Advanced 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

Advanced Statistical Computing in the Age of AI

A graduate textbook in biostatistics

First Edition · 2026

rgtlab

Welcome

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

The book is the second volume in a graduate sequence. It assumes the foundational material covered in the introductory companion volume, *Statistical Computing in the Age of AI*, and treats topics that build on that foundation: numerical stability and conditioning, numerical linear algebra at depth, advanced optimisation, EM and its extensions, Monte Carlo and MCMC in depth, modern Bayesian computation, high-performance and distributed computing, high-dimensional methods, machine learning for biostatistics, software engineering for statisticians, and advanced interactive visualisation.

A workflow companion volume, *Biostatistics Practicum*, covers the day-to-day infrastructure of reproducible biostatistical research (Git, Docker, renv, Quarto, CDISC, SAS) that surrounds the methods both volumes describe.

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

License

Advanced Statistical Computing in the Age of AI

A graduate textbook in biostatistics.

Copyright

To cite this book, please use:

> Thomas, R. G. (2026). *Advanced Statistical Computing in the Age of AI*. Available at https://scai-advanced.rgtlab.org.

Table of contents

Preface

This is the second volume in a graduate sequence on statistical computing for biostatistics. The first volume, *Statistical Computing in the Age of AI*, treats the foundational material a one-quarter graduate course typically covers: programming in R, numerical linear algebra, optimisation, simulation, bootstrap, the standard statistical models, and reproducibility infrastructure. This second volume picks up from where the first concludes and treats the topics that a one-year graduate sequence builds toward.

The book assumes one year of graduate biostatistics study, which means it expects: the linear algebra and probability of a typical first-year sequence, fluency in R at the level of *R for Data Science* and *Advanced R*, working knowledge of generalised linear models and mixed-effects models, and a working bootstrap and simulation literacy. Topics in the introductory SCAI volume are therefore prerequisite, not recap.

What this book covers

The 12-chapter structure is organised in five parts:

1. **Numerical foundations**: computer arithmetic and conditioning; numerical linear algebra in depth.
2. **Optimisation and estimation**: advanced optimisation; the EM algorithm and its extensions.
3. **Monte Carlo and Bayesian computation**: Monte Carlo methods in depth; MCMC in depth; modern Bayesian computation with Stan, variational inference, and model comparison.
4. **Scaling and modelling**: high-performance and distributed computing; high-dimensional and sparse methods; machine learning for biostatistics.

5. **Software engineering and communication**: software engineering for statisticians; advanced interactive visualisation.

The chapter list was constructed by surveying advanced statistical-computing syllabi from a dozen major US biostatistics programmes (the survey is documented in `docs/syllabi-survey.md`) and adopting the topics that appeared in three or more of them. The result is the mainstream curriculum for an advanced graduate computing course, with the addition of explicit AI-collaboration sections in every chapter.

What this book does not cover

The book deliberately omits topics that are typically taught in dedicated *methods* courses elsewhere in a biostatistics curriculum:

- Causal inference (propensity scores, IV, mediation, marginal structural models).
- Longitudinal data analysis beyond the GLMM material in SCAI.
- Missing-data methods in depth (multiple imputation, MNAR sensitivity, FIML).
- Meta-analysis (network and IPD).

Each of these is its own course; their inclusion here would make the book unusual relative to peers and would dilute the computing focus. Pointers to standard references appear where the topics arise.

Age-of-AI framing

Every chapter has two named structural sections that earn the 'in the Age of AI' subtitle. The first, **The statistician's contribution**, is front-loaded: it articulates the judgements at the centre of the chapter that no large language model can make on the reader's behalf. The second, **Collaborating with an LLM on *topic***, is at the end of the chapter and provides specific prompts paired with what to watch for and how to verify. Together they treat AI assistance as an amplifier to be used with discipline,

not a replacement for the statistical judgement the rest of the curriculum exists to build.

Advanced material is exactly where the human-LLM division of labour matters most. Foundational chapters can survive a careless prompt; advanced chapters cannot. The framework is applied throughout.

How to read this book

Each content chapter follows the same structure: Learning objectives, Orientation, The statistician's contribution, content sections (with collapsible Check-your-understanding callouts at natural pauses), Collaborating with an LLM, Exercises, Further reading.

Chapters can be read in order or out of order. Topics with a chain of dependencies (e.g., 06 builds on 05; 07 builds on 06) are noted in the relevant Orientation sections.

Acknowledgements

A peer survey of US biostatistics MS programmes shaped the chapter list. The authors of *R for Data Science*, *Advanced R*, *R Packages*, *Bayesian Data Analysis*, and *Statistical Rethinking* established conventions this book inherits. *Mastering Shiny* and *ggplot2* informed the visual-design decisions and the chapter-template layout.

Conventions

This page summarises the visual conventions used throughout the book.

Code

R code appears in syntax-highlighted blocks. Output is prefixed with `#>` to make the boundary between input and output explicit:

```
mean(c(1, 2, 3, 4, 5))
#> [1] 3
```

Inline code is in `monospace`. Function calls always include parentheses (`mean()` rather than `mean`) so that they are unambiguously functions. Package-qualified calls (`dplyr::filter`) appear when the function is not universally known, when there is name-collision risk, or when the chapter is teaching package usage.

Callouts

Three callout types appear:

> **Tip**
>
> A small practical recommendation.

Check your understanding: example

A short question testing comprehension of the just-read material. Click to expand the answer.

Warning

A pitfall the reader may otherwise hit.

Cross-references

Within this book, sections, figures, and tables are referenced by their Quarto label (`@sec-monte-carlo-human`, `@fig-mcmc-trace`, `@tbl-comparison`). These resolve to clickable links in HTML and proper figure/table numbers in PDF.

References to the companion volumes *Statistical Computing in the Age of AI* and *Biostatistics Practicum* use **prose pointers** rather than Quarto cross-references, because cross-references do not resolve across separate books. For example: 'see the Optimisation chapter of the companion *Statistical Computing in the Age of AI* volume'.

Mathematical notation

Conventional notation throughout. Vectors are bold lower-case ($\mathbf{x}$); matrices are bold upper-case ($\mathbf{X}$); scalars and parameters are non-bold. Estimators carry hats ($\widehat{\theta}$). Sample size is n; parameter dimension is p.

Chapter structure

Every content chapter follows the same template:

1. **Learning objectives.** What you will be able to do after reading.
2. **Orientation.** A short prose framing.

3. **The statistician's contribution.** What no tool can automate. The judgements at the centre of the chapter.
4. **Content sections** with **Check-your-understanding** callouts at natural pauses.
5. **Collaborating with an LLM on the chapter topic.** Prompt / Watch for / Verification triples for AI assistance.
6. **Exercises.** The work.
7. **Further reading.** Where to go next on the topic.

The pattern repeats deliberately. By the third chapter you know where to find each component.

1. Introduction

1.1. What this book is

A graduate textbook in advanced statistical computing for biostatistics, intended as the second volume in a two-book sequence. The introductory volume, *Statistical Computing in the Age of AI*, covers programming, numerical linear algebra, optimisation, simulation, bootstrap, the standard statistical models, and reproducibility infrastructure at a one-quarter graduate pace. This second volume picks up where that leaves off.

1.2. What 'advanced' means here

Three meanings, in increasing strength.

Deeper treatment of foundational topics. Chapters 1 and 2 (numerical stability, numerical linear algebra) revisit material the introductory volume touches but does not treat in depth. Floating-point arithmetic, condition numbers, sparse and iterative solvers, and BLAS-level performance issues are the load-bearing fundamentals for everything else; an advanced book that omits them is missing a layer.

Topics that exceed introductory scope. Chapters 3 through 7 (advanced optimisation, EM and its extensions, Monte Carlo in depth, MCMC in depth, modern Bayesian computation) extend the introductory treatment of each topic into the territory needed for current methodological research and applied practice with modern tools.

Topics that did not appear in the introductory volume at all. Chapters 8 through 12 (high-performance computing, high-dimensional methods, machine learning, software engineering for statisticians, advanced interactive visualisation) cover ground the introductory volume deliberately

did not. Each is the kind of topic a practising biostatistician encounters mid-career and that graduate training should prepare them for.

1.3. What 'in the Age of AI' commits the book to

The subtitle is a structural commitment, not decoration. Every chapter has two named sections that exercise it:

The statistician's contribution. Front-loaded; an explicit articulation of the judgements at the centre of the chapter that no large language model can make on the reader's behalf. Advanced material is exactly where the human-LLM division of labour matters most: a careless prompt in an introductory bootstrap chapter produces a bug that a re-run will catch, but a careless prompt in an HMC or high-dimensional chapter produces results that look plausible and are wrong.

Collaborating with an LLM on `<topic>`. End of chapter; specific prompts paired with what to watch for and how to verify. Each prompt is structured as a triple: the prompt itself, the failure modes the LLM may exhibit, and a verification step the reader runs to catch them.

The framework is applied uniformly across the 12 chapters.

1.4. Reading order

Chapters can be read in order or out of order, with the following dependencies:

- Chapters 1, 2 (numerical foundations) underpin all the numerical work in 3 through 7.
- Chapter 3 (advanced optimisation) feeds chapter 4 (EM and its extensions) and chapter 9 (high-dimensional methods).
- Chapter 5 (Monte Carlo) feeds chapter 6 (MCMC) which feeds chapter 7 (modern Bayesian).

- Chapters 10 (machine learning), 11 (software engineering), and 12 (interactive visualisation) are largely independent of each other and of the numerical chapters, except that 10 uses optimisation tools from chapter 3.

A reader following the dependencies in order treats the book as a course; a reader picking topics ad hoc treats it as a reference.

1.5. What this book does not cover

Topics that are typically taught in dedicated methods courses elsewhere in a biostatistics curriculum:

- Causal inference computing.
- Longitudinal data analysis beyond the introductory GLMM material.
- Missing-data computing in depth.
- Meta-analysis.

The book points to the standard references for each as they arise. The peer-syllabus survey in `docs/syllabi-survey.md` documents the rationale.

1.6. Software environment

The book assumes a current R installation, current Quarto, and the package set used in the introductory volume. New software introduced in this volume is named in the relevant chapter. The companion *Biostatistics Practicum* volume documents the workflow, infrastructure, and deployment conventions assumed throughout.

Part I.

Numerical Foundations

2. Numerical Stability and Conditioning

2.1. Learning objectives

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

- Describe IEEE 754 double-precision floating-point representation and identify the boundary cases (zero, subnormals, infinity, NaN) that affect statistical computation.
- Compute and interpret machine epsilon, and predict when ordinary arithmetic fails to satisfy the laws you remember from high-school algebra (associativity, distributivity).
- Recognise catastrophic cancellation when subtracting two nearly-equal floating-point numbers, and rewrite the affected expression to avoid it.
- Compute the condition number of a matrix or function and interpret it as 'how many digits will I lose on this operation'.
- Distinguish forward error from backward error and explain why backward stability is the right objective for a numerical algorithm.
- Diagnose numerical fragility in R using `kappa()`, `.Machine`, `bench::mark()`, and a small set of red-flag patterns.
- Replace several classic statistical anti-patterns (computing variance via the textbook formula, computing $(X^T X)^{-1}$ by explicit inverse, computing $\log \sum \exp$ naively) with their numerically stable counterparts.

2.2. Orientation

Computer arithmetic is not arithmetic. Every floating-point number is a real number rounded to one of a finite set, spaced unevenly across the real line. Every operation on floating-point numbers introduces a small relative error, on the order of 10^{-16} for double precision. Most of the time you will not notice; the errors are tiny and they do not accumulate to anything visible. Some of the time you will notice catastrophically: a regression that worked on simulated data fails on a real dataset; a likelihood that should be 0.999 returns 1.001; an MCMC chain produces NaN after a million iterations and you have no idea why.

The introductory volume taught the algorithms; this chapter teaches when those algorithms break and how to tell. The material is foundational for every numerical chapter that follows. It is also the kind of material a working biostatistician spends a career rediscovering one bug at a time. Understanding it once, in one place, saves the career-long version.

2.3. The statistician's contribution

An LLM can recite IEEE 754, the definition of a condition number, and the textbook anti-patterns. What it cannot do is recognise when *your* analysis is in numerical trouble. The judgements at the centre of this chapter are diagnostic: knowing what to suspect when something looks slightly off, and what to test for confirmation.

Suspect numerical fragility before suspecting bugs. When a regression returns coefficients that contradict prior work, or a likelihood maximises at a value that disagrees with a closed-form check, the first hypothesis should not be 'I have a bug'. It should be 'something here is ill-conditioned'. The diagnostic is fast: compute the condition number, compute the gradient by finite differences and compare to the analytic gradient, perturb the input slightly and watch the output. If the output moves by more than the perturbation should produce, the problem is not the code; the problem is the floating-point behaviour of the algorithm on this data.

Statistical formulas in textbooks are not numerically stable formulas. The shortcut variance form $\frac{1}{n-1}(\sum x_i^2 - n\bar{x}^2)$ is mathematically correct but numerically fragile: it subtracts two near-equal large numbers when $\bar{x}$ is large relative to σ, and the catastrophic cancellation can produce a *negative* variance estimate. The centred-deviation form $\frac{1}{n-1}\sum(x_i - \bar{x})^2$ is numerically robust and is what R's `var()` actually computes. The textbook often presents the shortcut form without warning. The stable formulations exist in libraries; the ones that you type in from memory will be the textbook ones.

Match precision to need; do not over-promise. A floating-point answer with 16 digits of nominal precision may have only 10 actual digits after a moderately ill-conditioned operation, and only 4 digits after a badly ill-conditioned one. A function that returns a 16-digit number does not advertise this; you advertise it (or fail to). Regression-coefficient point estimates with three digits of precision and a 95% CI are honest. The same estimate with eight digits of precision suggests a level of numerical certainty that the underlying computation cannot supply.

Test the boundary, not just the middle. Synthetic data drawn from $N(0, 1)$ rarely exposes numerical problems. Real biomedical data has near-zero variances, near-collinear covariates, near-saturated probabilities, and observations that span many orders of magnitude. A correctness test on the centre of the data distribution is necessary; a robustness test on the boundary is necessary too.

These judgements are what distinguish numerical code that works on the analyst's laptop from numerical code that works on the next analyst's harder dataset.

2.4. Floating-point arithmetic in one page

A double-precision floating-point number has 64 bits: 1 sign bit, 11 exponent bits, 52 mantissa bits. The number represented is

$$(-1)^s \cdot (1 + m/2^{52}) \cdot 2^{e-1023}$$

with special cases for zero, subnormals (when $e = 0$), infinity (when $e = 2047$ and $m = 0$), and NaN (when $e = 2047$ and $m \neq 0$).

The relative gap between consecutive representable numbers is roughly $2^{-52} \approx 2.22 \times 10^{-16}$. This is **machine epsilon**:

```
.Machine$double.eps
#> [1] 2.220446e-16

.Machine$double.xmax            # largest finite double
#> [1] 1.797693e+308

.Machine$double.xmin            # smallest positive normalised
#> [1] 2.225074e-308
```

The largest integer exactly representable as a double is $2^{53} = 9{,}007{,}199{,}254{,}740{,}992$. Beyond this, consecutive integers are no longer all representable.

```
2^53           == 2^53 + 1
#> [1] TRUE
```

This is one reason patient identifiers and sequence reads are stored as integers or strings, not doubles.

2.4.1. The arithmetic laws change

Floating-point operations satisfy *most* of the laws of real arithmetic, with three important exceptions:

Addition is not associative. $(a+b)+c$ may differ from $a+(b+c)$ because the rounding step between operations depends on operand magnitudes.

```
(1e100 + 1) - 1e100
#> [1] 0
1e100 + (1 - 1e100)
#> [1] 0
1 + 1e100 - 1e100
#> [1] 0     # the 1 is lost
```

The lost 1 is catastrophic cancellation: a small number added to a much larger number is invisible at double precision because the large number's rounding swallows it.

Multiplication is not associative either, though the deviations are typically smaller and harder to construct.

The distributive law is approximate. $a \cdot (b + c) \neq a \cdot b + a \cdot c$ in general, because the parenthesised sum rounds before the multiplication.

For algorithms whose correctness depends on the exact arithmetic laws, every operation is a place where a small error enters. The error analysis question is whether the errors stay bounded or grow.

2.5. Catastrophic cancellation

When you subtract two nearly-equal numbers, the leading significant digits agree and cancel; what remains is a small number whose precision is set by the *trailing* digits of the inputs. Those trailing digits are the rounding error. The result has lost most of its significant figures.

A canonical example: the textbook variance formula.

$$\hat{\sigma}^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2$$

This is mathematically correct. It is also the most numerically stable form when $\bar{x}$ is small. But for x_i near a large mean (a vector of HbA1c values around 6.5 mmol/L, say), an algebraically equivalent rearrangement is

$$\hat{\sigma}^2 = \frac{1}{n-1} \left(\sum x_i^2 - n\bar{x}^2 \right)$$

which is what the textbook 'shortcut formula' actually computes. This form subtracts two large nearly-equal numbers (the sum of squares and $n\bar{x}^2$) and is catastrophically unstable for data far from zero.

```r
x <- 1e9 + rnorm(1000, sd = 1)        # mean ~1e9, sd ~1

# stable: two-pass
mean_x <- mean(x)
var_stable <- sum((x - mean_x)^2) / (length(x) - 1)
var_stable
#> [1] 1.014       # close to true variance ~1

# unstable: shortcut formula
var_naive <- (sum(x^2) - length(x) * mean_x^2) / (length(x) - 1)
var_naive
#> [1] -23.5       # negative; clearly wrong

# what R's var() actually does (two-pass centred-deviation, stable)
var(x)
#> [1] 1.014
```

The shortcut formula gives a *negative* variance estimate. The naive code does not crash; it produces a plausible- looking number that is wrong by orders of magnitude.

The cure is one of:

1. **Two-pass.** Compute the mean first; then sum the squared deviations. Doubles the data passes; numerically robust. This is what R's `var()` and `sd()` use internally (the C source calls a centred-deviation sum after a separate mean pass).
2. **Welford's online algorithm.** Single-pass; updates running mean and sum-of-squares in a numerically stable way. The standard choice for streaming data where two passes are not available.
3. **Centre the data first.** Subtract any reasonable approximation of the mean from every observation; compute on the centred data. Useful when you need to express the variance via a formula that the shortcut form would otherwise tempt you into.

In a regression setting, the same issue appears as $X^T X$ becoming numerically near-singular when X has columns of very different magnitudes. The cure is to centre and scale columns before forming $X^T X$, or to work directly with the QR decomposition of X (which never forms $X^T X$).

Check your understanding: variance formulas

Question. You implement the shortcut variance formula on a vector of patient ages (range 18 to 95, mean about 55). Will it give you the wrong answer?

Answer.

Probably not, but it is on the edge of trouble. Patient ages are roughly $O(50)$, with variance on the order of $O(200)$. The sum of squares is $\sim n \cdot 50^2$ and $n\bar{x}^2$ is also $\sim n \cdot 50^2$; their difference is the variance times $n-1$, so the relative cancellation is about $(n-1)\sigma^2/(n\bar{x}^2) \approx \sigma^2/\bar{x}^2 \approx 200/2500 = 0.08$. You lose about $\log_{10}(1/0.08) \approx 1$ digit of precision. Not catastrophic for n in the thousands, but already worse than the two-pass formulation, which loses zero digits to cancellation. The relative cancellation grows with $\bar{x}^2/\sigma^2$, so for biomarkers measured in units far from zero (HbA1c around 6, eGFR around 90, serum creatinine around 0.9), the shortcut formula loses more digits and eventually gives plainly wrong answers. The right rule is to never use the shortcut formula in new code; the cost of `var()` is negligible and the behaviour is correct on every input.

2.6. The condition number

The condition number of a numerical problem measures how much the solution changes when the input is perturbed. For solving $A\mathbf{x} = \mathbf{b}$:

$$\kappa(A) = \|A\| \cdot \|A^{-1}\|$$

(in any consistent norm). For the 2-norm, $\kappa_2(A)$ is the ratio of the largest to the smallest singular value.

The interpretation: a relative perturbation of size ϵ in the right-hand side $\mathbf{b}$ produces a relative perturbation of size up to $\kappa(A)\epsilon$ in the solution $\mathbf{x}$. With double-precision input, a matrix with $\kappa(A) \approx 10^{12}$ permits roughly $\log_{10}(10^{16}/10^{12}) = 4$ digits of meaningful precision in the answer.

In R:

```r
A <- matrix(c(1, 1, 1, 1.0001), 2, 2)
kappa(A, exact = TRUE)
#> [1] 40004
log10(kappa(A, exact = TRUE))
#> [1] 4.602        # about 11 digits of precision out of 16
```

A few rules of thumb worth memorising:

- $\kappa(A) \leq 100$: well-conditioned. Most or all 16 digits are meaningful.
- $\kappa(A) \approx 10^6$: moderately ill-conditioned. About 10 digits meaningful.
- $\kappa(A) \approx 10^{12}$: severely ill-conditioned. About 4 digits meaningful.
- $\kappa(A) \geq 10^{16}$: numerically singular at double precision. The computed solution is dominated by rounding error.

