development has a lot of conventions; LLMs handle most of them reasonably and stumble on a few specific ones.

**Prompt 1: drafting a roxygen header.** Paste the function and ask: 'write a roxygen2 header with (**param?**), (**return?**), (**export?**), and (**examples?**). The example should be a realistic, runnable use of the function.'

*What to watch for.* The example needs to actually run. A common LLM error: the example uses a variable that is not defined. Run the example yourself before committing.

*Verification.* `devtools::document()` then run the example via `?function_name`. If the example fails, fix it.

**Prompt 2: diagnosing an `R CMD check` warning.** Paste the warning verbatim and ask: 'what does this mean and how do I fix it?'

*What to watch for.* The standard warnings (global variables, undocumented arguments, missing imports) have known fixes; LLMs handle them well. Less common warnings (invalid CITATION format, vignette engine issues) get mixed answers; verify against the R packages book or the CRAN policies.

*Verification.* Apply the fix and re-run `check()`. If the warning persists, look up the message in the R packages book.

**Prompt 3: deciding `Imports` vs `Suggests`.** Describe how you use a dependency (e.g., 'I call `ggplot2::ggplot` inside one of my exported functions; I also use it in a vignette'). Ask: 'should this be in Imports or Suggests?'

*What to watch for.* The rule is 'Imports if exported code uses it'. Vignette-only or test-only dependencies go in Suggests. The LLM should know this; if it hesitates, push for the rule.

*Verification.* Try installing the package on a fresh R session without the dependency installed. If `library(yourpkg)` fails, the dependency belongs in Imports.

## 19.15. Principle in use

Three habits define defensible package development:

1. **Use `usethis` for scaffolding.** Hand-creating the package skeleton is error-prone and wastes time. `usethis` encodes the canonical layout and conventions.
2. **Document every exported function.** A roxygen header with `@param`, `@return`, and a runnable `@examples` is the contract with users.
3. **Aim for a clean `R CMD check`.** No errors, warnings, or notes. The output of `check()` is the first thing CRAN reviewers (and any thoughtful collaborator) look at.

## 19.16. Exercises

1. Create a package `phb228utils` with a single function `summarise_numeric()` from chapter 1. Document it, add an example, and confirm that `?summarise_numeric` works after `devtools::document()` and `devtools::load_all()`.
2. Add a dependency on `dplyr` to `phb228utils`. Decide whether it belongs in `Imports` or `Suggests` and justify your choice.
3. Build the package as a `.tar.gz` with `devtools::build()` and install it on a clean R session. Verify it works.
4. Run `devtools::check()` on the package. Fix every warning and note until the output is clean.
5. Write a vignette demonstrating `summarise_numeric()` on a real dataset. Build the vignette and access it via `vignette()`.

## 19.17. Further reading

- (Wickham & Bryan, 2023), *R Packages*, 2nd ed., the canonical reference. Free at r-pkgs.org. Tracks the modern `usethis`/`devtools` workflow.
- The `usethis` and `devtools` package documentation.

- *Writing R Extensions* (the official R-core manual) for the technical reference; usually a last resort, but authoritative when conflicts arise.

## 19.18. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 19.18.1. Question 1

What are the two absolutely essential files/directories required for a minimal functional R package?

- A) `DESCRIPTION` file and `man/` directory
- B) `DESCRIPTION` file and `R/` directory
- C) `NAMESPACE` file and `R/` directory
- D) `R/` directory and `tests/` directory

> **Answer**
>
> **B.** `DESCRIPTION` provides metadata; `R/` contains source code. All other components (`NAMESPACE`, `man/`, `tests/`) are either auto-generated or optional.

### 19.18.2. Question 2

What is the primary function of the `DESCRIPTION` file?

- A) To contain the actual R function code
- B) To store example datasets
- C) To provide essential package metadata including dependencies, author information, and version numbers
- D) To automatically generate help documentation

Answer

**C.** `DESCRIPTION` stores package metadata including `Imports`, `Suggests`, `Depends`, `Author`, `Version`, and `License`.

### 19.18.3. Question 3

What is the key distinction between packages listed under `Imports` versus `Suggests`?