Common sources of large condition number in biostatistical work:

- **Collinear or near-collinear covariates.** Two columns of X that are nearly proportional inflate $\kappa(X^T X)$ to $\kappa(X)^2$, which is the squared condition number.
- **Wildly different scales.** A design matrix with one column in millions (genome positions) and another in tenths (proportions) has $\kappa(X^T X)$ much larger than necessary. Centre and scale before fitting.
- **Vandermonde-like structure.** Polynomial regression with degree above 5 or so produces matrices whose condition numbers explode. Use orthogonal polynomials (`poly()` instead of `cbind(x, x^2, x^3, ...)`).

For the regression problem $X^T X\beta = X^T y$, the condition number of the *linear system* is $\kappa(X^T X) = \kappa(X)^2$. Working with the QR decomposition of X avoids forming $X^T X$ at all and keeps the conditioning at $\kappa(X)$, not $\kappa(X)^2$. This is why `lm()` uses QR.

2.7. Forward and backward stability

A numerical algorithm computes $\hat{y} = f^*(x)$ rather than the exact $y = f(x)$. There are two ways to think about the error.

Forward error. $\|\hat{y} - y\|$, the distance between the computed and exact answers. This is what you would naturally want to bound, but it depends jointly on the algorithm and the conditioning of the problem.

Backward error. The smallest perturbation Δx such that $\hat{y} = f(x + \Delta x)$. That is, the algorithm produces the *exact* answer to a *slightly different* problem.

Backward error analysis (Wilkinson, 1960s) is the dominant analytical framework in modern numerical analysis. An algorithm is **backward stable** if $\|\Delta x\| \leq C \cdot \epsilon_{\text{mach}}$ for some modest constant C. The forward error is then bounded by $\kappa(f) \cdot \|\Delta x\|$, so:

$$\frac{\|\hat{y} - y\|}{\|y\|} \lesssim \kappa(f) \cdot \epsilon_{\text{mach}}$$

This factorisation separates *algorithm quality* (whether the algorithm is backward stable) from *problem difficulty* (the condition number). A backward stable algorithm produces forward errors as small as the problem permits; an ill-conditioned problem cannot be solved accurately by any algorithm operating in finite precision.

The practical consequence: when you observe a large forward error, ask which factor is responsible. If the condition number is huge, no algorithm can save you; either accept the limited precision, regularise the problem, or use higher-precision arithmetic. If the condition number is small but the answer is still wrong, the algorithm is unstable; replace it.

2.8. Diagnostic patterns

A small set of patterns surfaces most numerical fragility in statistical code.

Pattern 1: compare to a closed-form benchmark. When a numerical answer disagrees with a closed-form check, the question is whether the disagreement is at the rounding level (acceptable) or many orders of magnitude larger (a bug or instability).

```r
# closed form for OLS on synthetic data
set.seed(1)
n <- 100; p <- 3
X <- cbind(1, matrix(rnorm(n * p), n, p))
beta_true <- c(2, 1, -0.5, 0.3)
y <- X %*% beta_true + rnorm(n)

# fit two ways; compare
beta_lm  <- coef(lm(y ~ X[, -1]))
beta_qr  <- qr.coef(qr(X), y)

max(abs(beta_lm - beta_qr))
#> [1] 1.776e-15      # at machine precision; agreement
```

If this agreement breaks (becomes 10^{-6} or larger), investigate.

Pattern 2: perturb and watch. Add a small noise to the input; recompute. The output should move by an amount proportional to the noise.

```r
A <- matrix(c(1, 1, 1, 1.0001), 2, 2)
b <- c(2, 2.0001)

x_orig <- solve(A, b)
x_pert <- solve(A, b + c(0, 1e-10))   # tiny perturbation
max(abs(x_pert - x_orig))
#> [1] 1e-06          # output moves 10000x more than input
```

The output moved 10^4 times more than the input. That is the condition number ($\approx 4 \times 10^4$) at work. For research code, automate this check on real data and flag any operation where the output sensitivity is larger than expected.

Pattern 3: compute the gradient two ways and compare. For an analytic gradient, compare to a finite-difference gradient. Disagreement above 10^{-6} or so is a bug, typically in the analytic version.

```r
library(numDeriv)

f  <- function(x) sum((x - 0.3)^2)
gf <- function(x) 2 * (x - 0.3)

x  <- runif(5)
max(abs(gf(x) - grad(f, x)))
#> [1] 4.5e-11        # agreement at numerical-derivative precision
```

Pattern 4: monitor the magnitude of intermediate quantities. When debugging an MCMC chain that produces NaN, log every intermediate variance estimate, every covariance matrix's condition number, every likelihood contribution. The first to overflow, underflow, or become singular is the immediate cause; the algorithmic root cause is one or two steps upstream.

2.9. Stable rewrites of unstable formulas

Three classic anti-patterns and their stable replacements.

Antipattern: $\log \sum_i \exp(z_i)$ computed naively.

```r
z <- c(1000, 1001)
log(sum(exp(z)))
#> [1] Inf            # overflow
```

The fix is the log-sum-exp trick: factor out the maximum before the exponential.

```r
logsumexp <- function(z) {
  M <- max(z)
  M + log(sum(exp(z - M)))
}
logsumexp(z)
#> [1] 1001.313
```

This appears whenever you normalise a probability across many categories: softmax, mixture-model responsibilities, log-likelihood contributions in models with many random effects. R's `matrixStats::logSumExp` provides the robust implementation.

Antipattern: $1 - \exp(z)$ for z near zero.

```
z <- 1e-10
1 - exp(z)
#> [1] -1e-10         # severe cancellation
```

R provides `expm1(z)` which computes $\exp(z) - 1$ accurately for small z:

```
-expm1(z)
#> [1] -1e-10         # but with full precision
```

Likewise `log1p(z)` computes $\log(1 + z)$ accurately for small z. These appear in CDFs for distributions that are close to 1 in the upper tail, in the logistic-regression log-likelihood for predictions near 0 or 1, and in survival analysis.

Antipattern: solving $X^T X\beta = X^T y$ via explicit inverse.

```
beta <- solve(t(X) %*% X) %*% t(X) %*% y          # bad
beta <- solve(crossprod(X), crossprod(X, y))      # better
beta <- qr.coef(qr(X), y)                         # best
```

The first form forms $X^T X$ and inverts it explicitly, inflating the condition number from $\kappa(X)$ to $\kappa(X)^2$ and then computing an inverse, which is both numerically worse and computationally wasteful. The QR-based form avoids forming $X^T X$ entirely and preserves the original conditioning.

> Check your understanding: log-sum-exp
>
> **Question.** Why does the log-sum-exp trick avoid overflow when the naive form does not?
> **Answer.**
> The naive form computes $\exp(z_i)$ for every z_i. For any z_i above about

709, this overflows to infinity at double precision. The log-sum-exp trick factors out the maximum: $\log \sum \exp(z_i) = M + \log \sum \exp(z_i - M)$ where $M = \max_i z_i$. Now the exponentials inside the sum have arguments $z_i - M \le 0$, so each $\exp(\cdot)$ is in $(0, 1]$ and cannot overflow. The sum and log are applied to a small finite number, then M is added back on the log scale. The mathematics is identical; the floating-point behaviour is dramatically different. This is the canonical example of a *numerically stable rewrite*: same answer, different computation, no overflow.

2.10. Worked example: a numerically robust logistic likelihood

The logistic-regression log-likelihood is

$$\ell(\beta) = \sum_i \left[y_i \mathbf{x}_i^T \beta - \log(1 + \exp(\mathbf{x}_i^T \beta)) \right].$$

The naive computation of `log(1 + exp(eta))` overflows when η is large positive and underflows (returning $\log 1 = 0$) when η is large negative. The robust form uses `log1p` and a max trick:

```
loglik_logistic <- function(beta, X, y) {
  eta <- as.numeric(X %*% beta)
  # log(1 + exp(eta)) without overflow:
  # for eta >= 0: eta + log(1 + exp(-eta))
  # for eta <  0: log(1 + exp(eta))
  # combined via max trick:
  m   <- pmax(eta, 0)
  log1pe <- m + log1p(exp(-abs(eta)))
  sum(y * eta - log1pe)
}
```

Compare to the naive form on a difficult input:

```r
set.seed(1)
n <- 200
X <- cbind(1, scale(matrix(rnorm(n * 3), n, 3)))
beta <- c(0.5, 50, -50, 0)              # extreme coefs to force troubl
y <- rbinom(n, 1, plogis(X %*% beta))

beta_test <- c(0, 60, -60, 0)
naive <- function(beta, X, y) {
  eta <- as.numeric(X %*% beta)
  sum(y * eta - log(1 + exp(eta)))
}

naive(beta_test, X, y)
#> [1] -Inf            # log(exp(huge)) overflows
loglik_logistic(beta_test, X, y)
#> [1] -2754.3         # finite and meaningful
```

The naive version returns `-Inf`, which an optimiser treats as a barrier and the chain stops; the robust version returns a finite value the optimiser can work with.

2.11. Collaborating with an LLM on numerical stability

LLMs are reasonable at recalling the textbook anti-patterns in this chapter and at writing the stable replacements. They are less reliable at identifying which of *your* specific formulas are unstable, because that requires working through the magnitudes that arise in your data.

Prompt 1: review for instability. Paste a short function (a likelihood, a moment computation, a closed-form estimator) and ask: 'identify any numerical-stability risks in this code, and propose stable rewrites where relevant.'

What to watch for. The LLM should flag: subtractions of nearly-equal quantities, naive `log(1 + exp(x))` and `log(sum(exp(x)))`, explicit matrix

inverses, computations of variance via the shortcut formula, and accumulation of many small additions to a large running total. If it flags none on a function that contains any of these, push back; if it flags more, evaluate each.

Verification. Construct an extreme input (mean far from zero, near-collinear design, near-zero variance) and run both versions. The stable rewrite should produce a finite sensible number; the unstable version should produce overflow, underflow, NaN, or wildly wrong values.

Prompt 2: explain a numerical bug. Paste the failing output and the surrounding code, ask: 'this returns NaN on a small subset of inputs. Trace the failure: what is the first quantity that becomes NaN, and why?'

What to watch for. The LLM is good at the standard failure cascade (zero variance → divide by zero → NaN; a covariance matrix with a tiny negative eigenvalue from rounding → Cholesky fails). It is less reliable for domain-specific cascades (a hierarchical-prior scale collapsing to zero in a Bayesian model). If the answer sounds vague, ask the LLM to state the failure in terms of specific quantities and where they were last finite.

Verification. Add `stopifnot()` or `cat(...)` checkpoints at each stage of the computation. The first checkpoint that triggers identifies the precise failure point; compare to what the LLM predicted.

Prompt 3: produce a numerically robust implementation. Describe the mathematical formula and the input ranges, ask: 'implement this in R, prioritising numerical robustness over brevity. Use `log1p`, `expm1`, `logSumExp`, `crossprod`, `qr.coef`, and similar where applicable. Comment on each numerical-stability decision in the code.'

What to watch for. The LLM should produce code with explicit stability rationale in comments. If the comments are absent or generic, the implementation may be a straight transcription of the formula without thinking about magnitudes. If the code matches a textbook anti- pattern, ask for a specific rewrite.

Verification. Test on three input regimes: typical (centred near zero, modest variance), extreme (mean far from zero), and pathological (near-collinear design or saturated probabilities). The robust implementation should work on all three; the textbook implementation will fail on the second or third.

The meta-lesson: numerical robustness is not a correctness property of the formula; it is a property of how the formula is *computed*. LLMs assist with the computation but cannot recognise instability without you naming the specific risk.

2.12. Principle in use

Three habits define defensible numerical practice:

1. **Compute condition numbers.** Whenever you solve a linear system, fit a regression, or invert a matrix, compute and look at the condition number. It is one line of code and surfaces the numerical risk before it becomes a wrong answer.
2. **Use stable rewrites by default.** `log1p`, `expm1`, `logSumExp`, `crossprod`, `qr.coef` for OLS, two-pass variance. Make the stable form your habit; reach for the unstable form only when you have specifically verified its safety.
3. **Test the boundary.** Synthetic data centred near zero rarely exposes problems. Run on data with means far from zero, near-collinear covariates, and probabilities near 0 or 1. The boundary cases are where production data lives.

2.13. Exercises

1. Compute the variance of `1e9 + rnorm(1e5)` two ways: the shortcut formula and `var()`. Document the difference. Now repeat with `1e15 + rnorm(1e5)`. At what magnitude of mean does the shortcut formula become useless?
2. Construct a 5×5 Hilbert matrix $H_{ij} = 1/(i + j - 1)$. Compute its condition number. Solve $H\mathbf{x} = \mathbf{b}$ for $\mathbf{b}$ chosen so the exact answer is $\mathbf{x} = (1, 1, 1, 1, 1)$. Report the relative error. Repeat for $n = 10$ and $n = 12$.
3. Implement a numerically robust softmax function: $\sigma_i(\mathbf{z}) = \exp(z_i)/\sum_j \exp(z_j)$. Verify on inputs containing values larger than 700, where naive softmax overflows.

4. The logistic CDF $\Phi(z) = (1 + e^{-z})^{-1}$ has a naive implementation that returns 1 for large positive z (true) and 0 for large negative z (also true). Implement `log(plogis(z))` accurately for both tails. Hint: this is what R's `plogis(z, log.p = TRUE)` does; verify your implementation against it.
5. Take a regression of yours from a previous chapter or a real analysis. Compute the condition number of X and of $X^T X$. Note the ratio. Compute the OLS coefficients via the explicit-inverse formula and via `qr.coef`. Compare the two answers in their last few digits.

2.14. Further reading

- (Goldberg, 1991), 'What every computer scientist should know about floating-point arithmetic'. The canonical introduction. Free at `dl.acm.org`.
- (Higham, 2002), *Accuracy and Stability of Numerical Algorithms*, 2nd ed. The reference. Dense.
- (Trefethen & Bau III, 1997), *Numerical Linear Algebra*. The treatment of conditioning and stability in chapters 12–15 is excellent.
- The R documentation at `?.Machine` and the `matrixStats` package vignettes for stable implementations of common reductions.

3. Numerical Linear Algebra in Depth

3.1. Learning objectives

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

- Distinguish dense and sparse linear algebra and choose the appropriate representation for a given problem.
- Recognise the standard sparse storage formats (COO, CSR, CSC) and the `Matrix` package's `dgCMatrix`, `dsCMatrix`, and `dgRMatrix` classes.
- Choose between direct solvers (LU, Cholesky, QR) and iterative solvers (Conjugate Gradient, GMRES, BiCGSTAB) for a given linear system.
- Reuse a single matrix factorisation across many right- hand sides instead of refactoring per solve.
- Reason about the BLAS layer: levels 1, 2, and 3 BLAS, what each level does, and which dominates the runtime of a given algorithm.
- Identify when alternative BLAS implementations (OpenBLAS, MKL, vecLib) materially change runtime.
- Use specialised storage (banded, symmetric, positive definite) and matching specialised solvers for matrices that have exploitable structure.
- Apply preconditioning for iterative solvers when the unpreconditioned condition number is too large for reasonable convergence.

3.2. Orientation

The introductory volume (**?@sec-matrix-algebra**) covered dense matrix operations: `%*%`, `solve`, `crossprod`, the basic decompositions, the OLS normal equations done correctly. That treatment assumed the matrices fit

in RAM and that all entries were potentially nonzero. Real biostatistical work routinely violates one or both assumptions: design matrices for mixed-effects models with hundreds of subjects can be sparse with most entries exactly zero; genome-wide association matrices are large enough that direct factorisation costs prohibitive memory; matrices arising from PDE-based imaging methods have banded or block-banded structure that the dense algorithms ignore.

This chapter teaches the tools for those cases: sparse storage and sparse algorithms, iterative solvers that never form the matrix factorisation, and the BLAS layer that ultimately runs every dense algorithm beneath the R-language interface. The treatment is focused on the practical: which tool to reach for in which situation, and what the failure modes are when you reach for the wrong one.

3.3. The statistician's contribution

The choice of algorithm and storage format is a modelling choice in disguise. The statistical question is the same whether you store X as a dense matrix or a sparse one, but the dense path may use 100 GB of RAM and run for a week, while the sparse path uses 100 MB and runs in an hour. Knowing which path your problem permits is part of the job.

Recognise the structure before forming the matrix. A mixed-effects design matrix with subject-level random intercepts has a known sparsity pattern (one nonzero per row in the random-effects block) that you should exploit. Forming a dense $(n \times q)$ block where most entries are zero wastes orders of magnitude of memory and compute. The fix is upstream: build the matrix with `Matrix::sparseMatrix` from the start, not by densifying a sparse construction.

Match solver to structure. A symmetric positive- definite system should use Cholesky, not LU. A banded system should use a banded solver. A least-squares system should use QR. Reaching for the general-purpose `solve(A, b)` discards exploitable structure and produces solutions an order of magnitude slower than necessary.

Direct vs. iterative is a regime decision. For $n < 10^4$ and dense, direct solvers (LU, Cholesky, QR) are almost always right. For $n > 10^5$ and sparse,

iterative solvers (CG, GMRES) become competitive or necessary. The crossover depends on sparsity and conditioning, not just on size. Knowing where your problem sits is what decides the implementation.

The BLAS choice matters more than the R code. A matrix multiplication that takes 30 seconds with the reference BLAS may take 3 seconds with OpenBLAS or 1 second with MKL. The R code is identical; the BLAS implementation differs. For matrix-heavy work, verifying the active BLAS via `sessionInfo()` is the single highest- leverage diagnostic.

These judgements are what convert a textbook implementation that works on toy problems into production code that works on real biomedical data sizes.

3.4. Sparse linear algebra

A matrix is **sparse** when most of its entries are zero and the zeros can be exploited by a specialised storage format. There is no fixed threshold; in practice, a matrix is worth treating as sparse when fewer than 10% of entries are nonzero and the matrix is large enough that ordinary dense storage is wasteful.

3.4.1. Sparse storage formats

Three formats appear repeatedly. Each represents only the nonzero entries.

COO (coordinate list). A list of triples `(i, j, x)` giving the row, column, and value of each nonzero. The simplest format; convenient for construction from a data frame; not efficient for matrix operations.

CSR (compressed sparse row). Three arrays: the nonzero values in row-major order, the column indices of those values, and a row-pointer array indicating where each row starts. Efficient for row-wise operations and for sparse matrix-vector products $A\mathbf{x}$.

CSC (compressed sparse column). The transpose of CSR: nonzero values in column-major order, row indices, and a column-pointer array.

Efficient for column-wise operations and for sparse matrix-vector products $A^T\mathbf{x}$. This is the default in R's `Matrix` package.

```
library(Matrix)

# 5x5 sparse matrix; only 6 nonzero entries
A <- sparseMatrix(
  i = c(1, 2, 3, 3, 4, 5),
  j = c(1, 2, 1, 3, 4, 5),
  x = c(2.0, 1.5, 0.5, 3.0, 1.0, 2.5),
  dims = c(5, 5)
)
class(A)
#> [1] "dgCMatrix"            # general sparse, column-compressed

object.size(A)
#> 1248 bytes                  # vs ~200 bytes for dense 5x5,
                               # but for 1000x1000 with 6000 nonzeros
                               # sparse is ~30 KB vs dense 8 MB

# operations work like dense, but exploit sparsity
b <- runif(5)
solve(A, b)                    # uses sparse LU
```

The `Matrix` package class hierarchy:

Class	Properties
`dgCMatrix`	General sparse, double, column-compressed
`dsCMatrix`	Symmetric sparse, double, column-compressed
`dtCMatrix`	Triangular sparse, double, column-compressed
`dgRMatrix`	General sparse, double, row-compressed
`lgCMatrix`	General sparse, logical, column-compressed
`nsCMatrix`	Pattern (nonzero structure only), symmetric

The `d` prefix is double precision, `l` is logical, `n` is pattern. The next letter is the structural property: `g` general, `s` symmetric, `t` triangular. The last letter

is the storage: `C` column, `R` row, `T` triplet. For most user code, `dgCMatrix` is the default.

3.4.2. Sparse matrix-vector and matrix-matrix products

Sparse matrix-vector ($A\mathbf{x}$) is the workhorse operation in iterative solvers. Cost is proportional to the number of nonzero entries, not to n^2. For a sparse matrix with k nonzeros, $A\mathbf{x}$ is $O(k)$ rather than $O(n^2)$.