- A) `Imports` lists packages required for core functionality; `Suggests` lists optional packages for enhanced features
- B) `Imports` refers to newer packages; `Suggests` refers to older, deprecated ones
- C) `Imports` indicates packages from CRAN; `Suggests` indicates packages from GitHub
- D) There is no meaningful difference

Answer

**A.** `Imports` packages are mandatory dependencies; `Suggests` packages are used only for optional features (vignettes, tests, examples) and need not be installed by default.

### 19.18.4. Question 4

You add `@export` above one function and not above another. What does this mean?

- A) Only the `@export`ed function is part of the package's public API; the other is internal.
- B) Both are public; `@export` is a stylistic choice.
- C) The non-exported function is broken.
- D) The exported function is loaded faster.

Answer

**A.** Internal functions are accessible only via `pkg:::fn()` and may change without notice; exported functions are the package's stable public API.

### 19.18.5. Question 5

After modifying a roxygen header above a function, you should next:

- A) Edit `man/function.Rd` directly to match.
- B) Run `devtools::document()` to regenerate the `.Rd` file and `NAMESPACE`.
- C) Manually update `DESCRIPTION`.
- D) Restart R.

Answer

**B.** `devtools::document()` regenerates documentation and the namespace from roxygen comments. Hand-editing `.Rd` files is wrong; they will be overwritten.

## 19.19. Prerequisites answers

1. The `DESCRIPTION` file (metadata) and the `R/` directory (source code). Everything else (`NAMESPACE`, `man/`, `tests/`, `data/`, vignettes) is either auto-generated or optional. The minimum viable package is one function in `R/foo.R` plus a one-line `DESCRIPTION`.
2. `DESCRIPTION` stores package metadata: name, version, title, description, author, license, and, crucially, dependencies (`Imports`, `Suggests`, `Depends`). It is the file that distinguishes a package from a directory of R scripts.
3. `Imports` lists packages required for core functionality (they must be installed for the package to work). `Suggests` lists packages used only for optional features (vignettes, tests, examples) and need not

be installed by default. The rule of thumb: if your exported code calls `pkg::fn()`, the package goes in `Imports`. If only your tests, vignettes, or optional features use it, `Suggests`.

# 20. Package Testing and Documentation

## 20.1. Prerequisites

Answer the following questions to see if you can bypass this chapter. You can find the answers at the end of the chapter in Section 20.18.

1. What is the difference between `expect_equal()` and `expect_identical()` in `testthat`?
2. What is a snapshot test, and in what situation is it preferable to a direct value-comparison test?
3. Name one check that `R CMD check` performs that `devtools::test()` does not.

## 20.2. Learning objectives

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

- Write unit tests with `testthat` (3rd edition) and organise them in `tests/testthat/`.
- Use test expectations (`expect_equal`, `expect_error`, `expect_warning`, `expect_message`, `expect_snapshot`) appropriately.
- Choose between value-comparison and snapshot tests.
- Measure test coverage with `covr` and identify gaps.
- Produce a package vignette as a Quarto or R Markdown document.
- Run `R CMD check` and interpret its output.
- Set up continuous integration via GitHub Actions with `usethis::use_github_a`

## 20.3. Orientation

Tests are how you convince a future reader (including future-you) that your code does what you think it does. `R CMD check` is how you convince CRAN, your collaborators, and your build pipeline. Vignettes are how you convince a human to use your package at all.

This chapter is the testing companion to the package- development chapter. It treats testing as a first-class activity rather than something to add at the end. A package without tests is a research script with extra ceremony; a package with tests is a tool that other people can rely on.

The canonical R testing framework is `testthat` (currently version 3, accessed via `usethis::use_testthat(3)`). It is the standard, used by all of tidyverse, most of CRAN, and nearly every reproducible-research package.

## 20.4. The statistician's contribution

Testing is software engineering, but the testing priorities for statistical code are specific.

**Test the math, not just the syntax.** A test that verifies your function returns a tibble of the right shape is not the test you need; it is a structural check. The test you need is whether the function returns the right *answer* on a known case. Compare to a closed-form solution where one exists, to a reference implementation otherwise.

**Test edge cases that change the math.** Empty input. All NA. Single observation. Floating-point near-machine-zero. Inputs that exercise type coercion (logical to numeric, integer to double). These are where statistical functions break, and they break silently, the function returns something, just not the right thing.

**Snapshot tests for plots and printed output.** Direct equality fails for plots (rendering changes between ggplot2 versions) and for nicely-formatted print output (spacing varies). `expect_snapshot()` records a reference output and compares against it. Useful for things that are either too complex or too aesthetic to assert literally.

**Coverage is a floor, not a ceiling.** 90% line coverage sounds good but does not guarantee the lines were tested *correctly*. A test that runs every line but checks nothing is worse than no test, because it produces a false sense of security. Track coverage; aim for high numbers; but never substitute it for actual thought about what could go wrong.

**Tests are the spec.** When you cannot tell whether a piece of behaviour is a bug or a feature, the existence or absence of a test is the answer. A function that returns NA on an edge case has a test for that behaviour: it is intended. A function that crashes on the same input has no test: it is a bug. Tests document intentions in a machine-checkable form.

These judgements are what make tests useful rather than performative.

## 20.5. Writing tests with testthat

A test file lives in `tests/testthat/test-name.R`. Name your test files after the function they test: `R/summarise.R` → `tests/testthat/test-summarise.R`.

```
# tests/testthat/test-summarise.R
test_that("summarise_numeric works on a known input", {
  result <- summarise_numeric(1:10)
  expect_equal(result$mean, 5.5)
  expect_equal(result$sd,   sd(1:10))
  expect_equal(nrow(result), 1)
})

test_that("summarise_numeric handles NA correctly", {
  x <- c(1, 2, NA, 4, 5)
  result <- summarise_numeric(x)
  expect_equal(result$mean, mean(x, na.rm = TRUE))
  expect_false(is.na(result$mean))
})

test_that("summarise_numeric errors on non-numeric input", {
  expect_error(summarise_numeric("a"))
  expect_error(summarise_numeric(c(TRUE, FALSE)))
```

```
})

test_that("summarise_numeric handles empty input", {
  expect_warning(result <- summarise_numeric(numeric(0)))
  # what should it return on empty input? document and test that decisi
})
```

Each `test_that()` block is one test, with a human-readable name. Inside, one or more `expect_*` calls make assertions. If any fail, the test fails; if none fail, the test passes.

To run all tests:

```
devtools::test()
```

To run one file:

```
testthat::test_file("tests/testthat/test-summarise.R")
```

In RStudio, the keyboard shortcut Cmd-Shift-T (Mac) or Ctrl-Shift-T (Windows/Linux) runs the full test suite for the current package.

## 20.6. Expectations

The full library of `expect_*` functions:

```
# value comparisons
expect_equal(actual, expected)         # all.equal: numeric tolerance
expect_identical(actual, expected)     # identical(): bit-exact
expect_lt(actual, expected)            # less-than
expect_gt(actual, expected)
expect_lte(actual, expected)
expect_gte(actual, expected)
expect_true(actual)
expect_false(actual)
```

```
# class and type
expect_s3_class(actual, "lm")
expect_type(actual, "double")
expect_length(actual, 5)
expect_named(actual, c("x", "y", "z"))

# conditions
expect_error(expr, regexp = "must be numeric")
expect_warning(expr)
expect_message(expr)
expect_silent(expr)               # no message/warning/error
expect_no_error(expr)             # explicitly succeeds without erro
expect_no_warning(expr)

# snapshots
expect_snapshot(print(fit))       # captures printed output
expect_snapshot_value(complex_obj, style = "json2")
expect_snapshot_file(path)        # for non-text snapshot files
```

`expect_equal` uses `all.equal()` semantics: numeric values are compared with a small tolerance for floating-point round-off. `expect_identical` uses `identical()`: exact equality including types, attributes, and structure. For numeric tests, `expect_equal` is almost always the right choice; `expect_identical` for assertions about object structure (class, names, length, attributes).

## 20.7. Snapshot tests

Snapshots are useful for output that is complex or aesthetic:

```
test_that("summary prints correctly", {
  fit <- lm(mpg ~ wt, data = mtcars)
  expect_snapshot(summary(fit))
})
```

On the first run, the printed output is captured to a file in `tests/testthat/_snaps/`. On subsequent runs, the output is compared against the saved snapshot. If they differ, the test fails and you are prompted to either accept the new output (`testthat::snapshot_accept()`) or investigate the difference.