Sparse matrix-matrix products (AB) generally produce denser output than either input. A sparse-times-sparse that fills in many entries (high *fill-in*) loses the sparsity advantage; in such cases, the operation may be slower than its dense equivalent. Compute the result sparsity before assuming sparse multiplication is faster.

3.4.3. When to use sparse storage

- **Yes, sparse:** mixed-effects design matrices with subject-specific random effects, regression with many one-hot-encoded factor levels, kernel matrices with thresholded entries, finite-difference operators on grids, graph adjacency matrices, GWAS genotype matrices stored as 0/1/2.
- **Probably not sparse:** small dense matrices (under 1000×1000), correlation matrices for which every pair has nonzero correlation, design matrices with continuous covariates and no factor-encoded predictors.
- **Maybe sparse:** large design matrices where the sparsity is moderate (10–40%); benchmark to decide.

Check your understanding: when sparsity helps

Question. You are fitting a mixed-effects model with $n = 100{,}000$ observations, 10 fixed-effect predictors, and a random intercept per subject for 5{,}000 subjects. Roughly what fraction of the design matrix is nonzero? Should you use sparse storage?
Answer.
The full design matrix has 10 fixed-effect columns plus 5{,}000 random-

intercept columns, giving 5{,}010 columns total. Each row has nonzero entries in the 10 fixed-effect columns plus exactly 1 of the 5{,}000 random-intercept columns. So there are 11 nonzeros per row out of 5{,}010 entries, a sparsity of about 0.22%.
Yes, definitely use sparse storage. The dense matrix would be $100{,}000 \times 5{,}010$, requiring about 4 GB at double precision, while the sparse matrix requires only the storage for $11 \times 100{,}000 = 1.1$ million nonzeros, about 13 MB. The dense form is unworkable for fitting; the sparse form is routine. This is exactly the problem `lme4` solves internally using sparse linear algebra.

3.5. Iterative solvers

Direct methods (LU, Cholesky, QR) compute the factorisation explicitly and use it to solve. They are exact up to floating-point error and predictable in runtime. They are also $O(n^3)$ in time and $O(n^2)$ in memory for dense matrices. For large sparse problems, the factorisation often produces fill-in that destroys the sparsity, making the direct method impractical.

Iterative solvers compute an approximation to the solution by repeatedly applying the matrix to a sequence of vectors, never forming a factorisation. Each iteration involves one or two matrix-vector products and a few inner products. Convergence is governed by the matrix's spectrum, not by its dimension.

3.5.1. Conjugate gradient (CG)

For symmetric positive-definite systems, the conjugate gradient method finds the exact solution in at most n iterations and the approximate solution in far fewer when the eigenvalues are clustered. Each iteration costs one matrix-vector product plus $O(n)$ work.

```
# pseudocode
cg <- function(A, b, x0 = rep(0, length(b)),
               tol = 1e-8, max_iter = length(b)) {
```

```
  x <- x0
  r <- b - A %*% x
  p <- r
  for (k in seq_len(max_iter)) {
    Ap     <- A %*% p
    alpha  <- as.numeric(crossprod(r) / crossprod(p, Ap))
    x      <- x + alpha * p
    r_new  <- r - alpha * Ap
    if (sqrt(sum(r_new^2)) < tol) break
    beta   <- as.numeric(crossprod(r_new) / crossprod(r))
    p      <- r_new + beta * p
    r      <- r_new
  }
  x
}
```

CG is the standard tool for SPD systems whose factorisation would be too expensive. Common applications: covariance-matrix solves in Bayesian models, normal equations in penalised regression, kernel-method linear systems.

3.5.2. GMRES

For general (non-symmetric) systems, the Generalised Minimal Residual method (GMRES) is the analogue of CG. It is more expensive per iteration (memory grows linearly in the iteration count) and typically restarted every m iterations to bound the memory.

R packages: `Rlinsolve` provides CG, GMRES, BiCGSTAB, and others. Stan and similar use these internally for the preconditioner setup but expose them via different interfaces.

3.5.3. BiCGSTAB and other variants

Biconjugate Gradient Stabilised (BiCGSTAB) is a popular choice for non-symmetric systems where GMRES's memory growth is a problem. Convergence is less predictable but the per-iteration cost is fixed.

3.5.4. Preconditioning

The convergence rate of an iterative method depends on the condition number of the matrix. For poorly-conditioned systems, a *preconditioner* M is applied so that $M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}$ has a better- conditioned coefficient matrix. The preconditioner M should approximate A but be much cheaper to invert.

Common preconditioners:

- **Diagonal (Jacobi).** $M = \text{diag}(A)$. Cheap; mild improvement.
- **Incomplete Cholesky / LU.** Compute a factorisation but allow only certain fill-in patterns. Trade preconditioning quality for cost.
- **Algebraic multigrid (AMG).** For matrices arising from PDEs on grids; complex but very effective.

For most biostatistical applications, diagonal preconditioning is sufficient if needed at all; complex preconditioners are domain expertise.

3.5.5. Direct vs. iterative regime decision

Setting	Direct (LU/Cholesky/QR)	Iterative (CG/GMRES)
Dense, $n < 10^4$	Yes	No
Dense, $n > 10^5$	Memory-prohibitive	Yes
Sparse, low fill-in	Yes (sparse direct)	Optional
Sparse, high fill-in	No	Yes
Many right-hand sides, same matrix	Yes (factor once)	Less efficient
One right-hand side, large matrix	Less efficient	Yes
Need solution to high precision	Yes	Slower
Need approximate solution	Same cost	Stop early; faster

3.6. Reusing factorisations

A common antipattern: solving $A\mathbf{x}_i = \mathbf{b}_i$ for many right-hand sides $\mathbf{b}_i$ by calling `solve(A, b_i)` in a loop. Each call refactorises A.

```
# bad: O(n^3) per call
results <- vector("list", 100)
for (i in seq_len(100)) {
  results[[i]] <- solve(A, B[, i])
}

# good: factorise once, solve many times
A_factor <- chol(A)             # for SPD; use lu() / qr() otherwise
results <- matrix(0, nrow(A), 100)
for (i in seq_len(100)) {
  results[, i] <- backsolve(A_factor, forwardsolve(t(A_factor), B[, i
}
```

The factorisation cost is paid once; each subsequent solve is $O(n^2)$ (back-substitution), not $O(n^3)$. For 100 right-hand sides on a 1000×1000 matrix, the saving is two orders of magnitude.

The same pattern applies to QR for least squares with many response vectors:

```
qrA <- qr(A)
betas <- qr.coef(qrA, Y)        # solves for all columns of Y at onc
```

3.7. The BLAS layer

R's matrix operations dispatch to BLAS (Basic Linear Algebra Subprograms) underneath. BLAS is organised in three levels:

- **Level 1 BLAS.** Vector-vector operations: dot product, scalar-vector multiply (`axpy`). Cost $O(n)$.
- **Level 2 BLAS.** Matrix-vector operations: matrix- vector multiply (`gemv`). Cost $O(n^2)$.

- **Level 3 BLAS.** Matrix-matrix operations: matrix- matrix multiply (`gemm`). Cost $O(n^3)$.

Level 3 BLAS is where high-performance implementations shine. A naive triple loop for $C = AB$ is $O(n^3)$ but runs at perhaps 5–10% of theoretical peak performance because it suffers from poor cache utilisation. Optimised BLAS implementations (OpenBLAS, MKL, vecLib on macOS) use cache blocking, SIMD instructions, and threading to reach 60–90% of peak performance.

Check which BLAS R is using:

```
sessionInfo()
#> ...
#> Matrix products: default
#> BLAS:   /usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3
#> LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/liblapack.so.3
```

The reference BLAS that ships with stock R on Linux is slow. OpenBLAS and MKL produce 5–20× speedups on matrix-heavy code with no R-level changes. On macOS, R typically uses Apple's vecLib (Accelerate framework), which is optimised.

To switch BLAS implementations on Linux:

```
sudo apt install libopenblas-dev
sudo update-alternatives --config libblas.so.3-x86_64-linux-gnu
```

The change is transparent to R code; benchmarks improve immediately.

For matrix-heavy production work, verifying the BLAS is the single highest-leverage performance check. The practical cost of poor BLAS choice is a 5–20× slowdown on every numerical operation, persistent across the life of the code, with no diagnostic.

3.8. Specialised storage and solvers

For matrices with exploitable structure beyond sparsity, specialised storage and matched solvers offer further gains.

3.8.1. Banded matrices

A matrix is **banded** if all nonzero entries lie within a narrow diagonal band. Banded matrices arise in time- series analysis (autoregressive structure), spline smoothing (B-spline basis), and finite-difference methods. Banded LU factorisation is $O(nb^2)$ where b is the bandwidth, much faster than general $O(n^3)$.

```
library(Matrix)
# 1000x1000 tridiagonal matrix
n <- 1000
A <- bandSparse(n, n,
                k = -1:1,
                diagonals = list(rep(-1, n - 1),
                                 rep(2,  n),
                                 rep(-1, n - 1)))
class(A)                          # "dsCMatrix" (sparse symmetric)
solve(A, runif(n))                # uses banded solver
```

3.8.2. Symmetric positive-definite matrices

Cholesky decomposition $A = LL^T$ exists for SPD matrices. The factorisation is roughly half the cost of a general LU; the back-substitution is similar.

```
A_chol <- chol(A)                 # upper triangular R such that A = R
x <- backsolve(A_chol,            # solve R x = ...
       forwardsolve(t(A_chol),    # solve R' z = b
                    b))
```

For sparse SPD, the same idea applies via `Matrix::Cholesky()`, which uses a fill-reducing permutation (typically AMD or METIS) to reduce fill-in during factorisation.

3.8.3. Symmetric indefinite

Symmetric matrices that are not positive definite (e.g., KKT systems in constrained optimisation) use the LDLT decomposition: $A = LDL^T$ with D block-diagonal. LAPACK's `dsytrf` provides this.

3.8.4. Toeplitz and circulant

Toeplitz matrices have constant diagonals (every $T_{i,j}$ depends only on $i - j$). They arise in time- series analysis and signal processing. Circulant matrices are diagonalised by the discrete Fourier transform; matrix-vector multiplication is $O(n \log n)$ via FFT.

For most biostatistics, banded and sparse-SPD are the two structural specialisations that matter most. The others are useful when the application domain calls for them.

3.9. Worked example: GMRF for spatial smoothing

A Gaussian Markov random field (GMRF) prior on a regular grid produces a sparse precision matrix Q with banded structure. Given observations y and the GMRF prior, the posterior mean is the solution to

$$(Q + \tau I)\mu = \tau \mathbf{y}$$

where τ is the observation precision. For a 100×100 grid, Q is $10{,}000 \times 10{,}000$ and sparse with about 50{,}000 nonzeros (each cell has 4 neighbours plus itself).

```
library(Matrix)

# build a 1D GMRF precision (just for illustration; 2D is similar)
n <- 1000
Q <- bandSparse(n, n,
                k = -1:1,
                diagonals = list(rep(-1, n - 1),
```

```
                                  rep(2,  n),
                                  rep(-1, n - 1)))
Q[1, 1] <- 1                     # boundary
Q[n, n] <- 1

tau <- 0.1
y   <- rnorm(n)

# dense path: too expensive for n = 10000+
# system <- as.matrix(Q + tau * Diagonal(n))
# solve(system, tau * y)         # would refactorise general LU

# sparse Cholesky path
M       <- Q + tau * Diagonal(n)
M_chol  <- Cholesky(M, perm = TRUE, super = FALSE)
mu      <- solve(M_chol, tau * y)

# can also solve for many y at once
Y       <- matrix(rnorm(n * 100), n, 100)
mu_mat  <- solve(M_chol, tau * Y)     # one factorisation, 100 solves
```

The `Cholesky` factor is reused across all 100 right-hand sides; the cost is one factorisation plus 100 cheap back-substitutions, instead of 100 separate factorisations.

3.10. Collaborating with an LLM on numerical linear algebra

LLMs handle the syntactic translation between dense and sparse storage well. They are weaker at the structural recognition step (deciding whether to use sparse at all, choosing the right solver) and at BLAS-related diagnostics.

Prompt 1: choosing the right representation. Describe your matrix (dimensions, structural properties, sparsity pattern, frequency of operations) and ask: 'should I use dense or sparse storage? If sparse, which `Matrix` class? Justify the choice in terms of memory and operation cost.'

What to watch for. The LLM should consider whether operations against this matrix produce dense intermediate results (which would defeat the sparse representation), whether structural properties like symmetry or positive- definiteness are exploitable, and the ratio of construction cost to solve cost. If the LLM gives a generic 'use sparse if more than 80% zeros' rule, push for the structural specifics of your problem.

Verification. Build the matrix both ways on a small test case; benchmark a representative operation. The sparse path should be faster *and* lower memory if the choice is correct.

Prompt 2: factorisation reuse. Paste a function that calls `solve(A, b)` repeatedly with the same A and ask: 'rewrite to factorise once and reuse the factorisation across the loop, and explain the asymptotic improvement.'

What to watch for. The rewrite should use one of `chol`, `lu`, `qr`, or `Cholesky` (for sparse) at the top of the loop and `backsolve`/`forwardsolve` or `qr.coef`/`solve(factor, b)` inside. The asymptotic improvement depends on whether A is general ($O(n^3) \to O(n^2)$ per solve), SPD (Cholesky has half-cost factorisation), or sparse with low fill-in (potentially much larger improvement).

Verification. Benchmark both versions on a moderately large A and many right-hand sides. The factorisation- reuse version should be substantially faster.

Prompt 3: BLAS diagnosis. Paste the output of `sessionInfo()` plus a benchmark showing slow matrix multiplication and ask: 'is the BLAS the bottleneck? Recommend a faster alternative for this platform.'

What to watch for. Reference BLAS (no link mentioned in the BLAS line, or `libblas.so` without OpenBLAS or MKL) is slow. The LLM should recommend OpenBLAS or MKL on Linux, and identify Apple's vecLib on macOS as already fast. If the BLAS is already optimised, the bottleneck is elsewhere; the LLM should pivot to other diagnostics.

Verification. Switch the BLAS, restart R, re-run the benchmark. A 5–20× improvement on dense matrix multiply confirms the BLAS was the bottleneck. If the improvement is small, the bottleneck was something else.

The meta-pattern: numerical linear algebra is layered. The R-level code is the surface; the BLAS layer determines performance; the matrix structure determines what algorithm is appropriate. Each layer is a separate diagnostic axis.

3.11. Principle in use

Three habits define defensible numerical linear algebra:

1. **Recognise structure first.** Sparse, banded, symmetric, positive-definite. The structural recognition determines algorithm, storage, and conditioning behaviour.
2. **Factorise once, solve many times.** Reuse factorisations across right-hand sides. The pattern converts $O(kn^3)$ into $O(n^3 + kn^2)$, a large gain for k even moderately large.
3. **Verify the BLAS.** A fast BLAS is the single highest-leverage performance change. `sessionInfo()` tells you what is installed; benchmarks tell you whether it is fast.

3.12. Exercises

1. Build a sparse 1000×1000 tridiagonal matrix with `Matrix::bandSparse`. Compute its memory usage versus the dense equivalent. Solve a linear system against it; benchmark against the dense `solve`.
2. Implement conjugate gradient from the pseudocode in this chapter. Apply to a 5000×5000 sparse SPD matrix. Compare convergence iterations against the dimension; convergence should be much faster than n if the eigenvalues are clustered.
3. For a regression with 100 different response vectors $\mathbf{y}_1, \ldots, \mathbf{y}_{100}$ and a common design matrix X, write the loop two ways: one calling `lm()` 100 times, one factorising X once and reusing. Time both. Report the speedup.
4. Check your R installation's BLAS via `sessionInfo()`. Run `bench::mark(A %*% B)` on A, B each 1000×1000. If your BLAS is the reference BLAS, install OpenBLAS or MKL and re-run. Document the speedup.

5. The Hilbert matrix has condition number $\sim 10^{18}$ for $n = 12$. Solve $H\mathbf{x} = \mathbf{b}$ with exact $\mathbf{x} = \mathbf{1}$ via a direct solver and via CG with diagonal preconditioning. Compare the forward errors.

3.13. Further reading

- (Golub & Van Loan, 2013) Chapters 4–7, the canonical reference for direct methods, including sparse direct.
- (Saad, 2003), *Iterative Methods for Sparse Linear Systems*. Free PDF at `www-users.cse.umn.edu/~saad`.
- (Bates & Maechler, 2010), the `Matrix` package vignettes, in particular ‘Sparse Matrix Classes’ and ‘Sparse Cholesky’.
- The `Rlinsolve` package documentation for accessible R implementations of CG, GMRES, BiCGSTAB.

Part II.

Optimisation and Estimation

4. Advanced Optimisation

4.1. Learning objectives

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

- Recognise constrained optimisation problems and write the KKT conditions for equality-constrained and inequality-constrained cases.
- Express penalised regression (lasso, ridge, elastic net) as constrained optimisation, and use the equivalence to understand the geometry of regularisation.
- Apply proximal methods (proximal gradient, ISTA, FISTA) to non-smooth convex optimisation problems where ordinary gradient descent fails.
- Apply ADMM to splitting problems where two parts of the objective are individually easy but jointly hard.
- Choose between deterministic and stochastic optimisers based on the data size and the structure of the objective.
- Apply L-BFGS for large-scale unconstrained optimisation, and recognise when its limited memory becomes a bottleneck.
- Diagnose convergence in optimisers more rigorously than $\Delta f < 10^{-8}$: KKT residuals, gradient norms, duality gaps.
- Use modern R packages (`CVXR` for general convex, `glmnet` for lasso, `optimx` for unified non-linear optimisation) appropriately.

4.2. Orientation

The introductory volume covered Newton's method and quasi-Newton methods (BFGS, L-BFGS) for unconstrained optimisation. That treatment handles the vast majority of likelihood-based estimators: GLMs, mixed-effects models, survival models. It does not handle:

- **Constrained problems**, where the parameter must lie in a feasible set (probabilities sum to one, variances are nonnegative, the maximum margin classifier).
- **Non-smooth objectives**, where the gradient is undefined at the solution (lasso, group lasso, fused lasso, total-variation denoising, robust regression with absolute-value loss).
- **Very-large-scale problems**, where computing one full gradient is too expensive and stochastic methods are required (deep learning, large-n logistic regression).

This chapter treats the modern optimisation toolkit that covers these cases. The mathematical framing is convex optimisation; the implementations are R packages with mature backends (`CVXR`, `glmnet`, `optimx`, `Rsolnp`).

4.3. The statistician's contribution

Modern optimisation is a deep and rapidly-evolving field. A statistician does not need to become a numerical optimisation researcher; they need to identify which problem they have, choose the matching tool, and verify that the tool worked.

Recognise non-smoothness early. A non-smooth objective (an absolute value, a max, an indicator function) cannot be minimised by methods that assume the gradient exists everywhere. The classic mistake is to feed a lasso objective to `optim(method = 'BFGS')` and accept the warning-laden output. The right tools are proximal methods or coordinate descent; using them is non-optional for the result to be correct.

Constraints are statistical, not just numerical. A constraint that the parameter lies in $[0, 1]$ is usually a constraint that the parameter is a probability. A constraint that two parameters sum to a constant is usually an identifiability constraint imposed by the model. Treating constraints as numerical inconveniences to be reparameterised away can hide problematic model structure. The constraint exists because the science demands it; the optimiser should respect it explicitly, not work around it.

Regularisation is a modelling choice, not a stability trick. The ridge penalty $\lambda\|\beta\|_2^2$ shrinks coefficients toward zero; the lasso penalty $\lambda\|\beta\|_1$

shrinks them toward zero *and* sets some exactly to zero. Choosing among unpenalised, ridge, lasso, or elastic net is a choice about what bias is acceptable in exchange for what variance reduction. Choosing λ is a choice about how much bias. These are statistical choices; the optimiser executes the choice, but the choice does not emerge from the optimiser.

Verify with KKT residuals. A converged optimiser returns a candidate optimum. The verification is the KKT conditions: gradient stationarity, primal feasibility, dual feasibility, complementary slackness. For unconstrained problems this reduces to $\|\nabla f\| < \epsilon$; for constrained problems it is more elaborate and easier to check than to derive. Most modern solvers report KKT residuals; reading them is the analyst's job.

These judgements determine whether the converged result solves the right problem.

4.4. Constrained optimisation

The general constrained problem:

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & g_i(\mathbf{x}) \leq 0, \quad i = 1, \dots, m \\ & h_j(\mathbf{x}) = 0, \quad j = 1, \dots, p \end{aligned}$$

For convex f and convex g_i and affine h_j, the problem is a **convex programme** with strong duality: the global minimum is unique (or, for non-strictly-convex f, on a unique convex set), and the duality gap is zero.