For plots:

```
test_that("plot looks the same", {
  expect_snapshot_file("path/to/plot.png", "regression_plot")
})

# or with vdiffr
test_that("regression plot is unchanged", {
  vdiffr::expect_doppelganger("regression-fit", create_my_plot())
})
```

`vdiffr` is a separate package specifically for plot snapshot tests; it handles cross-platform rendering issues better than raw image comparison.

When to use snapshots vs. value comparisons:

- **Value comparison** when the answer is a small, literal value (`expect_equal(result, 42)`). Easy to understand at the test site, no separate file to manage.
- **Snapshot** when the output is large, formatted, or not easily expressible as a literal. Trade-off: the test asserts 'output is unchanged', not 'output is correct'.

Check your understanding: equal vs. identical

**Question.** You write `expect_equal(my_function(1:10), c(1, 4, 9, 16, 25, 36, 49, 64, 81, 100))`. The test passes. What if you used `expect_identical`?

**Answer.**

`expect_identical` is stricter: it requires bit-exact equality including types. `1:10` is an integer vector; `c(1, 4, 9, ...)` is a double vector (because the values are written as decimals). `my_function(1:10)`'s output type depends on the implementation. If your function re-

turns `c(1, 4, 9, ..., 100)` (doubles), the `expect_identical` test passes. If it returns `(1:10)^2` (integers, since integer^integer is integer in R), `expect_identical` fails because integer ≠ double, even though the values match. `expect_equal` ignores this distinction. The lesson: use `expect_identical` only when you genuinely care about the type as well as the value.

## 20.8. Coverage with covr

`covr` measures which lines of your package are exercised by your test suite:

```
library(covr)

# package-level coverage
cov <- package_coverage()
cov

# detailed report in a browser
report(cov)
```

Lines marked red are uncovered: tests do not exercise them. Lines marked green are covered.

Coverage thresholds vary by project. Tidyverse packages target 90%+ on average; a hard floor of 80% is reasonable for new packages. Some lines (defensive `stopifnot`s, error handlers for impossible-in-practice cases) are reasonably left uncovered.

`use_github_action_test_coverage()` sets up automated coverage reporting via Codecov on GitHub Actions, with a badge for your README.

## 20.9. R CMD check

`R CMD check` is the gold standard for package quality. It runs:

- **Documentation completeness.** Every exported function has documentation; every documented parameter exists.
- **Examples.** Every example runs without error.
- **Tests.** All tests pass.
- **Imports/Suggests.** Declared dependencies are used; used dependencies are declared.
- **Namespace.** Imports/exports in `NAMESPACE` are consistent with roxygen tags.
- **CRAN policies.** A subset of CRAN's submission rules (no use of `library()` in package code, no `assignInNamespace`, etc.).

```
devtools::check()
```

Output is one of:

- **OK.** No errors, warnings, or notes. Submittable.
- **NOTE.** Minor issues; usually acceptable for a private package, sometimes acceptable for CRAN submission.
- **WARNING.** Serious; almost always must be fixed.
- **ERROR.** The package does not build or pass; must be fixed.

Run `check()` before every commit if possible, and at minimum before every release. The longer you let warnings accumulate, the harder they are to dismantle.

## 20.10. Vignettes

Vignettes are long-form articles bundled with the package:

```
usethis::use_vignette("intro")
```

This creates `vignettes/intro.Rmd` with a YAML header. Edit it; build with `devtools::build_vignettes()`. Users access it via `vignette("intro", package = "yourpkg")`.

Format options:

- **R Markdown** (`.Rmd`): the traditional choice. Converted to HTML by default, optionally PDF.
- **Quarto** (`.qmd`): the modern alternative. More flexible, but requires Quarto installed on the user's machine to build from source.

For a CRAN-bound package, R Markdown is the safer choice because the build dependencies are universal. For internal packages, either works.

A good vignette:

- Introduces the problem the package solves.
- Walks through a worked example end to end.
- Highlights the key functions and how they fit together.
- Is short enough to read in 10–15 minutes.

A vignette is *not* a function-by-function reference; the help pages serve that purpose.

## 20.11. Continuous integration with GitHub Actions

`usethis::use_github_action_check_standard()` sets up a GitHub Actions workflow that runs `R CMD check` on every push and pull request, on multiple OS/R-version combinations:

```
usethis::use_github_action("check-standard")
```

This adds `.github/workflows/R-CMD-check.yaml`. On every push, GitHub runs `check()` on three platforms (macOS, Windows, Ubuntu) with the latest stable R. If anything fails, the commit is marked failed.

For test coverage:

```
usethis::use_github_action("test-coverage")
```

This runs `covr::package_coverage()` and uploads the results to Codecov, where you can see line-by-line coverage and how it has changed over time.

Setting up CI is a one-time investment that pays off for the life of the package. Every change is automatically checked; regressions are caught before merge.

## 20.12. Worked example: testing `summarise_numeric()`

```
# tests/testthat/test-summarise_numeric.R

test_that("returns correct mean and sd on a simple vector", {
  result <- summarise_numeric(1:10)
  expect_equal(result$mean, 5.5)
  expect_equal(result$sd, sd(1:10))
})

test_that("returns a 1-row tibble", {
  result <- summarise_numeric(rnorm(50))
  expect_s3_class(result, "tbl_df")
  expect_equal(nrow(result), 1)
})

test_that("ignores NAs", {
  x <- c(1, 2, 3, NA, 5)
  result <- summarise_numeric(x)
  expect_equal(result$mean, mean(x, na.rm = TRUE))
  expect_equal(result$sd,   sd(x, na.rm = TRUE))
})

test_that("respects custom probs argument", {
  result <- summarise_numeric(1:100, probs = c(0.1, 0.9))
  expect_named(result, c("mean", "sd", "q10", "q90"))
})

test_that("errors on non-numeric input", {
  expect_error(summarise_numeric("a"))
  expect_error(summarise_numeric(list(1, 2, 3)))
})

test_that("handles all-NA input gracefully", {
  result <- summarise_numeric(rep(NA_real_, 5))
```

```
  expect_true(is.na(result$mean))
  expect_true(is.na(result$sd))
})

test_that("printed output is stable", {
  expect_snapshot(print(summarise_numeric(1:10)))
})
```

This suite tests:

- Happy path (correct values on a known input).
- Structure (1-row tibble, correct column names).
- Edge cases (NAs, all-NA input, non-numeric input).
- Stability of printed output (snapshot).

`devtools::test()` runs all of these in seconds.

## 20.13. Collaborating with an LLM on testing

LLMs draft tests well; the judgement about which tests to write is harder for them.

**Prompt 1: drafting tests.** Paste the function and ask: 'write `testthat` tests covering happy path, edge cases, and error handling. The tests should be specific assertions, not just shape checks.'

*What to watch for.* The default LLM tests tend to be shape-checks ('returns a tibble', 'has 5 columns'). Push for value-checks: 'the mean is 5.5 on input 1:10'. Edge cases: empty input, NA input, type mismatch input.

*Verification.* The most useful test of test quality is to introduce a bug in the function and re-run the tests. If the tests catch the bug, they are useful. If not, they need to be more specific.

**Prompt 2: snapshot tests.** Ask: 'when should I use snapshot tests, and when should I use direct value comparison?'

*What to watch for.* Standard answer: snapshots for complex/aesthetic output, value comparison for literal values. The LLM should know this. If

it suggests snapshots for everything, push for selectivity: snapshots that change frequently are noise.

*Verification.* For each snapshot test, ask: 'what would this catch?' If the answer is 'it catches changes in the output, including refactors that don't change behaviour', the snapshot may produce false positives.

**Prompt 3: diagnosing a `check()` warning.** Paste the warning verbatim and ask: 'how do I fix this?'

*What to watch for.* Standard fixes for standard warnings. If the LLM suggests workarounds rather than fixes (e.g., 'just add it to .Rbuildignore'), push for the proper fix.

*Verification.* Apply the suggested fix and re-run `check()`. The warning should be gone.

## 20.14. Principle in use

Three habits define defensible testing practice:

1. **Test the math.** Shape checks are useful; value checks against known cases are essential. Tests of structure alone produce false confidence.
2. **Test edge cases deliberately.** Empty input, NA, single observation, type mismatch. These are where silent failures hide.
3. **Use CI.** Tests that run only on your machine catch only your bugs. Tests that run on every push, on multiple OS/R-version combinations, catch the bugs that affect your users.

## 20.15. Exercises

1. Add a `tests/testthat/test-summarise_numeric.R` file to the package from chapter 19 with at least three tests: happy path, empty input, and all-NA input. Run `devtools::test()` and make them pass.
2. Run `covr::package_coverage()` on your package. Identify the uncovered lines and write tests until coverage is above 90%.