4.4.1. Karush-Kuhn-Tucker (KKT) conditions

For a candidate $\mathbf{x}^*$ to be a constrained optimum, the KKT conditions must hold at some Lagrange multipliers $\lambda \geq 0$ and ν:

$$\nabla f(\mathbf{x}^*) + \sum_i \lambda_i \nabla g_i(\mathbf{x}^*) + \sum_j \nu_j \nabla h_j(\mathbf{x}^*) = 0$$
$$g_i(\mathbf{x}^*) \leq 0$$
$$h_j(\mathbf{x}^*) = 0$$
$$\lambda_i \geq 0$$
$$\lambda_i g_i(\mathbf{x}^*) = 0$$

In words: stationarity (gradient of Lagrangian is zero), primal feasibility (constraints are satisfied), dual feasibility (multipliers are nonnegative), complementary slackness (each multiplier is zero unless its constraint is active).

The KKT conditions generalise 'set the gradient to zero' to constrained problems. They are necessary for an optimum and sufficient when the problem is convex and a constraint qualification holds (Slater's condition: the strict feasibility of at least one point).

4.4.2. Solving constrained problems

For convex problems, R's `CVXR` package is the modern interface. It accepts a problem in disciplined-convex-programming form and dispatches to appropriate solvers:

```
library(CVXR)

# minimise ||X b - y||^2 subject to b >= 0
n <- 200; p <- 5
X <- matrix(rnorm(n * p), n, p)
y <- as.numeric(X %*% c(1, -1, 0, 2, 0) + rnorm(n))

beta <- Variable(p)
objective <- Minimize(sum_squares(X %*% beta - y))
constraints <- list(beta >= 0)

problem <- Problem(objective, constraints)
result  <- solve(problem)
```

```
result$status
#> [1] "optimal"
result$getValue(beta)
#> [1] 0.892 0.000 0.000 1.978 0.030
```

The non-negative solution differs from the unconstrained LS solution; the constraint forces some coefficients exactly to zero. CVXR handles the disciplined-convex algebra and dispatches to ECOS, SCS, or other backends.

For non-convex constrained problems, `Rsolnp::solnp` and `alabama::auglag` provide augmented Lagrangian methods. Convergence is local and the user supplies the Jacobian of the constraints if available.

4.4.3. Equality constraints by reparameterisation

For some equality constraints, reparameterisation eliminates the constraint and makes the problem unconstrained. Common cases:

- $\sum p_i = 1$: parameterise via softmax $p_i = \exp(z_i)/\sum \exp(z_j)$.
- $p \in (0, 1)$: parameterise via logit $p = \text{plogis}(z)$.
- $\sigma > 0$: parameterise via $\log \sigma = z$.
- Correlation $\rho \in (-1, 1)$: parameterise via Fisher z transform $\rho = \tanh(z)$.

Reparameterisation is computationally simpler than constrained optimisation but loses the ability to report constraints' Lagrange multipliers (which carry interpretation as shadow prices in the optimisation). For constraints whose multipliers are not of interest, reparameterisation is preferred.

Check your understanding: when to use CVXR

Question. You want to fit a linear model with the constraint that the coefficient on `age` is between -0.1 and 0.1 (a clinical bound). Should you use CVXR, or can you reparameterise?
Answer.
Either works; CVXR is more transparent. The reparameterisation route would set $\beta_{\text{age}} = 0.1 \tanh(z)$ and optimise uncon-

strained over z. This is fine but hides the constraint inside the parameterisation, and the gradient at the boundary becomes infinite (a barrier in disguise). The CVXR route writes the constraint explicitly: `constraints <- list(beta[age_index] >= -0.1, beta[age_index] <= 0.1)`. The reported solution either lies strictly inside the constraint (the constraint is inactive) or on the boundary with a Lagrange multiplier reporting how much the optimum would improve if the constraint were relaxed. The Lagrange multiplier is clinically interpretable as 'how much would the fit improve if I loosened my bound by 1 unit'. For a constraint with clinical meaning, the explicit form preserves the interpretation.

4.5. Regularisation as constrained optimisation

The lasso problem

$$\min_{\beta} \frac{1}{2n}\|\mathbf{y} - X\beta\|_2^2 + \lambda\|\beta\|_1$$

has an equivalent constrained form

$$\min_{\beta} \frac{1}{2n}\|\mathbf{y} - X\beta\|_2^2 \quad \text{subject to} \quad \|\beta\|_1 \leq t$$

for some t depending on λ. The two are **Lagrangian duals**: as you sweep λ from 0 to infinity, t sweeps from infinity to 0, tracing out the same set of solutions in different parameterisations.

The constrained form makes the geometry visible: the feasible region is the ℓ_1 ball, whose corners on the axes are where the unconstrained-quadratic loss-minimiser usually contacts it. At those corners, some coordinates are exactly zero. This is why the lasso produces sparse solutions: the geometry of the ℓ_1 ball forces zeros at the corners.

The same geometry argument distinguishes lasso (ℓ_1, sparse solutions, corners on axes) from ridge (ℓ_2, shrunk solutions, no corners) and from

group lasso (block ℓ_2, sparse at the group level). Each penalty's ball shape determines what kind of sparsity the solution has.

For implementation, `glmnet` is the production-grade solver for lasso, ridge, and elastic net. It uses coordinate descent and produces the entire regularisation path in a single fit.

```
library(glmnet)
fit <- glmnet(X, y, alpha = 1)        # lasso (alpha = 1)
plot(fit, xvar = "lambda")            # solution paths
cv  <- cv.glmnet(X, y, alpha = 1)
coef(cv, s = "lambda.min")
```

`alpha = 1` is lasso; `alpha = 0` is ridge; `alpha` between 0 and 1 is elastic net. The `cv.glmnet` cross- validation chooses λ. We treat the high- dimensional setting in detail in Chapter 10.

4.6. Proximal methods

Gradient descent updates $\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x})$. This requires f to be differentiable. For non-smooth objectives, gradient descent does not directly apply.

The **proximal gradient** method handles $f(\mathbf{x}) = g(\mathbf{x}) + h(\mathbf{x})$ where g is smooth and h is convex but possibly non-smooth. The update is

$$\mathbf{x}^{(k+1)} = \operatorname{prox}_{\eta h}\big(\mathbf{x}^{(k)} - \eta \nabla g(\mathbf{x}^{(k)})\big)$$

where the **proximal operator** of h is

$$\operatorname{prox}_{\eta h}(\mathbf{z}) = \arg\min_{\mathbf{x}} \left\{ \frac{1}{2\eta} \|\mathbf{x} - \mathbf{z}\|^2 + h(\mathbf{x}) \right\}.$$

For many non-smooth h, the proximal operator has a closed form:

$h(\mathbf{x})$	$\text{prox}_{\eta h}(\mathbf{z})$
$\lambda\|\mathbf{x}\|_1$	$\text{sign}(z_i)\cdot\max(\|z_i\|-\eta\lambda,0)$ (soft thresh)
$\lambda\|\mathbf{x}\|_2^2$	$\mathbf{z}/(1+2\eta\lambda)$
$\mathbf{1}_{\mathbf{x}\geq 0}$	$\max(\mathbf{z},0)$ (projection onto nonnegative orthant)
$\mathbf{1}_{\|\mathbf{x}\|_2\leq 1}$	projection onto the unit ball

So lasso is proximal gradient with the soft-thresholding operator. Each iteration is one gradient step on the smooth part followed by one soft-threshold; the cost per iteration is the same as gradient descent.

ISTA (Iterative Soft-Thresholding Algorithm) is the proximal gradient applied to lasso. **FISTA** (Fast ISTA) adds Nesterov acceleration for an $O(1/k^2)$ rather than $O(1/k)$ convergence rate.

```
# proximal gradient for lasso, illustrative
prox_grad_lasso <- function(X, y, lambda, eta = 0.001,
                            n_iter = 1000) {
  beta <- rep(0, ncol(X))
  for (k in seq_len(n_iter)) {
    grad <- t(X) %*% (X %*% beta - y) / nrow(X)
    z    <- beta - eta * grad
    beta <- sign(z) * pmax(abs(z) - eta * lambda, 0)
  }
  as.numeric(beta)
}
```

In practice, use `glmnet` for lasso; `prox_grad_lasso` is educational.

4.7. ADMM

The Alternating Direction Method of Multipliers solves

$$\min_{\mathbf{x},\mathbf{z}} f(\mathbf{x})+g(\mathbf{z}) \quad \text{subject to} \quad A\mathbf{x}+B\mathbf{z}=\mathbf{c}$$

by alternating between updating $\mathbf{x}$ (treating $\mathbf{z}$ and the dual variable $\mathbf{u}$ as fixed), updating $\mathbf{z}$ (treating $\mathbf{x}$ and $\mathbf{u}$ as fixed), and updating $\mathbf{u}$ (the dual ascent step).

ADMM is most useful when f and g are individually easy to optimise (each has a simple proximal operator) but their sum is hard. Examples:

- **Generalised lasso** with a structured penalty $\|D\beta\|_1$ for some difference matrix D (fused lasso, total variation).
- **Distributed optimisation**, where the data is split across machines and each machine handles its own block with a consensus constraint.
- **Constrained estimation** where the constraint is expressed via g as an indicator function.

The Boyd et al. (2011) monograph is the standard reference; the `ADMM` and `genlasso` R packages provide implementations for common biostatistical cases.

4.8. Stochastic optimisation

For very large n, computing one full gradient $\nabla f(\beta) = \sum_i \nabla \ell_i(\beta)$ is the bottleneck. Stochastic gradient descent (SGD) replaces the full gradient with a stochastic estimate based on one observation or a small mini-batch:

$$\beta^{(k+1)} = \beta^{(k)} - \eta_k \nabla \ell_{i(k)}(\beta^{(k)})$$

where $i(k)$ is sampled uniformly from $\{1, \dots, n\}$.

The trade-off: each iteration is $O(1)$ rather than $O(n)$, but the iterations are noisy. With a decreasing step size $\eta_k = O(1/k)$, SGD converges in expectation at the optimal rate for non-strongly-convex problems.

Modern variants:

- **SGD with momentum.** Adds a velocity term; accelerates convergence on poorly-conditioned problems.
- **Adam.** Combines momentum with per-parameter step-size adaptation. Default optimiser for deep learning; competitive for many statistical learning problems.

- **RMSProp, AdaGrad.** Earlier per-parameter adaptive methods.

For statistical applications, SGD-family optimisers are the right choice when:

- n is in the millions or more.
- The model is differentiable end-to-end (deep learning, large-n GLMs).
- Approximate solutions are acceptable.

For n in the thousands or tens of thousands, full-batch methods (BFGS, L-BFGS, Newton, IRLS for GLMs) are usually faster and more accurate. The crossover is problem- specific; benchmark before defaulting to either side.

4.9. L-BFGS in detail

L-BFGS is the default for large-scale unconstrained optimisation in R (`optim(method = 'L-BFGS-B')` for box constraints, `lbfgs` package for unconstrained). Its distinguishing feature is the limited-memory approximation to the inverse Hessian: instead of storing an $n \times n$ matrix, it stores the last m pairs of parameter and gradient differences (typically $m = 5$ to 20).

The cost per iteration is $O(mn)$, where n is the parameter dimension. For n in the thousands, this is tractable; for n in the millions (deep neural networks), even L-BFGS is too expensive and SGD is the only option.

L-BFGS is sensitive to the initial step size and to the line-search routine. Convergence problems often resolve by tightening the tolerance, increasing the memory length m, or providing better initial values.

```
# example: maximum likelihood for a logistic regression
neg_loglik <- function(beta, X, y) {
  eta <- as.numeric(X %*% beta)
  # numerically stable: -log(1 + exp(eta)) is unsafe for
  # eta >> 0 (overflow). Use the rewriting from Ch 1.
  -sum(y * eta - pmax(eta, 0) - log1p(exp(-abs(eta))))
}
```

```r
gradient <- function(beta, X, y) {
  p <- plogis(as.numeric(X %*% beta))
  -as.numeric(t(X) %*% (y - p))
}

set.seed(1)
n <- 1000; p <- 50
X <- cbind(1, matrix(rnorm(n * p), n, p))
y <- rbinom(n, 1, plogis(X %*% rnorm(p + 1)))

fit <- optim(rep(0, p + 1), neg_loglik, gr = gradient,
             X = X, y = y, method = 'L-BFGS-B',
             control = list(maxit = 200))
fit$convergence
#> [1] 0
```

For larger problems, `lbfgs::lbfgs` is faster and more configurable. For very large problems, `RhpcBLASctl::blas_set_num_threads(1)` inside the L-BFGS call prevents BLAS oversubscription when multiple L-BFGS instances run in parallel.

4.10. Convergence diagnostics beyond Δf

Most optimisers stop when $|f^{(k+1)} - f^{(k)}| < \epsilon_f$ or $\|\beta^{(k+1)} - \beta^{(k)}\| < \epsilon_\beta$. These are necessary conditions but not sufficient; the optimiser may stall at a flat region without being at an optimum.

Better diagnostics:

- **Gradient norm.** $\|\nabla f(\beta)\| < \epsilon_g$. Direct test of stationarity.
- **KKT residuals.** For constrained problems, all KKT conditions must hold. Modern solvers report each residual.
- **Duality gap.** For convex problems, the duality gap $f(\mathbf{x}^*) - g(\lambda^*)$ where g is the dual function bounds the suboptimality. Some solvers (CVXR's interior-point methods) report this directly.

- **Multiple starting points.** For non-convex problems, run from several starting values and confirm the same optimum. Disagreement signals multiple local optima.

For likelihood maximisation, the gradient norm should be small relative to $\sqrt{n}$ (the gradient at the truth is asymptotically $O(\sqrt{n})$, so the gradient at the MLE should be much smaller).

4.11. Worked example: lasso via three different solvers

A small comparison: the same lasso problem solved via proximal gradient, coordinate descent (`glmnet`), and the disciplined convex form (`CVXR`). All three should reach the same solution.

```
library(glmnet)
library(CVXR)

set.seed(1)
n <- 200; p <- 50
X <- scale(matrix(rnorm(n * p), n, p))
beta_true <- c(rep(c(2, -1, 0), length.out = p))
y <- as.numeric(X %*% beta_true + rnorm(n))

lambda_val <- 0.1

# 1. glmnet (production-grade)
fit_glmnet <- glmnet(X, y, alpha = 1, lambda = lambda_val,
                     standardize = FALSE, intercept = FALSE)
b_glmnet <- as.numeric(coef(fit_glmnet))[-1]

# 2. CVXR (disciplined convex)
b_var <- Variable(p)
objective <- Minimize(
  sum_squares(y - X %*% b_var) / (2 * n) +
  lambda_val * p_norm(b_var, 1)
)
```

```
prob <- Problem(objective)
res  <- solve(prob)
b_cvxr <- as.numeric(res$getValue(b_var))

# 3. proximal gradient (educational)
b_prox <- prox_grad_lasso(X, y, lambda_val, eta = 0.001,
                          n_iter = 5000)

# all three should agree
max(abs(b_glmnet - b_cvxr))
#> [1] 4e-05            # CVXR is iterative; tolerance is set there
max(abs(b_glmnet - b_prox))
#> [1] 6e-04            # prox grad converges slowly without accelera
```

The three solvers reach the same solution to within their respective tolerances. `glmnet` is fastest in wall-clock time; `CVXR` is most flexible (any disciplined convex problem); proximal gradient is the educational implementation that makes the geometry visible.

4.12. Collaborating with an LLM on advanced optimisation

LLMs handle the textbook material well; they are weaker at recognising which advanced tool fits a specific applied problem.

Prompt 1: tool selection. Describe the objective (smooth or non-smooth, constrained or not, problem size, structure). Ask: 'recommend a solver and an R package, and justify the choice in terms of the problem's structure.'

What to watch for. For non-smooth problems, the LLM should recommend proximal methods, coordinate descent (via `glmnet` or similar), or `CVXR`. For constrained problems, `CVXR` or augmented-Lagrangian methods (`Rsolnp`, `alabama`). For very large n, SGD or L-BFGS. If the LLM defaults to `optim` regardless of problem structure, push for the more specific tool.

Verification. Solve a small instance with the recommended tool and with a reference (e.g., closed form for a small lasso). The two should agree.

Prompt 2: convergence diagnosis. Paste a converged optimiser's output (parameters, gradient norm, function value, iteration count) and ask: 'is this converged? What additional diagnostics should I check?'

What to watch for. For unconstrained, the gradient norm should be small (relative to data scale). For constrained, KKT residuals should be small. For non-convex, multi-start verification. The LLM should mention these; if it accepts $\Delta f < \epsilon$ as sufficient, push.

Verification. Run from a different starting point; check the gradient norm; for constrained problems inspect each KKT residual.

Prompt 3: writing custom proximal operators. Describe a non-smooth penalty (e.g., 'fused lasso with graph-based smoothness') and ask the LLM to write the proximal operator and an ADMM splitting.

What to watch for. Closed-form proximal operators are known for many penalties; the LLM should know the common ones. For a custom penalty, the proximal operator may not be closed-form; an iterative inner solve is required. The LLM should distinguish these cases.

Verification. Test the proximal operator on a small problem where the closed-form answer is known. For ADMM, verify primal and dual feasibility at convergence.

The meta-pattern: advanced optimisation is a deep subject. LLMs are reliable for the standard cases (lasso, ridge, basic ADMM); for novel splittings or custom proximal operators, treat the output as a draft and verify carefully.

4.13. Principle in use

Three habits define defensible advanced-optimisation practice:

1. **Match the tool to the problem structure.** Non-smooth requires proximal methods; constrained requires augmented Lagrangian or interior point; large-n requires stochastic. Generic `optim()` is for smooth unconstrained.

2. **Verify with KKT, not with Δf.** Stationarity, feasibility, complementary slackness. The textbook convergence test is necessary but not sufficient.
3. **Use production solvers for production problems.** `glmnet` for lasso/ridge/elastic net; `CVXR` for general convex; `lbfgs` for large-scale unconstrained. Reach for hand-rolled implementations only when teaching or for problems the production tools do not cover.

4.14. Exercises

1. Implement the soft-thresholding proximal operator and verify it against `glmnet` on a small lasso problem.
2. Use `CVXR` to fit a non-negative least-squares problem. Compare to `nnls::nnls`. Both should give the same answer to high precision.
3. Reformulate ridge regression as a constrained problem; fit it via `CVXR`; verify it matches `glmnet(alpha = 0)`.
4. Take a logistic regression on a moderate dataset ($n = 10{,}000$). Fit via L-BFGS (`optim`), via `glm()` (which uses IRLS), and via SGD with manual step-size schedule. Compare wall-clock times and final coefficients.
5. For a constrained problem in your work, write the KKT conditions explicitly. Identify which constraints are active at the solution; report the corresponding Lagrange multipliers and interpret them as shadow prices.

4.15. Further reading

- (Boyd & Vandenberghe, 2004), *Convex Optimization*. The reference. Free at `web.stanford.edu/~boyd/cvxbook`.
- (Parikh & Boyd, 2014), 'Proximal algorithms'. Free at `web.stanford.edu/~boyd`
- (Boyd et al., 2011), 'Distributed optimization and statistical learning via the alternating direction method of multipliers'. The ADMM monograph.
- (Nocedal & Wright, 2006), the constrained-optimisation chapters give the engineering view (interior point, SQP, augmented Lagrangian).

- The `CVXR` and `glmnet` package vignettes for applied R usage.

5. EM and Its Extensions

5.1. Learning objectives

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

- Derive the EM algorithm from the marginal likelihood for a generic latent-variable model, and identify the E-step and M-step as the two halves of a coordinate ascent on the evidence lower bound.
- Implement EM for two canonical biostatistical cases: Gaussian mixture models and the multinomial-logit formulation of mixed-effects models.
- Recognise when ECM (Expectation-Conditional- Maximisation), MCEM (Monte Carlo EM), or variational EM is more appropriate than ordinary EM.
- Diagnose convergence rigorously: monitor the marginal log-likelihood, watch for slow tails, recognise non- identifiability via flat regions in the M-step.
- Compute observed-information standard errors for EM-fitted models via Louis's method or supplemented EM (SEM).
- Avoid the common antipatterns: stopping at the conditional log-likelihood instead of the marginal, using EM where direct numerical maximisation is faster, missing label-switching in mixture models.

5.2. Orientation

EM is the workhorse of latent-variable estimation. Mixed- effects models, mixture models, hidden Markov models, factor analysis, principal components with missing data, and many missing-data problems all reduce to maximising a likelihood with unobserved or missing components. The EM algorithm is the standard tool for that maximisation.

The introductory volume (**?@sec-mixed-models**) treated mixed-effects models via `lme4`, which uses penalised quasi-likelihood plus a deterministic Laplace approximation rather than EM. EM is needed when the likelihood has missing or latent components that admit a closed-form expectation given the observed data. Variational EM and MCEM extend the framework to cases where that expectation is intractable.