3. Set up GitHub Actions via `usethis::use_github_action_check_standard()` and push. Verify that the workflow runs green on GitHub.
4. Introduce a bug into `summarise_numeric()` (e.g., compute SD without `na.rm = TRUE`). Run the test suite. Does it catch the bug? If not, add a test that does.
5. Add a snapshot test for the printed output of one of your functions. Run the tests. Then make a small formatting change to the function and re-run. Does the snapshot fail? Accept or reject the change.

## 20.16. Further reading

- (Wickham & Bryan, 2023) testing chapters, the canonical `testthat` reference.
- (Wickham, 2019) testing chapter, discusses tests in the broader context of robust R programming.
- The `testthat` and `covr` package documentation.

## 20.17. Practice test

The following multiple-choice questions exercise the chapter's content. Attempt each question before expanding the answer.

### 20.17.1. Question 1

What is the difference between `expect_equal()` and `expect_identical()` in `testthat`?

- A) They are aliases for each other.
- B) `expect_equal()` uses numeric tolerance (`all.equal` semantics); `expect_identical()` requires bit-exact equality including types and attributes.
- C) `expect_equal()` checks length; `expect_identical()` checks values.
- D) `expect_identical()` works only on vectors.

Answer

**B.** Use `expect_equal()` for most numeric comparisons; `expect_identical()` when you specifically care about type and attribute exactness.

### 20.17.2. Question 2

What is a snapshot test, and when is it preferable to a direct value comparison?

- A) A test of the function's source code; preferable when the source changes frequently.
- B) A captured reference output that the test compares against on subsequent runs; preferable when the output is complex, formatted, or aesthetic.
- C) A backup of the test database.
- D) A test that runs in a sandbox.

Answer

**B.** Snapshots capture complex output (formatted text, plots) once and compare on later runs. Use them when output is too complex to assert literally.

### 20.17.3. Question 3

Which of the following is a check that `R CMD check` performs that `devtools::test()` does not?

- A) Running unit tests.
- B) Verifying that every exported function is documented and that examples run.
- C) Running benchmarks.
- D) Generating coverage reports.

Answer

**B.** `check()` runs the test suite *plus* documentation checks, example execution, namespace consistency, and CRAN policies. `test()` only runs the test suite.

### 20.17.4. Question 4

You add a new feature and `devtools::test()` reports all tests pass. Should you trust the package is working?

- A) Yes; passing tests guarantee correctness.
- B) Mostly: passing tests are a good sign, but `R CMD check` may still flag documentation, namespace, or example issues.
- C) No; tests are useless.
- D) Only if coverage is exactly 100%.

Answer

**B.** Always run `check()` after `test()` before shipping. Test passes do not guarantee documentation is complete or examples run.

### 20.17.5. Question 5

`covr::package_coverage()` reports 92% line coverage. Should you stop adding tests?

- A) Yes, 90%+ is the goal.
- B) No: coverage measures lines hit, not whether they were tested correctly. The 8% uncovered may include important edge cases; the 92% covered may include shape-only checks. Treat coverage as a floor, not a ceiling.
- C) Run more tests until coverage is exactly 100%.
- D) Coverage is irrelevant.

Answer

**B.** Coverage is necessary but not sufficient. Treat it as a floor and continue thinking about what could go wrong.

## 20.18. Prerequisites answers

1. `expect_equal()` uses `all.equal()` semantics: numerical closeness within a small tolerance, with automatic handling of floating-point round-off. `expect_identical()` uses `identical()`: exact bit- level equality, including types and attributes. For most numeric tests, `expect_equal` is the right choice; for assertions about object structure, use `expect_identical`.
2. A snapshot test captures an expected output (printed text, a complex data structure, or a plot) to a file on first run, then compares against that saved snapshot on subsequent runs. Use it when the output is complex, plot-like, or not easily expressed as a literal value. Trade-off: the test asserts 'output is unchanged', not 'output is correct'.
3. `R CMD check` runs static analysis: it checks `DESCRIPTION` syntax, `NAMESPACE` consistency, undocumented functions, Rd cross-references, that examples run, and that the package installs cleanly from source. `devtools::test()` only runs the `tests/testthat/` suite.

# References


Barr, D. J., Levy, R., Scheepers, C., & Tily, H. J. (2013). Random effects structure for confirmatory hypothesis testing: Keep it maximal. *Journal of Memory and Language*, *68*(3), 255–278.

Bates, D., Mächler, M., Bolker, B. M., & Walker, S. C. (2015). Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, *67*(1), 1–48. https://doi.org/10.18637/jss.v067.i01

Bengtsson, H. (2021). A unifying framework for parallel and distributed processing in R using futures. *The R Journal*, *13*(2), 208–227.

Blischak, J. D., Davenport, E. R., & Wilson, G. (2016). A quick introduction to version control with Git and GitHub. *PLOS Computational Biology*, *12*(1), e1004668. https://doi.org/10.1371/journal.pcbi.1004668

Bolker, B. M. (n.d.). *GLMM FAQ: Mixed models in R.* Online reference. Retrieved https://bbolker.github.io/mixedmodels-misc/glmmFAQ.html

Boyd, S., & Vandenberghe, L. (2004). *Convex optimization.* Cambridge University Press. https://web.stanford.edu/~boyd/cvxbook/

Brown, V. A. (2021). An introduction to linear mixed-effects modeling in R. *Advances in Methods and Practices in Psychological Science*, *4*(1), 1–19. https://doi.org/10.1177/2515245920960351

Bryan, J. (2019). *Happy git and GitHub for the useR.* https://happygitwithr.com

Chacon, S., & Straub, B. (2014). *Pro Git* (2nd ed.). Apress. https://git-scm.com/book

Dowle, M., & Srinivasan, A. (2021). *data.table: Extension of data.frame.* https://rdatatable.gitlab.io/data.table/

Efron, B. (1979). Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, *7*(1), 1–26.

Efron, B., & Tibshirani, R. (1986). Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical Science*, *1*(1), 54–75. https://doi.org/10.1214/ss/1177013815

Efron, B., & Tibshirani, R. J. (1993). *An introduction to the bootstrap*. Chapman; Hall/CRC.

Faraway, J. J. (2016). *Extending the linear model with R: Generalized linear, mixed effects and nonparametric regression models* (2nd ed.). Chapman; Hall/CRC.

Gelman, A., Hill, J., & Vehtari, A. (2020). *Regression and other stories*. Cambridge University Press.

Gentle, J. E. (2024). *Matrix algebra: Theory, computations, and applications in statistics* (3rd ed.). Springer. https://doi.org/10.1007/978-3-031-42144-0

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press.

Grolemund, G., & Wickham, H. (2017). *R for data science* (1st ed.). O'Reilly Media. https://r4ds.had.co.nz/

Halko, N., Martinsson, P.-G., & Tropp, J. A. (2011). Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, *53*(2), 217–288.

Harrell, F. E. (2015). *Regression modeling strategies* (2nd ed.). Springer.

Hastie, T., Tibshirani, R., & Wainwright, M. (2015). *Statistical learning with sparsity: The lasso and generalizations*. Chapman; Hall/CRC. https://web.stanford.edu/~hastie/StatLearnSparsity/

Healy, K. (2018). *Data visualization: A practical introduction*. Princeton University Press. https://socviz.co

Hesterberg, T. C. (2015). What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum. *The American Statistician*, *69*(4), 371–386. https://doi.org/10.1080/00031305.2015.1089789

Lange, K. (2010). *Numerical analysis for statisticians* (2nd ed.). Springer.

Legler, J., & Roback, P. (2019). *Broadening your statistical horizons: Generalized linear models and multilevel models*. https://bookdown.org/roback/bookdown-bysh/

Matuschek, H., Kliegl, R., Vasishth, S., Baayen, H., & Bates, D. (2017). Balancing Type I error and power in linear mixed models. *Journal of Memory and Language*, *94*, 305–315.

McCullagh, P., & Nelder, J. A. (1989). *Generalized linear models* (2nd ed.). Chapman; Hall/CRC.

McCulloch, C. E., Searle, S. R., & Neuhaus, J. M. (2008). *Generalized, linear, and mixed models* (2nd ed.). Wiley.