This chapter develops EM from first principles, then treats its modern variants. The mathematical derivation gives the variants their structure; the implementations are R packages or short hand-coded loops, depending on the model.

5.3. The statistician's contribution

EM is famously easy to implement and famously easy to get wrong. The judgements at the centre of this chapter are about recognising when EM is the right tool, when it is converged, and when it is producing the wrong answer for a subtle reason.

EM is not always the fastest. For problems where the joint likelihood is tractable as a function of the parameters, direct maximisation via Newton or BFGS is typically faster than EM. EM's appeal is conceptual (separating the latent from the observed) and its guarantee of monotonic likelihood improvement; its runtime is often slower than direct methods. When you can compute the gradient of the marginal log-likelihood, you can probably skip EM.

Convergence to a local maximum is the rule, not the exception. Mixture models in particular have multiple modes, and EM converges to whichever is closest to the starting point. The fix is multi-start: run from many initial values, keep the best. Without this, the reported solution is whichever starting value happened to land near a good mode.

Label switching matters and is not a bug. A two- component mixture has two equivalent solutions permuting the component labels; neither is preferred. For a single fit this is invisible; for posterior summaries or comparison across multi-start runs, labels must be aligned (post-process by, say, ordering components by their mean). Forgetting this produces artefacts in summaries that look real and are not.

Standard errors require deliberate work. EM produces parameter estimates but does not compute standard errors as a side effect. Louis's method or supplemented EM (SEM) computes the observed information matrix explicitly. Reporting parameter estimates without SEs is common in EM applications; doing it without acknowledging the gap is dishonest reporting.

These judgements distinguish a working EM implementation from one that confidently reports the wrong answer.

5.4. The EM algorithm

Consider a model with parameters θ, observed data $\mathbf{y}$, and unobserved (latent or missing) data $\mathbf{z}$. The complete-data log- likelihood is $\ell_c(\theta; \mathbf{y}, \mathbf{z})$; the marginal log-likelihood (the one we actually want to maximise) is

$$\ell(\theta; \mathbf{y}) = \log \int p(\mathbf{y}, \mathbf{z} \mid \theta)\, d\mathbf{z}.$$

The marginal is intractable in general because of the integral. EM circumvents the integral by alternating between two steps.

E-step. Given the current parameter estimate $\theta^{(k)}$, compute the expected complete- data log-likelihood:

$$Q(\theta \mid \theta^{(k)}) = E_{\mathbf{z}|\mathbf{y},\theta^{(k)}}\left[\ell_c(\theta; \mathbf{y}, \mathbf{z})\right].$$

M-step. Maximise Q to obtain the new estimate:

$$\theta^{(k+1)} = \arg\max_{\theta} Q(\theta \mid \theta^{(k)}).$$

The key theorem (Dempster, Laird, Rubin 1977): each EM iteration weakly increases the marginal log-likelihood, $\ell(\theta^{(k+1)}; \mathbf{y}) \geq \ell(\theta^{(k)}; \mathbf{y})$, with equality only at a stationary point.

The proof uses Jensen's inequality and the Kullback-Leibler divergence. The intuition: the E-step finds the best lower bound on the marginal log-likelihood given the current parameter; the M-step maximises that lower bound. Both halves move uphill on the marginal.

5.4.1. Worked example: two-component Gaussian mixture

Consider $y_i \sim \pi_1 N(\mu_1, \sigma^2) + \pi_2 N(\mu_2, \sigma^2)$ with $\pi_1 + \pi_2 = 1$. The latent indicator z_i takes value 1 if observation i came from component 1 and 0 otherwise.

E-step: given current parameters, compute the responsibility (posterior probability that observation i came from component 1):

$$\gamma_i = P(z_i = 1 \mid y_i, \theta^{(k)}) = \frac{\pi_1 \phi(y_i; \mu_1, \sigma)}{\pi_1 \phi(y_i; \mu_1, \sigma) + \pi_2 \phi(y_i; \mu_2, \sigma)}.$$

M-step: update parameters using the responsibilities:

$$\hat{\pi}_1 = \frac{1}{n} \sum_i \gamma_i, \quad \hat{\mu}_1 = \frac{\sum_i \gamma_i y_i}{\sum_i \gamma_i}, \quad \hat{\mu}_2 = \frac{\sum_i (1 - \gamma_i) y_i}{\sum_i (1 - \gamma_i)}.$$

The variance update is similar, weighted by the responsibilities.

```
em_mixture <- function(y, n_iter = 100, tol = 1e-8) {
  # initialise
  k     <- 2
  mu    <- quantile(y, c(0.25, 0.75))
  sigma <- sd(y)
  pi_1  <- 0.5

  ll_old <- -Inf
  for (k in seq_len(n_iter)) {
    # E-step
    p1   <- pi_1 * dnorm(y, mu[1], sigma)
    p2   <- (1 - pi_1) * dnorm(y, mu[2], sigma)
    gamma <- p1 / (p1 + p2)
```

```
    # M-step
    pi_1   <- mean(gamma)
    mu[1]  <- sum(gamma * y) / sum(gamma)
    mu[2]  <- sum((1 - gamma) * y) / sum(1 - gamma)
    sigma  <- sqrt(
      (sum(gamma * (y - mu[1])^2) +
       sum((1 - gamma) * (y - mu[2])^2)) / length(y)
    )

    # marginal log-likelihood for convergence check
    ll <- sum(log(pi_1 * dnorm(y, mu[1], sigma) +
                  (1 - pi_1) * dnorm(y, mu[2], sigma)))
    if (abs(ll - ll_old) < tol) break
    ll_old <- ll
  }
  list(pi = c(pi_1, 1 - pi_1), mu = mu, sigma = sigma,
       loglik = ll, iter = k)
}

set.seed(1)
y <- c(rnorm(200, 0, 1), rnorm(300, 4, 1))
fit <- em_mixture(y)
fit$mu
#> [1] 0.04 4.06          # close to truth
fit$pi
#> [1] 0.41 0.59          # close to truth
```

The same idea extends to k-component mixtures, mixtures with covariate-dependent component probabilities, and hidden Markov models (where the latent variable is a Markov chain rather than independent indicators).

5.4.2. When EM applies

The EM algorithm is the right tool when:

1. The complete-data likelihood is *much simpler* than the marginal. If the M-step has a closed form given z but the marginal does not, EM converts a hard optimisation into many easy ones.
2. The latent structure is *interpretable*. Mixture components, latent classes, missing-at-random data: the latent variable has substantive meaning.
3. Direct numerical maximisation of the marginal is difficult. If the marginal is smooth and tractable, BFGS is faster than EM and produces standard errors as a side effect.

EM is the *wrong* tool when the complete-data likelihood is nearly as hard as the marginal, or when the M-step has no closed form (in which case ECM or generalised EM becomes necessary; see below).

Check your understanding: EM vs. direct

Question. You want to fit a logistic regression $P(Y = 1 \mid X) = \text{plogis}(X\beta)$. Should you use EM?
Answer.
No, ordinary logistic regression has no latent variables. The likelihood is $\prod_i p_i^{y_i}(1-p_i)^{1-y_i}$ which is directly maximisable by IRLS (what `glm()` does internally) or by Newton on β. EM applies where the model has a latent component: a logistic regression with random intercepts (mixed-effects logistic), a logistic mixture model, a logistic regression with missing covariates, a latent-class analysis. For ordinary logistic regression, EM would be a more complicated way to compute the same answer. The 'do I have a latent variable' question is the diagnostic for whether EM is appropriate.

5.5. ECM: Expectation-Conditional-Maximisation

Ordinary EM requires the M-step to maximise Q jointly over all parameters. For complex models, this joint maximisation may not have a closed form. **ECM** (Meng and Rubin, 1993) replaces the single joint M-step with a sequence of conditional maximisations: maximise over one parameter (or block) at a time, holding the others fixed.

The convergence guarantee carries over: each conditional maximisation increases Q, and so the marginal log- likelihood increases monotonically.

ECM is appropriate when:

- The joint M-step has no closed form but the conditional M-steps do.
- Some parameters have closed-form updates and others require iterative inner maximisation.

A common biostatistical case: mixed-effects models with several variance components. The variance components do not have a joint closed form but each, given the others, does. ECM updates one component at a time.

5.5.1. MCEM: Monte Carlo EM

When the E-step expectation is intractable (no closed form, no efficient deterministic computation), **MCEM** replaces it with a Monte Carlo average:

$$\tilde{Q}(\theta \mid \theta^{(k)}) = \frac{1}{M} \sum_{m=1}^{M} \ell_c(\theta; \mathbf{y}, \mathbf{z}^{(m)})$$

where $\mathbf{z}^{(m)} \sim p(\mathbf{z} \mid \mathbf{y}, \theta^{(k)})$ are samples from the conditional posterior of the latent variables given the data and the current parameter.

MCEM applies when:

- The E-step expectation has no closed form.
- Sampling from $p(\mathbf{z} \mid \mathbf{y}, \theta)$ is feasible (often via MCMC).

The trade-offs:

- The Monte Carlo average has variance $O(1/M)$; convergence to a fixed point is noisy.
- M should typically grow with the iteration to ensure proper convergence (Booth and Hobert, 1999).
- Total cost is the inner sampling cost times the outer iteration count, which can be substantial.

MCEM is the standard for generalised linear mixed models in some packages (e.g., `glmmADMB`, parts of `lme4`'s internals for non-Gaussian outcomes).

5.5.2. Variational EM

When the E-step expectation is intractable but Monte Carlo is too expensive, **variational EM** approximates the conditional posterior of $\mathbf{z}$ by a tractable family $q(\mathbf{z}; \phi)$ and minimises the KL divergence to the true posterior.

The E-step becomes: optimise ϕ to minimise $\mathrm{KL}(q \| p(\mathbf{z} \mid \mathbf{y}, \theta))$. The M-step is unchanged in form but uses q rather than the exact posterior.

Variational EM is the engine behind variational autoencoders, latent Dirichlet allocation, and the variational implementations of Bayesian models in Stan (`variational()`) and PyMC. We treat the broader variational-inference picture in Chapter 8.

The trade-off: the variational approximation is biased (unlike Monte Carlo, which is unbiased). The bias is hard to characterise in advance and depends on the choice of q. Variational EM is the right tool when speed is essential and the bias is acceptable.

5.6. Convergence diagnostics

Three diagnostics characterise EM convergence rigorously.

1. Marginal log-likelihood trace. Plot the marginal log-likelihood against iteration. The trace should be monotonically non-decreasing (the EM theorem); if it ever decreases, there is a bug. The trace should plateau at convergence; the rate of approach gives the convergence rate (linear for ordinary EM, faster for ECM-Newton or accelerated EM variants).

2. Parameter trace. Plot each parameter against iteration. Slow drift in late iterations indicates poor identifiability or a near-flat region of the likelihood; not necessarily a bug, but a signal that standard errors will be large.

3. KKT or gradient at termination. Compute the gradient of the marginal log-likelihood at the candidate solution. It should be small. EM

does not directly produce this gradient; you compute it separately. A large gradient indicates premature termination.

```
# at convergence, check the marginal score
score <- function(theta, y) {
  numDeriv::grad(function(t) marginal_loglik(t, y), theta)
}
max(abs(score(theta_hat, y)))
#> [1] 4e-08            # well-converged
```

5.7. Standard errors via Louis's method

EM does not compute standard errors as a side effect. The observed Fisher information $I(\theta)$ must be computed separately. Louis's method (Louis, 1982) gives:

$$I(\theta) = -E\left[\frac{\partial^2 \ell_c}{\partial \theta^2} \mid \mathbf{y}\right] - \operatorname{Var}\left[\frac{\partial \ell_c}{\partial \theta} \mid \mathbf{y}\right].$$

In words: the observed information equals the expected complete-data Hessian minus the variance of the complete- data score, both evaluated at $\hat{\theta}$ and conditional on the observed data.

The two terms have intuitive content. The first is what the information would be if we observed $\mathbf{z}$ directly (the *complete information*). The second penalises that quantity for the uncertainty introduced by not observing $\mathbf{z}$ (the *missing information*). Their difference is what the data actually tell us about θ.

For models where the conditional expectations are tractable, Louis's method is implementable in a few lines beyond the EM loop. For complex models, **supplemented EM (SEM)** numerically estimates the rate of convergence of EM and uses it to compute the observed information; the implementation is more involved but more general.

5.8. Identifiability and label switching

Mixture models have a fundamental non-identifiability: permuting the component labels gives an equivalent model with the same likelihood. EM converges to one of the equivalent solutions; which one depends on the starting values.

For a single fit, this is invisible (you report *some* labelling). For multiple fits (multi-start runs, bootstrap, posterior simulation), the labels must be aligned. Standard approaches:

- **Order by mean.** After each fit, sort components by $\mu_1 < \mu_2 < ... < \mu_k$. Works when the means separate the components clearly.
- **Order by mixing weight.** Sort by $\pi_1 \leq \pi_2 \leq ....$
- **Hungarian algorithm.** For more complex cases (multivariate components), match across runs by minimising the total distance between matched components. The `label.switching` R package provides this.

For ordinary EM with a single starting point, label switching does not affect the parameter estimates but does affect interpretation. State explicitly which component is which when reporting.

5.9. Worked example: SE for the mixture model

Adding Louis's method to the mixture-model EM loop above:

```
em_mixture_with_se <- function(y, n_iter = 100, tol = 1e-8) {
  fit <- em_mixture(y, n_iter, tol)

  # parameters: pi_1, mu_1, mu_2, sigma
  theta <- c(fit$pi[1], fit$mu, fit$sigma)

  # marginal log-likelihood as a function of theta
  marginal_ll <- function(theta) {
    pi_1   <- theta[1]
    mu     <- theta[2:3]
```

```
    sigma  <- theta[4]
    sum(log(pi_1 * dnorm(y, mu[1], sigma) +
            (1 - pi_1) * dnorm(y, mu[2], sigma)))
  }

  # observed information via numerical differentiation of the score
  H <- -numDeriv::hessian(marginal_ll, theta)
  se <- sqrt(diag(solve(H)))

  list(estimates = theta, se = se,
       ci_lo = theta - 1.96 * se,
       ci_hi = theta + 1.96 * se,
       loglik = fit$loglik)
}

fit <- em_mixture_with_se(y)
fit$estimates
#> [1] 0.41 0.04 4.06 1.00
fit$se
#> [1] 0.022 0.082 0.080 0.041
fit$ci_lo
#> [1] 0.36 -0.12  3.91  0.92
```

For mixture models specifically, `mclust::Mclust` provides production-grade fitting with BIC-based model selection and standard errors via the bootstrap. The hand-coded version above is illustrative.

5.10. Collaborating with an LLM on EM

LLMs are reliable for the textbook EM derivations and for the standard mixture / hidden Markov / missing-data implementations. They are weaker on:

- Recognising when ECM is needed (vs ordinary EM).
- Recognising when MCEM or variational EM is needed (vs direct optimisation of the marginal).

- The bookkeeping of label switching across multiple runs.
- Standard-error computation via Louis or SEM.

Prompt 1: derive the E-step and M-step. Describe the model (likelihood, latent structure). Ask the LLM to derive the E-step expectation and the M-step update, showing intermediate steps.

What to watch for. For standard models (Gaussian mixture, multinomial, Poisson regression with random effects), the LLM should produce correct closed-form updates. For non-standard models, the LLM may produce plausible-looking but wrong derivations. Verify every step against a textbook or published derivation.

Verification. Implement the algorithm; check that the marginal log-likelihood is non-decreasing. If the likelihood ever decreases, the M-step is wrong.

Prompt 2: write convergence diagnostics. Paste the inner EM loop. Ask the LLM to add: a marginal-log- likelihood trace, a parameter trace, and a final gradient check.

What to watch for. The marginal log-likelihood is the key diagnostic; the conditional log-likelihood (complete-data, summed with current responsibilities) is *not*. The LLM may confuse these.

Verification. Run on a problem where the answer is known. The marginal log-likelihood at convergence should match the closed-form value (where one exists).

Prompt 3: compute standard errors. Describe the model and the converged parameter. Ask the LLM to implement Louis's method or supplemented EM.

What to watch for. Louis's method requires the expected complete-data Hessian and the variance of the complete-data score. The LLM may compute the *observed* complete-data Hessian (without the expectation) which is wrong. Verify the expectation is taken over the conditional posterior of the latent variable.

Verification. Compare to bootstrap standard errors on the same problem. The two should agree to within bootstrap noise. If they disagree by an order of magnitude, the Louis implementation is wrong.

The meta-pattern: EM derivations are simple in principle and prone to subtle errors in practice. Using an LLM to draft the derivation is fine; using it to generate the implementation without a manual derivation check is risky.

5.11. Principle in use

Three habits define defensible EM practice:

1. **Choose EM deliberately.** EM is the right tool when the complete-data likelihood is much simpler than the marginal. For models where direct maximisation works, use it.
2. **Multi-start.** Run from several initial values; keep the best by marginal log-likelihood. Local optima are common; multi-start is the cheap defence.
3. **Compute standard errors explicitly.** Louis's method, supplemented EM, or bootstrap. Reporting point estimates without SEs is incomplete; doing so silently is dishonest.

5.12. Exercises

1. Implement EM for a three-component Gaussian mixture with shared variance. Test on synthetic data where the truth is known. Run from 50 random starts; verify the best run reaches the same likelihood each time.
2. Modify the mixture-model EM to allow component- specific variances. Show that the algorithm can collapse a component onto a single observation (variance shrinks to zero, density goes to infinity). Add a regularisation that prevents this.
3. Implement Louis's method standard errors for the two-component mixture. Compare to bootstrap SEs from 1000 resamples. They should agree to within bootstrap noise.
4. Use `mclust::Mclust` to fit a Gaussian mixture model on a real dataset (e.g., `iris[, 1:4]`). Compare BIC across $k = 1, \ldots, 8$. Report the selected number of components and interpret.

5. For a missing-data problem (any biostatistical dataset with a missing covariate), implement EM for the complete-case parameter estimates. Compare to multiple imputation via `mice`. Both should agree to roughly the precision permitted by the data.

5.13. Further reading

- (Dempster et al., 1977), the original EM paper.
- (McLachlan & Krishnan, 2008), *The EM Algorithm and Extensions*, the canonical reference, includes ECM, MCEM, accelerated EM.
- (Meng & Rubin, 1993), Meng and Rubin, the ECM paper.
- (Booth & Hobert, 1999), Booth and Hobert, MCEM with automated M scheduling.
- The `mclust` and `flexmix` package documentation for applied mixture-model fitting in R.

Part III.

Monte Carlo and Bayesian Computation

6. Monte Carlo Methods in Depth

6.1. Learning objectives

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

- Apply importance sampling to estimate integrals or expectations under a target distribution from which direct sampling is impractical.
- Diagnose the failure mode of importance sampling (heavy-tailed weights) and apply truncation or resampling fixes.
- Apply variance-reduction techniques (control variates, antithetic variates, stratified sampling, common random numbers) and quantify their efficiency gains.
- Implement Approximate Bayesian Computation (ABC) for models with intractable likelihoods.
- Implement permutation and randomisation tests correctly, including the choice of test statistic and the handling of ties and one-sided alternatives.
- Account for Monte Carlo error in published results, reporting Monte Carlo standard errors alongside point estimates and choosing M to achieve a target precision.

6.2. Orientation

The introductory volume's simulation chapter (**?@sec-simulation**) treated Monte Carlo as a tool for estimating sampling distributions and method properties. That treatment used the simplest version: simulate from the target distribution, average. The ground covered there is sufficient for most simulation studies and most bootstrap inferences.

This chapter treats Monte Carlo as a more general toolbox for high-dimensional integration, intractable likelihoods, and rigorous error reporting. Importance sampling extends Monte Carlo to targets you cannot sample from directly. Variance-reduction techniques compress the error bar for a given Monte Carlo budget, sometimes by factors of ten or more. ABC handles models where the likelihood is so complex you cannot even write it down but you can simulate from it. Permutation tests provide distribution-free inference whose Monte Carlo error is explicitly bounded.

6.3. The statistician's contribution

Monte Carlo is honest about uncertainty in a way that many other tools are not: every Monte Carlo estimate carries an explicit standard error. The judgements at the centre of this chapter are about choosing the right Monte Carlo method for the problem at hand and interpreting the resulting error correctly.

Match method to problem structure. Direct simulation when the target is easy to sample from. Importance sampling when the target is hard but a related proposal is easy. Variance reduction when the budget is fixed and the goal is the tightest possible error bar. ABC when even the likelihood is intractable. Permutation when distributional assumptions are unwanted. Reaching for the wrong tool produces correct-but-inefficient results in the best case and wrong results in the worst.

Importance sampling fails silently. When the proposal distribution has lighter tails than the target, the importance weights become heavy-tailed and a few extreme weights dominate the estimate. The point estimate looks fine; the variance is enormous. The diagnostic is the effective sample size of the weights; if it is much less than the nominal sample size, the estimator is unstable.

Variance reduction is not free. Control variates, stratified sampling, and Rao-Blackwellisation all require additional structure in the problem. Implementing them on a problem that does not benefit wastes engineering time. Recognising when the structure is present is the analyst's contribution.

Report Monte Carlo standard errors. Every Monte Carlo estimate has an SE; reporting it makes the error bar visible and forces honest

accounting of how many simulations were enough. The convention 'we used 1000 simulations' without an SE is undisciplined; the convention 'we used 1000 simulations, with Monte Carlo SE 0.003 on the reported coverage' is disciplined.

These judgements distinguish a Monte Carlo analysis that documents its uncertainty from one that hides it.

6.4. Importance sampling

To estimate $E_p[h(X)] = \int h(x)p(x)\,dx$ when p is hard to sample from but a related distribution q (the **proposal**) is easy, importance sampling uses

$$E_p[h(X)] = E_q\left[h(X)\frac{p(X)}{q(X)}\right]$$

and estimates the right-hand side by simple Monte Carlo under q. With samples $x_1, \dots, x_M \sim q$ and weights $w_i = p(x_i)/q(x_i)$, the importance-sampling estimator is

$$\hat{\mu}_{\mathrm{IS}} = \frac{1}{M}\sum_{i=1}^{M} w_i h(x_i).$$

In practice the unnormalised target $\tilde{p}(x) \propto p(x)$ is often what you can compute. **Self-normalised importance sampling** uses

$$\hat{\mu}_{\mathrm{SN}} = \frac{\sum_i \tilde{w}_i h(x_i)}{\sum_i \tilde{w}_i}, \quad \tilde{w}_i = \tilde{p}(x_i)/q(x_i)$$

and avoids needing the normalising constant of p. The self-normalised estimator is biased but consistent; the ordinary importance-sampling estimator is unbiased.

6.4.1. Choosing the proposal

The variance of the importance-sampling estimator depends critically on q. The textbook result: minimum variance is achieved when $q(x) \propto p(x)|h(x)|$. In practice, q should resemble p in shape but should have *heavier tails*. A proposal lighter in the tails than the target produces unbounded weights at the target's tail values, and the variance can be infinite.

Common choices:

- **Gaussian proposal centred at the posterior mode.** A good first attempt for log-concave targets.
- **Student-t proposal centred at the posterior mode.** Heavier tails than Gaussian; safer for targets whose tails you do not know.
- **Mixture proposal.** When the target has multiple modes, a mixture of Gaussians or Student-t distributions, one per mode, is robust.

6.4.2. Diagnosing failure: effective sample size

The **effective sample size** of the importance weights:

$$\mathrm{ESS} = \frac{\left(\sum_i w_i\right)^2}{\sum_i w_i^2}.$$

For weights all equal, ESS equals the nominal sample size M. For one weight dominant, ESS approaches 1. A useful rule: if ESS / $M < 0.1$, the importance- sampling estimator is unstable; reconsider the proposal.

```
# importance sampling for E[X^2] under N(0,1), proposing N(0,2)
M <- 1e4
x <- rnorm(M, 0, sqrt(2))             # proposal
w <- dnorm(x, 0, 1) / dnorm(x, 0, sqrt(2))   # weights

# self-normalised estimate
mu_hat <- sum(w * x^2) / sum(w)
mu_hat
#> [1] 0.998              # should be 1
```

```
# effective sample size
ess <- sum(w)^2 / sum(w^2)
ess
#> [1] 7842                # 78% of nominal; healthy
```

Increasing the proposal variance further (e.g., to $\mathrm{Var}(q) = 4$) drops ESS substantially; making the proposal lighter than the target ($\mathrm{Var}(q) = 0.5$) produces infinite-variance weights and the estimator is useless.

6.4.3. Truncation and resampling

For heavy-tailed weights, two practical fixes:

- **Truncation.** Replace w_i with $\min(w_i, w_{\max})$ for some threshold; biases the estimator but stabilises the variance.
- **Resampling (sequential importance sampling).** Sample M values from $\{x_i\}$ with probabilities proportional to w_i; the resampled set is roughly iid from a discrete approximation to p.

Both are workarounds; the cure is usually a better proposal. Truncation and resampling are part of the particle-filter toolkit but are also useful for one-shot importance-sampling problems with bad proposals.

Check your understanding: proposal tail-heaviness

Question. Why does importance sampling fail when the proposal q has *lighter* tails than the target p?
Answer.
The importance weight $w(x) = p(x)/q(x)$ grows like $p(x)/q(x)$ in the tails. If q has lighter tails, $q(x)$ decays faster than $p(x)$ as $|x| \to \infty$, so $w(x)$ grows to infinity. A few samples from the shared bulk get weight near 1; a single rare sample near the tail gets weight near ∞. The estimator is dominated by that one sample; its variance is infinite. The fix is a proposal with *heavier* tails than the target. Student-t proposals are popular exactly because their polynomial tails dominate any exponential-tailed target. The intuition: you would rather oversample the tails (big sample, small weights) than miss

them (small sample, huge weights). The latter is the failure mode; the former is merely inefficient.

6.5. Variance reduction

For a fixed Monte Carlo budget M, variance-reduction techniques produce estimators with smaller variance than crude Monte Carlo. Each requires extra structure in the problem.

6.5.1. Control variates

If $g(X)$ is a function whose expectation $E[g(X)] = \mu_g$ is known, the estimator

$$\hat{\mu}_{\text{cv}} = \frac{1}{M} \sum_i \left(h(x_i) - c \cdot (g(x_i) - \mu_g) \right)$$

is unbiased for $E[h(X)]$ for any c, and the variance is minimised at $c = \text{Cov}(h, g)/\text{Var}(g)$. The optimal c is estimated from the same samples.

The variance reduction equals $\text{Var}(h) \cdot \rho^2$ where $\rho = \text{Cor}(h, g)$. For $|\rho| > 0.9$, the reduction is dramatic.

Applications:

- **Pricing options** with control variate the Black-Scholes formula (known closed form for a related payoff).
- **Estimating** $E[h(X, Y)]$ with control variate $E[h(X, \bar{Y})]$ where $\bar{Y}$ is the marginal mean.
- **Bootstrap estimation** of bias with control variate the linearised estimator.

6.5.2. Antithetic variates

For a symmetric distribution, sample X and use $-X$ together. The pair has correlation -1 for h that is antisymmetric, halving the variance. For h that is neither symmetric nor antisymmetric, the variance reduction is partial.

6.5.3. Stratified sampling

Partition the sample space into strata, sample proportionally within each, and combine. The within-stratum variance is smaller than the overall variance, so the combined estimator has smaller variance than crude Monte Carlo with the same total sample size.

For a one-dimensional uniform variable, stratification into K equal strata reduces variance by a factor of $1 - O(1/K)$; for $K \to \infty$ (one sample per infinitesimal stratum), the reduction is the full variance. In higher dimensions, **Latin hypercube sampling** is the analogue.

6.5.4. Rao-Blackwellisation

If the target expectation can be written $E[h(X, Y)] = E[E[h(X, Y) \mid X]]$, computing the inner conditional expectation in closed form (where possible) reduces variance:

$$\hat{\mu}_{\mathrm{rb}} = \frac{1}{M} \sum_i E[h(x_i, Y) \mid X = x_i].$$

The reduction equals $\mathrm{Var}(E[h \mid X])$, which is non-negative by the law of total variance.

Applications: many Bayesian posterior summaries can be Rao-Blackwellised by integrating out latent variables analytically when their conditional distribution is known.

6.5.5. Common random numbers

When comparing two methods on the same simulated dataset, use *the same* simulated dataset for both. This induces positive correlation between the two methods' estimates, reducing the variance of their difference. We treated this in the introductory volume (**?@sec-simulation**); it is the simplest variance-reduction technique and the highest-leverage in method comparison.

6.6. Approximate Bayesian Computation

Some models are easy to simulate from but hard or impossible to write a likelihood for. Examples: agent-based models in epidemiology, complex stochastic process models, models with many latent variables and no closed-form marginal.

ABC handles such models by replacing likelihood evaluation with summary-statistic comparison:

1. Sample candidate parameters θ' from the prior.
2. Simulate data $\mathbf{y}'$ from the model at θ'.
3. Accept θ' if a summary statistic $T(\mathbf{y}')$ is close to the observed $T(\mathbf{y}_{\text{obs}})$: $\|T(\mathbf{y}') - T(\mathbf{y}_{\text{obs}})\| < \epsilon$.

The accepted θ' values approximate samples from $p(\theta \mid T(\mathbf{y}_{\text{obs}}))$. For sufficient statistics, this is the true posterior; for non-sufficient statistics, it is an approximation whose quality depends on the choice of T and ϵ.

```
# very small ABC example: estimate the mean of a normal
# pretend we cannot evaluate dnorm() but can simulate
y_obs <- rnorm(100, mean = 2.5)
T_obs <- mean(y_obs)

abc_sample <- function(N = 10000, eps = 0.05) {
  # prior
  theta <- runif(N, -5, 5)
  T_sim <- vapply(theta, function(t) mean(rnorm(100, t)), numeric(1))
  accept <- abs(T_sim - T_obs) < eps
```

```
  theta[accept]
}

post <- abc_sample()
mean(post)
#> [1] 2.498
sd(post) / sqrt(length(post))     # SE of posterior mean
#> [1] 0.003
```

Modern variants:

- **Sequential ABC.** Use a sequence of decreasing tolerances, refining the proposal at each stage. The `EasyABC` and `abc` R packages implement this.
- **Regression-adjusted ABC.** After accepting candidates near T_{obs}, fit a local regression of θ on T and adjust each accepted θ as if T had exactly equalled T_{obs}.
- **ABC with summary statistic learning.** Use machine learning to identify informative summary statistics.

ABC is useful but not without limitations:

- Choosing T requires substantive knowledge.
- The tolerance ϵ trades bias for variance; too large gives biased posteriors, too small gives few accepted samples.
- Acceptance rates can be very low; computational cost scales accordingly.

6.7. Permutation tests

Permutation tests are exact (or arbitrarily close to exact) under the null hypothesis of exchangeability, without parametric distributional assumptions.

The basic procedure for testing whether two groups have the same distribution:

1. Compute the test statistic T on the observed data.

2. Permute the group labels among the observations many times; for each permutation, compute T on the relabelled data.
3. The p-value is the fraction of permuted statistics at least as extreme as the observed.

```r
permutation_test <- function(x, y, M = 10000,
                             alternative = "two.sided") {
  n_x <- length(x)
  T_obs <- mean(x) - mean(y)
  pooled <- c(x, y)

  T_perm <- replicate(M, {
    idx <- sample.int(length(pooled))
    mean(pooled[idx[seq_len(n_x)]]) - mean(pooled[idx[-seq_len(n_x)]])
  })

  if (alternative == "two.sided") {
    p_value <- mean(abs(T_perm) >= abs(T_obs))
  } else if (alternative == "greater") {
    p_value <- mean(T_perm >= T_obs)
  } else {
    p_value <- mean(T_perm <= T_obs)
  }

  # Monte Carlo SE on the p-value
  mcse <- sqrt(p_value * (1 - p_value) / M)
  list(statistic = T_obs, p_value = p_value, mcse = mcse, M = M)
}
```

Three considerations matter for practice:

- **Choice of test statistic.** Mean difference is the default for location shifts. For variance differences, use a variance-ratio statistic. For distributional differences without a specific direction, use the Kolmogorov-Smirnov statistic. The test detects whatever the statistic is sensitive to.
- **Handling ties.** Standard sampling-with-permutation handles ties correctly because each permutation is a distinct relabelling. Some

implementations using ranks need adjustment; check the package documentation.
- **One-sided vs. two-sided.** State the alternative explicitly. Reporting a 'p-value of 0.03' without specifying the side invites confusion.

For unbalanced or matched designs (paired samples, stratified data, time-series with autocorrelation), permutation must respect the design: shuffle within strata, within blocks, or in a way that preserves the correlation structure. The `coin` R package handles many of these correctly.

6.8. Monte Carlo error accounting

Every Monte Carlo estimate $\hat{\mu}_M$ has variance $\mathrm{Var}(\hat{\mu}_M) = \sigma^2/M$ where σ^2 is the variance of the underlying samples. The Monte Carlo standard error is

$$\mathrm{MCSE}(\hat{\mu}_M) = \sqrt{\frac{\sigma^2}{M}}.$$

For proportions (e.g., simulated power, Type-I error, coverage probability), $\sigma^2 = p(1-p)$ and

$$\mathrm{MCSE} = \sqrt{\frac{p(1-p)}{M}}.$$

Practical interpretation:

- For an estimated power of 0.8 with $M = 1000$ replicates, MCSE ≈ 0.013. Reporting power as 0.80 with two decimal places is honest; reporting it as 0.802 implies false precision.
- Differences in power smaller than $2 \cdot \mathrm{MCSE}$ are within Monte Carlo noise; do not interpret them.
- To halve the MCSE, quadruple M. For high-precision results, M must be very large.

For continuous estimates (bias, MSE, expected loss), the MCSE is the empirical SD divided by $\sqrt{M}$:

$$\text{MCSE}(\hat{\mu}_M) = s/\sqrt{M}$$

where s is the sample SD of the per-replicate estimates.

The discipline: always report MCSE alongside the point estimate. A Monte Carlo result without an MCSE is incomplete; a Monte Carlo result with an MCSE smaller than the precision being claimed is dishonest.

6.9. Worked example: power simulation with variance reduction

Consider a simulation comparing the power of the t-test and the Wilcoxon rank-sum test under log-normal data. The introductory volume covered this under simple Monte Carlo. Here we add common random numbers (already covered) and then a control variate.

```
set.seed(1)

# parameters
n        <- 30
delta    <- 0.5             # mean shift on the log scale
M        <- 5000            # simulation replicates
alpha    <- 0.05

# simulate; common random numbers across both methods
sim_one <- function() {
  y1 <- rlnorm(n, meanlog = 0, sdlog = 1)
  y2 <- rlnorm(n, meanlog = delta, sdlog = 1)
  c(t = t.test(y1, y2)$p.value < alpha,
    w = wilcox.test(y1, y2)$p.value < alpha)
}

results <- replicate(M, sim_one())
```

```
power <- rowMeans(results)
mcse  <- sqrt(power * (1 - power) / M)

power
#>     t     w
#> 0.62  0.81

mcse
#>      t      w
#> 0.0069 0.0055
```

The Wilcoxon test has higher estimated power (0.81 vs 0.62), well outside the Monte Carlo error ($\pm 2 \cdot 0.0069 = \pm 0.014$). Common random numbers are baked in (we use the same `y1`, `y2` for both tests).

To add a control variate: under the null hypothesis $\delta = 0$, both methods have power equal to α. The deviation from α at $\delta = 0$ is a known-zero quantity that can serve as a control. In this single-δ design the savings would be modest; the technique becomes powerful when comparing power across many δ values.

6.10. Collaborating with an LLM on Monte Carlo

LLMs handle the textbook Monte Carlo material well; they are weaker on importance-sampling diagnostics and on permutation tests for unusual designs.

Prompt 1: importance sampling diagnostics. Paste your importance-sampling code (target, proposal, sample size). Ask: 'compute the effective sample size and diagnose whether this proposal is appropriate.'

What to watch for. The LLM should compute ESS and compare to nominal M. A healthy proposal has ESS / M above 0.5; a marginal one has 0.1 to 0.5; a broken one has under 0.1. The LLM should also identify whether the proposal has heavier or lighter tails than the target.

Verification. Plot the weights; the histogram should not have one or two extreme outliers. If the maximum weight is more than 100 times the median, the proposal is too light in the tails.

Prompt 2: choose a variance-reduction technique. Describe the simulation (target quantity, computational budget, problem structure). Ask: 'recommend one or two variance-reduction techniques for this setting.'

What to watch for. The LLM should match technique to structure: control variate when a related quantity has known expectation; antithetic when the integrand is antisymmetric; stratified when the input distribution admits natural strata; common random numbers when comparing methods. If the LLM recommends every technique without discrimination, push back.

Verification. Implement the recommended technique; compare the empirical variance to that of crude Monte Carlo. The reduction should be proportional to the correlation (or stratification structure, etc.) the technique exploits.

Prompt 3: permutation test design. Describe the data structure (paired, matched, stratified, with covariates, etc.) and ask: 'design a permutation test that respects the design.'

What to watch for. For paired data, permute within pairs only (sign-flipping). For matched designs, permute within matched sets. For stratified data, permute within strata. The LLM may default to the simple two-sample permutation; if so, push for the design-aware version.

Verification. The null distribution of the permuted statistic should be approximately symmetric around zero (for two-sided tests) under truly null data; if it is biased, the permutation respects the wrong exchangeability.

The meta-pattern: Monte Carlo is honest about uncertainty if the user is honest about the failure modes. LLMs accelerate the routine cases; they need human judgement for non-standard designs.

6.11. Principle in use

Three habits define defensible Monte Carlo:

1. **Always compute and report MCSE.** A Monte Carlo estimate without an SE is incomplete. Differences smaller than the MCSE are not real.
2. **Diagnose importance-sampling weights.** ESS, max weight, weight histogram. Without these, importance sampling can fail invisibly.
3. **Use variance reduction when structure permits.** Common random numbers, control variates, stratified sampling: each is a free precision gain when the structure supports it. They are not optional polish; they are part of competent Monte Carlo.

6.12. Exercises

1. Implement importance sampling to estimate the mean of a Cauchy distribution from a Gaussian proposal. The weights are heavy-tailed (the Gaussian has lighter tails); the estimator is unstable. Increase the proposal scale; report when ESS exceeds $0.5M$.
2. Estimate $E[X^2]$ for $X \sim N(0,1)$ via crude Monte Carlo, antithetic variates (X and $-X$ together), and Rao-Blackwellisation (use $E[X^2 \mid X^+] = (X^+)^2$ where $X^+ = |X|$). Compare empirical SDs at fixed M.
3. Implement ABC for a normal-mean inference where $T(\mathbf{y}) = \bar{y}$. Compare the ABC posterior to the analytic posterior under a flat prior. Vary ϵ; document the bias-variance trade-off.
4. Conduct a permutation test for a two-sample comparison on real data. Report the p-value and its MCSE. Compare to the parametric t-test p-value.
5. Run a small simulation study with $M = 1000$ replicates and $M = 10{,}000$ replicates. Verify that the MCSE drops by approximately $\sqrt{10}$.

6.13. Further reading

- (Robert & Casella, 2004), *Monte Carlo Statistical Methods*. The reference. Comprehensive but dense.
- (Owen, 2013), *Monte Carlo theory, methods, and examples*. Free at `statweb.stanford.edu/~owen/mc/`. Excellent applied treatment.

- (Beaumont, 2010), review of ABC for population geneticists, accessible introduction.
- The `coin`, `EasyABC`, `abc`, and `boot` R packages for production-grade implementations.

7. MCMC in Depth

7.1. Learning objectives

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

- Describe Hamiltonian Monte Carlo (HMC) as a Metropolis proposal that uses gradient information to construct long, low-rejection trajectories.
- Identify the No-U-Turn Sampler (NUTS) as the adaptive variant of HMC that automates trajectory length, and explain why it is the default for modern probabilistic programming systems.
- Compute and interpret modern MCMC convergence diagnostics: $\hat{R}$ (rank-normalised), bulk and tail effective sample size, divergent transitions, E-BFMI, and the energy diagnostic.
- Recognise the reparameterisation patterns that fix hierarchical-prior pathologies: the non-centred parameterisation, the change-of-variable Jacobian, and the standardisation of input variables.
- Run multiple chains in parallel correctly, with reproducible RNG and proper diagnostics across chains.
- Diagnose failures: chains stuck on different modes, funnel pathologies, ill-conditioned posteriors, and what each looks like in `bayesplot::mcmc_pairs()`.

7.2. Orientation

The introductory volume's Bayesian chapter (**?@sec-bayesian**) introduced MCMC at the Metropolis-Hastings and Gibbs-sampler level. That treatment shows the mathematical structure but produces samplers that are slow on realistic posteriors. Modern applied Bayesian work uses Hamiltonian

Monte Carlo (HMC), or its adaptive descendant the No-U-Turn Sampler (NUTS), via Stan, PyMC, NumPyro, or the R interfaces `rstan`, `cmdstanr`, `rstanarm`, and `brms`.

This chapter treats HMC and NUTS in depth: enough mechanics to understand why they work and what their diagnostics mean, plus the practical bookkeeping that distinguishes a converged chain from one that looks converged. The next chapter (Chapter 8) covers the application layer (Stan, posterior workflows, LOO/WAIC); this chapter covers the algorithmic and diagnostic layer.

7.3. The statistician's contribution

Modern MCMC tools handle the algorithmic mechanics automatically. The judgements at the centre of this chapter are about reading diagnostics correctly, recognising pathological posterior shapes from their diagnostic signatures, and making the parameterisation choices that determine whether the sampler can converge at all.