McElreath, R. (2020). *Statistical rethinking: A bayesian course with examples in R and Stan* (2nd ed.). Chapman; Hall/CRC. https://xcelab.net/rm/statistical-rethinking/

Morris, T. P., White, I. R., & Crowther, M. J. (2019). Using simulation studies to evaluate statistical methods. *Statistics in Medicine*, *38*(11), 2074–2102. https://doi.org/10.1002/sim.8086

Nocedal, J., & Wright, S. J. (2006). *Numerical optimization* (2nd ed.). Springer.

Pennsylvania State University Department of Statistics. (n.d.). *STAT 504: Introduction to GLMs*. Online course notes. Retrieved https://online.stat.psu.edu/stat504/lesson/6/6.1

Petersen, K. B., & Pedersen, M. S. (2012). *The matrix cookbook*. https://www2.imm.dtu.dk/pubdb/edoc/imm3274.pdf

Rizzo, M. L. (2019). *Statistical computing with R* (2nd ed.). Chapman; Hall/CRC.

Sievert, C. (2020). *Interactive web-based data visualization with R, plotly, and shiny*. Chapman; Hall/CRC. https://plotly-r.com/

Strang, G. (2016). *Introduction to linear algebra* (5th ed.). Wellesley-Cambridge Press. https://ocw.mit.edu/courses/18-06-linear-algebra-spring-2010/

Trefethen, L. N., & Bau III, D. (1997). *Numerical linear algebra*. SIAM.

Tufte, E. R. (2001). *The visual display of quantitative information* (2nd ed.). Graphics Press.

UCLA OARC Statistical Methods and Data Analytics. (n.d.). *Regression diagnostics*. Online tutorial. Retrieved https://stats.oarc.ucla.edu/r/dae/regression-diagnostics/

University of Wisconsin-Madison Social Science Computing Cooperative. (n.d.). *Generalized linear models in R*. Online tutorial. Retrieved https://sscc.wisc.edu/sscc/pubs/glm-r/

Wickham, H. (2016). *ggplot2: Elegant graphics for data analysis*. Springer-Verlag. https://ggplot2-book.org

Wickham, H. (2019). *Advanced r* (2nd ed.). Chapman; Hall/CRC. https://adv-r.hadley.nz

Wickham, H. (2021). *Mastering Shiny*. O'Reilly Media. https://mastering-shiny.org/

Wickham, H., & Bryan, J. (2023). *R packages* (2nd ed.). O'Reilly Media. https://r-pkgs.org

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). *R for data*

*science: Import, tidy, transform, visualize, and model data* (2nd ed.). O'Reilly Media. https://r4ds.hadley.nz

Wilke, C. O. (2019). *Fundamentals of data visualization*. O'Reilly Media. https://clauswilke.com/dataviz/

# Credits

## Cover

TODO: add source and licence of the cover image here once chosen.

## Portraits

The portraits of foundational figures in statistical computing appear in this book with attribution to their original sources. Each image is reproduced under the terms stated; please contact the book author for removal if any attribution is incorrect.

| Figure | Where | Source | Licence |
|---|---|---|---|
| Gene H. Golub | Chapter 6 | TODO | TODO |
| Jorge Nocedal | Chapter 7 | TODO | TODO |
| Bradley Efron | Chapter 9 | TODO | TODO |
| John A. Nelder | Chapter 11 | TODO | TODO |
| Douglas Bates | Chapter 12 | TODO | TODO |

## Data sources

- `palmerpenguins::penguins` under CC0.

- Examples drawn from `ADNIMERGE` are subject to the ADNI Data Use Agreement and are not redistributed.

## Code snippets

Unless otherwise stated, code blocks in this book are released under the MIT License.

# Colophon

This book was written in Quarto using the `cosmo` theme with local overrides in `scai.scss`. R code is executed by `knitr` and rendered with `downlit` for function-level hyperlinks.

Body text is set in Source Serif 4 (fallbacks Charter, Georgia, Times New Roman); headings and UI in the same face; code in JetBrains Mono (fallbacks Fira Mono, Menlo). Syntax highlighting uses the accessibility-tuned `arrow` palette, consistent between HTML and PDF output.

PDF output is produced by `xelatex` with the `scrbook` document class at $6.5 \times 9$ inches. The PDF monofont is DejaVu Sans Mono for complete Unicode coverage of box-drawing and mathematical glyphs.

Build reproducibility is managed through `renv` for R packages and a pinned `zzcollab` Docker image for the operating-system and system-library layer.

Source available at https://github.com/rgt47/scai.