$\hat{R}$ near 1 is necessary, not sufficient. A chain with $\hat{R} = 1.001$ may still have low effective sample size, divergences, or trapped behaviour on a multimodal posterior. The full diagnostic suite (divergences, E-BFMI, ESS bulk and tail, trace plots, pair plots) is the verification; $\hat{R}$ alone is insufficient.

Reparameterisation is a modelling choice. A hierarchical model with a centred parameterisation ($\mu_j \sim N(\mu, \sigma)$) has a different posterior geometry from the same model with a non-centred parameterisation ($\mu_j = \mu + \sigma z_j, z_j \sim N(0, 1)$). The mathematics is the same; the sampler's behaviour is not. Knowing which parameterisation matches your data's information level is the difference between a fast, clean fit and an MCMC chain that looks converged but never explores the funnel of the variance parameter.

Divergences are not a tuning issue. When NUTS reports divergent transitions, the standard advice is 'increase `adapt_delta`'. That treats the symptom. The underlying cause is geometry: the posterior has a region the sampler cannot integrate accurately. Increasing `adapt_delta` may hide the divergences but not the geometry; the chain still mis-samples the

difficult region. The cure is reparameterisation, prior tightening, or model simplification.

Multiple chains, multiple starting points. Convergence of one chain to a fixed point does not imply convergence to *the* posterior. Multiple chains starting from dispersed values, with $\hat{R}$ across chains as the consistency check, is the protection against multimodality and non-stationarity.

These judgements are what distinguish a converged Bayesian analysis from one that mis-reports its uncertainty.

7.4. Hamiltonian Monte Carlo

HMC is a Metropolis-Hastings sampler whose proposal combines the parameter θ with an auxiliary momentum variable $\mathbf{p}$ and uses the Hamiltonian dynamics on the joint $(\theta, \mathbf{p})$ space to propose long, low-rejection moves.

The Hamiltonian for a target $p(\theta)$ is

$$H(\theta, \mathbf{p}) = -\log p(\theta) + \frac{1}{2}\mathbf{p}^T M^{-1}\mathbf{p}$$

where M is a (positive-definite) **mass matrix**, often chosen to roughly equal the posterior covariance. The two halves of H are the **potential energy** (the negative log target) and the **kinetic energy** (a quadratic in $\mathbf{p}$).

Hamilton's equations describe how $(\theta, \mathbf{p})$ evolves under H:

$$\frac{d\theta}{dt} = \frac{\partial H}{\partial \mathbf{p}} = M^{-1}\mathbf{p}, \quad \frac{d\mathbf{p}}{dt} = -\frac{\partial H}{\partial \theta} = \nabla \log p(\theta).$$

The continuous-time evolution preserves H exactly. In practice, we discretise via the **leapfrog integrator**:

$$\begin{aligned}\mathbf{p}^{(t+\epsilon/2)} &= \mathbf{p}^{(t)} + \frac{\epsilon}{2}\nabla \log p(\theta^{(t)}) \\ \theta^{(t+\epsilon)} &= \theta^{(t)} + \epsilon M^{-1}\mathbf{p}^{(t+\epsilon/2)} \\ \mathbf{p}^{(t+\epsilon)} &= \mathbf{p}^{(t+\epsilon/2)} + \frac{\epsilon}{2}\nabla \log p(\theta^{(t+\epsilon)}).\end{aligned}$$

Take L leapfrog steps; propose $(\theta^{(t+L\epsilon)}, -\mathbf{p}^{(t+L\epsilon)})$ as the Metropolis candidate; accept with probability $\min(1, \exp(H_{\text{old}} - H_{\text{new}}))$. The discretisation error in the leapfrog integrator is small but nonzero, so the acceptance probability is slightly below 1; the deviation from 1 is what makes the sampler proper.

The two tuning knobs are:

- **Step size ϵ.** Smaller produces more accurate trajectories and higher acceptance, but needs more steps to cover the same distance.
- **Trajectory length L.** Longer trajectories produce proposals farther from the current state, reducing autocorrelation, but cost more gradient evaluations.

Tuning these by hand is delicate; NUTS automates it.

7.4.1. The No-U-Turn Sampler (NUTS)

NUTS (Hoffman and Gelman, 2014) extends HMC with two adaptations:

1. **Automatic trajectory length.** NUTS extends the trajectory in random directions until it begins to double back on itself (forms a U-turn) or reaches a maximum tree depth (typically 10, giving up to $2^{10} - 1 = 1023$ leapfrog steps). The trajectory automatically stops just before the proposal would loop back.
2. **Automatic step-size adaptation.** During warmup, NUTS adjusts ϵ to achieve a target acceptance rate (default 0.8 in Stan).

The result is a sampler with no user-tunable parameters except the target acceptance rate (`adapt_delta` in Stan, defaulting to 0.8). For most well-formulated posteriors, NUTS produces effective samples at hundreds to thousands per second, hundreds of times faster than random-walk Metropolis.

NUTS is the default in Stan, PyMC, NumPyro, and via the R wrappers `rstanarm`, `brms`, `rstan`, and `cmdstanr`. For routine Bayesian analyses, you do not implement NUTS yourself; you use one of these.

7.5. Modern convergence diagnostics

7.5.1. $\hat{R}$ (rank-normalised, split, with folding)

The classical Gelman-Rubin diagnostic compares within- chain to between-chain variance. The 2021 update by Vehtari et al. introduces three improvements:

- **Rank normalisation.** Replace samples by their ranks, then transform via Φ^{-1} to a normal scale. Robust to heavy tails and asymmetric posteriors.
- **Split.** Each chain is split in two halves; $\hat{R}$ is computed across the resulting $2C$ chains rather than the original C. Detects within-chain non-stationarity.
- **Folding.** Compute $\hat{R}$ on $|x - \text{median}(x)|$ in addition to x itself. Detects scale (variance) non-stationarity.

The threshold is $\hat{R} < 1.01$ for both the regular and folded versions. The convention $\hat{R} < 1.1$ from older literature is too lax for modern applied Bayesian work; use 1.01.

```
library(posterior)
draws <- as_draws_array(stan_fit)
summarise_draws(draws, default_convergence_measures())
#> # A tibble: 5 × 4
#>   variable    rhat ess_bulk ess_tail
#>   <chr>      <dbl>    <dbl>    <dbl>
#> 1 b_intercept 1.00     2840     2120
#> 2 b_x         1.00     2950     2470
#> 3 sigma       1.00     1980     2350
#> 4 ...
```

7.5.2. Effective sample size

ESS is the number of independent samples that would give the same standard error as the autocorrelated MCMC samples. **ESS bulk** measures the effective sample size for typical-value summaries (mean, median); **ESS tail** measures it for tail summaries (95% credible intervals). The convention: at least 400 in each is required for reliable summaries. Below that, the posterior interval bounds are noisy.

For most posteriors, ESS bulk and tail are similar. When they differ substantially, the tail of the posterior is slow-mixing; tail-quantile reporting is unreliable.

7.5.3. Divergent transitions

A divergence in NUTS occurs when the leapfrog integrator fails to track the true Hamiltonian dynamics, typically because the integrator step size is too large for a high-curvature region. Divergences indicate that NUTS is not exploring some part of the posterior accurately; the resulting samples are biased away from that region.

Default Stan reports divergence count as part of the warning system. Any positive count is concerning; double digits are a serious problem.

The textbook fix: increase `adapt_delta` from 0.8 to 0.95 or 0.99. This reduces the step size, which often fixes the immediate problem. But this treats the symptom; the underlying cause is usually a posterior geometry that requires a different parameterisation (see below).

7.5.4. E-BFMI

The Energy-based Bayesian Fraction of Missing Information diagnoses whether the chain is exploring the energy distribution efficiently. Values below 0.3 indicate problems; the canonical example is the funnel pathology, where the chain repeatedly fails to sample the bottom of the funnel.

```
# Stan reports E-BFMI per chain
rstan::check_energy(fit)        # warning if any chain has E-BFMI < 0.2
```

7.5.5. Pair plots

For diagnosing geometric pathologies, the most useful visual is a pair plot coloured by divergence:

```
library(bayesplot)
mcmc_pairs(stan_fit, pars = c("mu", "sigma", "tau"),
           np = nuts_params(stan_fit))
```

Divergent transitions cluster in regions of the posterior the sampler cannot integrate accurately. In hierarchical models, divergences typically cluster in the bottom of the funnel (small τ, small group means).

7.5.6. Trace plots

The classical trace plot shows the sampled value of each parameter against iteration. A converged chain looks like white noise around a stable mean. Chains that drift, chains that get stuck, and chains that explore different regions on different runs are visually obvious in a trace plot.

```
mcmc_trace(stan_fit, pars = c("mu", "sigma"))
```

7.6. Reparameterisation: centred vs. non-centred

The single most common reason a hierarchical Bayesian model produces divergences is a parameterisation mismatch with the data's information level. The canonical example: a hierarchical normal with group means $\mu_j \sim N(\mu, \tau)$ and observation-level likelihood $y_{ij} \sim N(\mu_j, \sigma)$.

Centred parameterisation (CP):

```
parameters {
  real mu;
  real<lower=0> tau;
  real<lower=0> sigma;
```

```
  vector[J] mu_j;
}
model {
  mu_j ~ normal(mu, tau);
  for (j in 1:J)
    y[j] ~ normal(mu_j[j], sigma);
}
```

Non-centred parameterisation (NCP):

```
parameters {
  real mu;
  real<lower=0> tau;
  real<lower=0> sigma;
  vector[J] z_j;       // standard normal
}
transformed parameters {
  vector[J] mu_j = mu + tau * z_j;
}
model {
  z_j ~ normal(0, 1);
  for (j in 1:J)
    y[j] ~ normal(mu_j[j], sigma);
}
```

The two parameterise the *same* model. The posterior on (μ, τ, μ_j) is identical in distribution. But the posterior geometry differs:

- Under CP, the posterior on (τ, μ_j) has a funnel shape: when τ is small, μ_j is squeezed close to μ, producing high curvature in the joint posterior.
- Under NCP, the posterior on (τ, z_j) is roughly spherical: z_j stays around its prior $N(0, 1)$ regardless of τ.

When CP works: when the data are informative about each μ_j (many observations per group). The posterior on μ_j is then dominated by the data, not the prior, and the funnel is mostly absent.

When NCP works: when the data are weakly informative about each μ_j (few observations per group). The posterior is dominated by the prior; the funnel is present in CP but absent in NCP.

The general rule: start with NCP for any hierarchical-prior parameter that is not strongly identified by the data. Switch to CP only when both work and CP gives more efficient sampling.

Check your understanding: when does CP fail?

Question. Your hierarchical model has $J = 50$ groups, with observations per group ranging from 2 to 10. Stan reports 47 divergences out of 4000 post-warmup samples. You increase `adapt_delta` to 0.99 and the divergences drop to 8. Should you publish?
Answer.
No. The divergences dropped because the smaller step size made the sampler more careful, but the underlying geometry is still bad. The few-observations-per-group setting is exactly the case where CP fails: the data do not pin down each μ_j, so the posterior has a funnel shape that NCP would handle. The fix is to reparameterise to NCP: $\mu_j = \mu + \tau z_j$ with $z_j \sim N(0, 1)$. This typically eliminates the divergences and produces a faster sampler. Publishing with `adapt_delta = 0.99` and 8 remaining divergences means the bottom of the funnel was never properly sampled; the variance estimate τ is biased toward larger values, and the group-level estimates are biased toward the global mean. Re-fit with NCP, verify zero divergences, then report.

7.7. Other reparameterisations

Beyond centred / non-centred, several other reparameterisations recur:

Standardise predictors. A predictor measured in millions has a coefficient that takes values in 10^{-6}; one in tenths has a coefficient in 1. Mixing both in one regression produces a posterior on β with very different scales per coefficient, which the sampler's mass matrix has trouble adapting to. Standardising every continuous predictor to mean 0, SD 1 puts coefficients on comparable scales and aids sampling.

Log-transform positive variables. A posterior on $\sigma > 0$ is more naturally sampled in the unconstrained $\log \sigma$ space. Stan automatically applies the transform for `<lower=0>` declarations and includes the Jacobian; PyMC and similar do the same. Hand-rolled MCMC must include the change-of-variable Jacobian explicitly.

Logit-transform probabilities. Probabilities live in $(0, 1)$; sampling on the logit scale removes the boundary effects.

Cholesky parameterisation of covariance matrices. For multivariate-normal random effects with covariance Σ, parameterise via the lower-triangular Cholesky factor L such that $\Sigma = LL^T$. Stan's `lkj_corr_cholesky` prior works on the Cholesky factor of the correlation matrix. This avoids the positive-definiteness constraint and produces substantially better sampling.

7.8. Parallel chains

Modern Bayesian software runs multiple chains in parallel by default. The standard configuration:

- **4 chains** (Stan default), each on its own CPU core.
- **1000 warmup + 1000 sampling iterations** per chain (Stan default).
- **Random initialisation** dispersed in the parameter space (Stan default: uniform on $[-2, 2]$ in the unconstrained space).

The four chains' worth of samples are pooled for inference; their consistency provides the $\hat{R}$ diagnostic.

For long-running models, `cmdstanr` and `rstan` parallelise across chains automatically. To extract maximum parallelism, set `cores = 4` (or as many as you have).

```
library(cmdstanr)
fit <- stan_model$sample(
  data    = stan_data,
  chains  = 4,
  parallel_chains = 4,
```

```
  iter_warmup   = 1000,
  iter_sampling = 1000,
  seed    = 47
)
```

For RNG reproducibility, set `seed`. Each chain receives a derived seed; running with the same `seed` produces identical samples. Without setting `seed`, results are not reproducible.

7.9. Worked example: Eight Schools

The Eight Schools dataset (Rubin, 1981; Gelman et al., 2013) is the canonical hierarchical-Bayesian example. The data: estimated treatment effects of an SAT-prep coaching intervention at eight schools, with their standard errors:

```
schools_data <- list(
  J = 8,
  y = c(28, 8, -3, 7, -1, 1, 18, 12),
  sigma = c(15, 10, 16, 11, 9, 11, 10, 18)
)
```

A hierarchical model with non-centred parameterisation:

```
data {
  int<lower=0> J;
  vector[J] y;
  vector<lower=0>[J] sigma;
}
parameters {
  real mu;
  real<lower=0> tau;
  vector[J] eta;
}
transformed parameters {
```

```
  vector[J] theta = mu + tau * eta;
}
model {
  mu  ~ normal(0, 5);
  tau ~ cauchy(0, 5);
  eta ~ normal(0, 1);
  y   ~ normal(theta, sigma);
}
```

Fit and check diagnostics:

```
library(cmdstanr)
mod <- cmdstan_model('eight-schools-ncp.stan')
fit <- mod$sample(data = schools_data, chains = 4,
                  parallel_chains = 4, seed = 47, refresh = 0)

fit$summary()                 # tibble of point estimates, ESS, R-hat
fit$diagnostic_summary()      # divergences, E-BFMI, max tree depth
```

Under NCP, this model samples cleanly with zero divergences, $\hat{R}$ near 1, and ESS in the thousands in seconds. Re-running with the centred parameterisation typically produces dozens of divergences.

7.10. Collaborating with an LLM on MCMC

LLMs handle Stan and PyMC code competently, including both centred and non-centred parameterisations. They are less reliable on the diagnostic interpretation: specifically, on translating diagnostic output into either 'this is fine, publish' or 'this is broken, fix the parameterisation'.

Prompt 1: write Stan code with appropriate parameterisation. Describe the model and the data's information level (how many observations per group). Ask: 'write Stan code for this hierarchical model. Choose between centred and non-centred parameterisation based on the information level.'

What to watch for. Few observations per group (under roughly 5–10) calls for NCP. Many observations per group (50+) is the regime where CP works. The LLM should make this choice and explain it. If it picks one without explanation, ask why.

Verification. Fit the model with the chosen parameterisation; check for divergences. If divergences appear with the chosen parameterisation, try the alternative. The right one usually has zero divergences.

Prompt 2: interpret diagnostic output. Paste the output of `fit$diagnostic_summary()` (divergences, E-BFMI, max tree depth) and `fit$summary()` (R-hat, ESS). Ask: 'is this chain converged? What should I worry about?'

What to watch for. The LLM should check $\hat{R} < 1.01$ on every parameter, ESS bulk and tail above 400 on every parameter, divergences = 0, E-BFMI > 0.3, and max tree depth not saturated. If any fail, the diagnosis depends on which: divergences point to parameterisation or `adapt_delta`; low ESS points to chain length; saturated tree depth points to step size.

Verification. Run with adjusted settings (more iterations, NCP, higher `adapt_delta`) and confirm the diagnostic improves.

Prompt 3: diagnose a specific pathology. Paste a pair plot description (or the actual `mcmc_pairs` output) showing divergences clustered in a particular region. Ask: 'what kind of geometric pathology produces this pattern, and what reparameterisation would fix it?'

What to watch for. Divergences clustered at small τ in a hierarchical model: funnel pathology, fix with NCP. Divergences clustered at the boundary of a constrained parameter (e.g., $\rho \to 1$ in a correlation): scale issue, consider tighter prior. Divergences along a thin diagonal: posterior is ill-conditioned, need to reparameterise to break the correlation.

Verification. Apply the recommended reparameterisation; refit; verify divergences are gone.

The meta-pattern: modern MCMC's diagnostics are informative if you read them; LLMs help interpret them faster than you would alone. The judgement is whether to trust the LLM's diagnosis enough to publish, which requires checking it against the actual posterior behaviour.

7.11. Principle in use

Three habits define defensible MCMC use:

1. **Read the full diagnostic suite.** $\hat{R}$, ESS (bulk and tail), divergences, E-BFMI. Any failure is a reason to investigate before reporting.
2. **Reparameterise before retuning.** When divergences appear, the first move is non-centred parameterisation (for hierarchical models) or input standardisation (for regression). Tightening `adapt_delta` is the second move, not the first.
3. **Multiple chains, dispersed starts.** Four chains with random initialisation are the protection against multimodality and against chains that look converged in isolation but disagree across chains.

7.12. Exercises

1. Fit the Eight Schools model with both centred and non-centred parameterisations. Compare divergences, ESS, and posterior summaries. Confirm the reparameterisation reproduces the same posterior in distribution but with cleaner sampling.
2. Construct a hierarchical model where the data are highly informative (many observations per group) and one where they are weakly informative (few). For each, compare CP and NCP. The advantage of NCP should diminish in the data-rich regime.
3. Implement the leapfrog integrator and a basic HMC sampler in R. Test on a 2D Gaussian target. Compare to random-walk Metropolis at fixed computational budget.
4. Use `bayesplot::mcmc_pairs` with divergence colouring on a hierarchical model that produces divergences. Visually identify the funnel; describe its location in (τ, μ_j)-space.
5. Run a Stan model with `seed = 47` four separate times. Confirm that all four runs produce identical samples (same `fit$draws()`). Then run without a seed and observe non-reproducibility.

7.13. Further reading

- (Hoffman & Gelman, 2014), Hoffman and Gelman, the NUTS paper.
- (Betancourt, 2017), Betancourt, 'A conceptual introduction to Hamiltonian Monte Carlo'. The clearest geometric treatment.
- (Vehtari et al., 2021), Vehtari et al., the modern $\widehat{R}$ paper.
- The Stan User's Guide and the `bayesplot`, `posterior`, and `cmdstanr` package documentation.

8. Modern Bayesian Computation

8.1. Learning objectives

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

- Distinguish probabilistic programming languages (Stan, PyMC, Turing) from front-end packages (`rstanarm`, `brms`, `cmdstanr`) and choose between them for a given task.
- Implement a Bayesian regression model in `brms` and inspect the generated Stan code to verify that priors and likelihood match your intent.
- Diagnose a fit with the four-quadrant workflow: convergence ($\hat{R}$, ESS), posterior predictive checks, prior predictive checks, and influential observations (Pareto-$\hat{k}$).
- Compute and interpret leave-one-out cross-validation (`loo`) and WAIC for model comparison, recognising when Pareto-$\hat{k}$ warns that the importance-sampling approximation has failed.
- Decide when full HMC, when variational inference (ADVI, Pathfinder), and when a Laplace or INLA approximation is the right tool for the problem at hand.
- Articulate Gelman et al.'s Bayesian workflow as a sequence of falsifiable checks rather than a one-shot fit-and-summarise.

8.2. Orientation

The introductory SCAI volume showed how to fit a Bayesian model in `brms` and how to read a posterior. This chapter takes the next step: how the fit is produced, how to verify it, how to compare competing models, and what to do when HMC is too slow to be practical.

Three currents shape modern Bayesian computation. First, **probabilistic programming languages (PPLs)**, Stan, PyMC, Turing, NumPyro, separate model specification from inference algorithm. You write the joint density; the compiler produces samplers and gradient code. Second, **the Bayesian workflow** (Gelman et al., 2020) reframes inference as iterative model-criticism rather than a single posterior summary. Third, **scalable approximations**, variational inference, Pathfinder, INLA, Laplace, extend Bayesian computation to problem sizes where MCMC is hopeless.

The chapter follows that arc. We start with the PPL stack and the front-end packages most users meet first (`rstanarm`, `brms`). We then turn to model checking and comparison, where `loo` and `bayesplot` do most of the heavy lifting. Finally we cover variational inference and adjacent approximations, with explicit attention to when each fails.

8.3. The statistician's contribution

LLMs can write Stan code, run `brms` calls, and produce `loo` comparisons. What they cannot do is decide which model is actually relevant to the scientific question, which prior encodes domain knowledge honestly, and when a passing diagnostic is hiding a deeper failure.

(Judgement 1.) Priors are part of the model, not nuisance. Default priors in `brms` are weakly informative and adequate for many regression problems, but they are not universally appropriate. A logistic regression on rare-event data with default flat priors on coefficients will produce posteriors driven by the few events present and will be sensitive to the parameterisation. The statistician decides what the prior should encode: a rough plausible range for treatment effects on the log-odds scale, a known boundary on a variance component, a hierarchical structure that shares information across centres. None of these decisions are visible from the data alone, and an LLM that proposes 'use the default priors' is dodging the question rather than answering it.

(Judgement 2.) A passing diagnostic is necessary, not sufficient. $\hat{R} = 1.00$ and ESS in the thousands tell you the chains agree on the posterior they have explored. They do not tell you the chains explored the right posterior. Two pathologies in particular evade automatic checks:

multimodal posteriors where chains separately settle in different modes (each chain looks fine; pooled diagnostics flag nothing if all modes are visited proportionally), and posteriors with regions of high curvature that the sampler silently skips. The defence is **posterior predictive checking**, does the fitted model produce data that look like the observed data on quantities you care about?, and **prior predictive checking**, does the model produce remotely plausible data before seeing any observations? These are scientific questions that the statistician frames.

(Judgement 3.) Model comparison is a means, not an end. `loo_compare()` will return a difference in expected log-pointwise predictive density (elpd) with a standard error and rank competing models. It will not tell you whether either model is good enough for the decision the analysis informs. A Bayesian workflow that ends at 'model B is preferred by 4 elpd units' has skipped the step where the statistician asks whether B's predictions are in the right ballpark, whether it extrapolates safely, and whether its assumptions are defensible to a reviewer. LLMs are strong at running `loo_compare`; they are weak at noticing when the answer is 'neither model is fit for purpose.'

These judgements are what distinguish an analysis that survives review from one that does not.

8.4. The probabilistic programming stack

Bayesian computation in 2026 is layered. From bottom to top:

Inference engines. Stan (Carpenter et al., 2017) is a C++ library implementing HMC/NUTS, ADVI, Pathfinder, and a Laplace approximation. PyMC (Salvatier et al., 2016) is a Python library built on PyTensor that offers similar samplers. Turing (Ge et al., 2018) is a Julia counterpart. NumPyro is JAX-based and prized for speed on hierarchical models. All four expose roughly the same primitives: a domain-specific syntax for declaring parameters and a log-density, plus algorithms that consume that density.

R interfaces. `cmdstanr` is the recommended R interface to Stan; `rstan` is older and bundles its own Stan version. `reticulate` plus `cmdstanpy` lets you reach Stan from R via Python, but `cmdstanr` is the simpler path.

PyMC is reachable from R only via `reticulate`; if your team is R-first, Stan is the natural choice.

Front-end packages. `rstanarm` (Goodrich et al., 2024) ships with pre-compiled Stan models for common GLMs and GLMMs and is the fastest way to fit a Bayesian regression in R. `brms` (Bürkner, 2017) generates Stan code on the fly from an R formula and supports a much wider class of models, multivariate outcomes, distributional regression, nonlinear models, missing-data imputation, time-varying effects via splines and Gaussian processes. `brms` is the workhorse of applied Bayesian regression in R.

Post-processing packages. `posterior` standardises the draws-as-array data structure across engines. `bayesplot` provides diagnostic and posterior-predictive-check plots. `tidybayes` exposes posterior draws as tidy tibbles, playing nicely with the rest of the tidyverse. `loo` implements approximate leave-one-out cross-validation.

The implication for your workflow: **most applied work should start in `brms` or `rstanarm`, drop to raw Stan only when the model cannot be expressed in the front-end syntax, and never reimplement the post-processing stack.**

Check your understanding: choosing the layer

Question. A collaborator asks you to fit a Bayesian two-level random-intercept logistic regression with about 20 centres and 8000 patients, with a known prior on the treatment effect informed by an earlier trial. Which layer of the stack is the right starting point?

Answer.

`brms` is the right starting point. The model is a standard GLMM with an informative prior on a single coefficient: both expressible in `brm(y ~ trt + (1 | centre), family = bernoulli(), prior = prior(normal(0.3, 0.1), class = b, coef = "trt"), ...)`. Dropping to raw Stan would be premature. `rstanarm`'s `stan_glmer()` would also work, but the ability to inspect and modify the generated Stan code is more straightforward in `brms` if the prior must later be replaced with something non-standard. Inspect the generated code with `make_stancode()` before running the fit.

8.5. A brms fit, end to end

The minimum viable fit has six steps: declare the model, inspect the generated code, run prior predictive checks, fit, run posterior diagnostics, and run posterior predictive checks. We illustrate with a simulated logistic regression with a known random-intercept structure.

```
library(brms)
library(bayesplot)
library(loo)

set.seed(228)
n_centres <- 20
n_per <- 80
sim <- tibble::tibble(
  centre = rep(1:n_centres, each = n_per),
  x = rnorm(n_centres * n_per)
)
b0 <- rnorm(n_centres, 0, 0.6)
sim$y <- rbinom(
  nrow(sim), 1,
  plogis(-0.5 + 0.4 * sim$x + b0[sim$centre])
)

priors <- c(
  prior(normal(0, 1), class = "b"),
  prior(normal(0, 2), class = "Intercept"),
  prior(student_t(3, 0, 1), class = "sd")
)

fit <- brm(
  y ~ x + (1 | centre),
  data = sim,
  family = bernoulli(),
  prior = priors,
  chains = 4, cores = 4, iter = 2000,
  backend = "cmdstanr",
  sample_prior = "yes"
```

```
)
```

`sample_prior = "yes"` retains prior draws so we can run prior predictive checks side by side with posterior checks.

Step 1: inspect the generated code. `make_stancode(fit)` prints the Stan program `brms` produced. Read it. If the priors do not match what you wrote, or the parameter names are not what you expected, fix the formula before fitting. Skipping this step is the single most common source of silent specification errors with `brms`.

Step 2: prior predictive checks. Before fitting, draw samples from the prior alone and pass them through the likelihood. Plausible? Implausible? `pp_check(fit, type = "bars", prefix = "ppd")` plots prior predictive draws against the data. If the priors imply outcomes far from anything the collaborator considers possible, tighten them.

Step 3: convergence diagnostics. `summary(fit)` reports $\hat{R}$ and ESS bulk/tail per parameter; flag any $\hat{R} > 1.01$ or ESS bulk < 400. `mcmc_trace()` and `mcmc_pairs()` from `bayesplot` reveal divergent transitions and funnel pathologies in hierarchical models.

Step 4: posterior predictive checks. `pp_check(fit, type = "bars")` for binary or count outcomes; `pp_check(fit, type = "dens_overlay")` for continuous. Look for systematic discrepancies between simulated and observed data, the model under-predicts large values, mis-locates the mean, under-disperses the tails. These are the structural failures $\hat{R}$ will not catch.

Step 5: targeted posterior predictive checks. Beyond 'does the marginal density match,' ask: does the model reproduce the *quantity the analysis is about*? If the report quotes the centre-level event rate, run `pp_check(fit, type = "stat_grouped", group = "centre", stat = "mean")` and check the centre means line up. A model can match the marginal and miss the conditional badly.

Step 6: model criticism. `loo(fit)` returns the leave-one-out elpd estimate and Pareto-$\hat{k}$ diagnostics for each observation. Observations with

$\hat{k} > 0.7$ indicate that the importance-sampling approximation to leave-one-out is unreliable for that point, usually because the point is influential. Refit those points exactly with `reloo()` if the comparison hinges on them.

Check your understanding: failing the right check

Question. A `brms` fit returns $\hat{R} = 1.00$, ESS bulk in the low thousands, no divergent transitions, and `pp_check(fit, type = "dens_overlay")` shows the simulated densities tracking the observed density well. The collaborator declares the model fit and ready for inference on a centre-specific event rate. What additional check is non-negotiable before agreeing?
Answer.
`pp_check(fit, type = "stat_grouped", group = "centre", stat = "mean")`. The marginal density check confirms the model matches the overall outcome distribution; it does not confirm the model matches centre-specific rates. If the report quotes a centre-specific quantity, the centre-level predictive check is the relevant one. A passing marginal check with a failing grouped check is a common signature of mis-specified random-effects structure.

8.6. Model comparison: `loo` and WAIC

Cross-validation answers the question 'which model predicts new observations better.' True K-fold CV is expensive when each fit takes hours. **Pareto-smoothed importance-sampling LOO** (Vehtari et al., 2017) approximates exact LOO from a single fit by reweighting posterior draws, it is the default in `loo`.

`loo_compare(loo(fit1), loo(fit2))` returns the difference in expected log-pointwise predictive density and its standard error. The convention is to interpret the comparison as decisive only when $|\Delta\text{elpd}|$ is several times its SE; small differences are often within sampling noise.

WAIC (Watanabe, 2010) is a related criterion computable from the same machinery; it generally agrees with LOO on well-behaved problems but

can be unstable on hierarchical models with strong shrinkage. `loo` is the recommended default; report WAIC only when it is required.

Two failure modes deserve emphasis. First, **Pareto-$\hat{k}$ flags above 0.7** indicate the importance-sampling step has broken for that observation; the elpd estimate is unreliable unless those observations are refit exactly. Second, **comparison across data subsets is invalid**. `loo` compares models on the *same* data; refitting model B with one observation removed and comparing its elpd to model A's full elpd is not a comparison.

A worked comparison:

```
fit_a <- brm(y ~ x + (1 | centre), data = sim,
             family = bernoulli(), prior = priors,
             chains = 4, cores = 4)
fit_b <- brm(y ~ x + (x | centre), data = sim,
             family = bernoulli(), prior = priors_b,
             chains = 4, cores = 4)

loo_a <- loo(fit_a)
loo_b <- loo(fit_b)
loo_compare(loo_a, loo_b)
```

The comparison reads:

```
       elpd_diff se_diff
fit_b    0.0       0.0
fit_a   -8.4       3.7
```

The random-slope model is preferred by 8.4 elpd units with SE 3.7, about 2.3 SE units, evidence in favour of the slope-varying model but not overwhelming. Re-examine Pareto-$\hat{k}$ values for both fits before concluding.

8.7. Variational inference and faster approximations

When HMC is too slow, the model has hundreds of thousands of parameters, or the user needs a fit in seconds for an interactive dashboard, three approximations are in common use.

Mean-field ADVI (Kucukelbir et al., 2017) approximates the posterior by a factorised Gaussian (after a transform to unconstrained space) and minimises the KL divergence to the true posterior using stochastic gradients. It is fast and scales to large data. It is also wrong in characteristic ways: it underestimates posterior variance (mean-field ignores correlation between parameters), can return draws from a Gaussian when the true posterior is multimodal, and provides no diagnostic for whether the approximation is adequate. Use it for exploration; verify with HMC before quoting an interval.

Full-rank ADVI retains correlation and is more accurate but quadratically more expensive. **Pathfinder** (L. Zhang et al., 2022) is a newer alternative that runs an L-BFGS optimisation, fits a Gaussian along the path, and returns importance-resampled draws. It is fast, more robust than ADVI, and serves as a high-quality initialisation for HMC.

Laplace approximation treats the posterior as Gaussian centred at the MAP with covariance equal to the inverse observed information. For posteriors that are nearly Gaussian , well-identified models with substantial data, it is accurate and trivially fast. It fails badly on skewed or multimodal posteriors and on hierarchical models with funnel geometry.

INLA (Rue et al., 2009) (Integrated Nested Laplace Approximation) is a structured approximation specialised for latent Gaussian models, GLMMs, spatial Gaussian random fields, time-series models with Gaussian latent structure. For models that fit its template, INLA is dramatically faster than MCMC and accurate enough that the spatial-statistics community treats it as the default. The `R-INLA` package is mature; `inlabru` extends it for spatial-point processes and ecology applications.

The decision rule:

Approximation	Use when	Avoid when
HMC/NUTS	Default; correctness matters	Many parameters with weak data
Pathfinder	Fast warm-start needed	Highly multimodal target
ADVI mean-field	Quick exploration only	Reporting intervals

Approximation	Use when	Avoid when
Laplace	Posterior near Gaussian	Skewed or hierarchical funnels
INLA	Latent Gaussian model	Non-conjugate non-Gaussian latent

8.8. Worked example: hospital readmission with `brms` and `loo`

We illustrate the full Bayesian workflow on a simulated hospital-readmission problem. The outcome is 30-day readmission for 12 hospitals over 4 quarters; the predictors are case-mix index, admission service line, and a quarterly indicator.

```
library(brms); library(loo); library(bayesplot)

set.seed(2026)
hosp <- tibble::tibble(
  hospital = factor(rep(1:12, each = 200)),
  quarter = factor(rep(rep(1:4, each = 50), 12)),
  service = factor(sample(
    c("med","surg","cards"), 12 * 200, replace = TRUE)),
  cmi = rnorm(12 * 200, 1.5, 0.4)
)
b_hosp <- rnorm(12, 0, 0.4)
b_q <- c(0, 0.1, -0.1, 0.05)
hosp$y <- rbinom(
  nrow(hosp), 1,
  plogis(-2 + 0.5 * (hosp$cmi - 1.5) +
         b_q[as.integer(hosp$quarter)] +
         b_hosp[as.integer(hosp$hospital)])
)

priors_m1 <- c(
  prior(normal(0, 1), class = "b"),
```

```
  prior(normal(-2, 1), class = "Intercept"),
  prior(student_t(3, 0, 0.5), class = "sd")
)

m1 <- brm(
  y ~ cmi + service + quarter + (1 | hospital),
  data = hosp, family = bernoulli(),
  prior = priors_m1, chains = 4, cores = 4,
  iter = 2000, backend = "cmdstanr"
)

m2 <- brm(
  y ~ cmi + service + quarter +
      (1 + cmi | hospital),
  data = hosp, family = bernoulli(),
  prior = priors_m1, chains = 4, cores = 4,
  iter = 2000, backend = "cmdstanr",
  control = list(adapt_delta = 0.95)
)
```

The convergence summary for `m1` reports $\hat{R} \leq 1.00$ across parameters, ESS bulk in the low thousands, no divergent transitions. The summary for `m2` initially flagged a small number of divergent transitions, addressed by raising `adapt_delta` to 0.95.

The posterior predictive check at the hospital level: `pp_check(m1, type = "stat_grouped", group = "hospital", stat = "mean")`, shows the hospital-specific readmission rates are well-recovered for `m1`. The same check on `m2` shows similar performance.

The model comparison:

```
loo_m1 <- loo(m1); loo_m2 <- loo(m2)
loo_compare(loo_m1, loo_m2)
#       elpd_diff se_diff
# m1     0.0       0.0
# m2    -2.1       1.4
```

`m1` is preferred by 2.1 elpd units with SE 1.4, a difference within sampling noise. The simpler model is adequate; the data do not support the additional flexibility. This is the kind of result that LLMs frequently mis-interpret as 'no difference, both fine.' The correct reading is 'no evidence the slope-varying model improves predictive accuracy here, prefer the simpler model on parsimony grounds.'

A Pareto-$\hat{k}$ check on `loo_m1` reports all values below 0.7. The elpd estimate is reliable. If a handful of values exceeded 0.7 we would refit those exactly with `reloo(m1, loo_m1)`.

The marginal posterior on the case-mix coefficient is 0.51 [0.41, 0.62] (90% CI), in line with the simulation truth of 0.5. The hospital-level standard deviation posterior is 0.42 [0.27, 0.61].

8.9. Collaborating with an LLM on modern Bayesian computation

Three patterns dominate. LLMs are reliable at translating a verbal model description into `brms` formula syntax and at suggesting standard prior choices. They are unreliable at diagnosing posterior failures and at interpreting `loo` comparisons in context. They cannot judge whether a posterior predictive check is passing on the right quantity.

Prompt 1: 'Translate this model to `brms` syntax.' Provide a clean specification, outcome, family, fixed effects, random-effects structure, priors, and ask for the formula and `prior()` calls.

What to watch for. The LLM will sometimes add or drop random effects, conflate `(1 | g)` with `(1 + x | g)`, or specify priors on the wrong `class`. The `class` argument in `prior()` is particularly error-prone: priors on slopes are `class = "b"`, intercept is `class = "Intercept"`, group SDs are `class = "sd"`. Run `make_stancode()` and confirm.

Verification. Print the generated Stan code with `make_stancode(fit)`. Confirm parameter names, prior distributions, and the likelihood block match your specification line by line.

Prompt 2: 'Diagnose this fit.' Paste the `summary(fit)` table and any divergent-transition counts. Ask the LLM to flag concerns and recommend next steps.

What to watch for. The LLM will typically suggest raising `adapt_delta`, increasing iterations, or reparameterising the random effects. These are correct standard moves, but the LLM cannot tell whether the *posterior itself* is the problem (e.g., a multimodal posterior arising from a non-identifiable parameterisation) versus a sampler tuning issue. Trace plots, pair plots, and prior predictive checks remain your job.

Verification. After applying suggestions, re-run diagnostics. If divergent transitions persist after `adapt_delta = 0.99` and non-centred parameterisation, the issue is the model, not the sampler. Stop tuning, look at pair plots, suspect non-identifiability.

Prompt 3: 'Interpret this `loo_compare()` output.' Paste the `loo_compare` table and the Pareto-$\hat{k}$ summaries and ask for an interpretation.

What to watch for. The LLM will quote the elpd difference and the SE and recommend the model with higher elpd. It typically does not check Pareto-$\hat{k}$ flags before doing so; it does not ask about the standard error of the difference relative to the magnitude; it does not consider whether the analysis question is even decided by predictive accuracy. A model with worse elpd may still be the right answer if its parameters are interpretable and the elpd difference is small.

Verification. Always read Pareto-$\hat{k}$ before reading elpd. If $\hat{k} > 0.7$ on more than a handful of points, elpd is unreliable until you `reloo()`. Then ask whether the elpd difference is several SE units; if not, the comparison is inconclusive.

The meta-pattern: LLMs are excellent at the *mechanics* of Bayesian workflow, formula syntax, function calls, diagnostic interpretation in textbook cases. They are poor at the *adjudication*, deciding when a passing diagnostic is misleading, when a comparison is too noisy to act on, when a model is fit-for-purpose. Treat the LLM as a fluent intern who knows the syntax but has never sat through a model-criticism review.

8.10. Principle in use

Three habits define defensible work in this area:

1. **Read the generated Stan code before running the fit.** `make_stancode()` is two seconds of effort that eliminates the most common silent failure mode in `brms` analyses. If the priors or likelihood do not match your intent, find out before the chains run for an hour.

2. **Run prior predictive checks before posterior predictive checks.** Posterior predictive checks confirm the model fits the data; prior predictive checks confirm the model is even capable of producing plausible data. A model that produces implausible draws under the prior will draw the wrong inferences in regions where data is sparse.

3. **Read `loo_compare` together with Pareto-$\hat{k}$.** Never quote an elpd difference without checking that the importance-sampling approximation has not failed. The `loo` package prints both; the temptation to skim to the elpd table is a recipe for over-confident model selection.

8.11. Exercises

1. Refit the readmission worked example with the default `brms` priors (omit the `prior` argument). Compare the posterior on the case-mix coefficient to the informative-prior fit. Under what data conditions would the difference matter?

2. Use `make_stancode()` on a `brm(y ~ x + (1 + x | g))` call with no explicit priors. Identify the correlation-matrix prior `brms` chose by default and explain what it implies about the relationship between the random intercept and slope.

3. Implement a Bayesian negative binomial regression in `brms` for an over-dispersed count outcome. Run prior predictive checks and verify that the prior on the dispersion parameter does not place excessive mass on near-Poisson dispersion.

4. Take a `brms` fit and approximate its posterior with ADVI (`vi(fit)`) and with Pathfinder (via the `cmdstanr` interface). Compare the marginal posterior for one regression coefficient across HMC, ADVI, and Pathfinder. Which approximation underestimates variance? On which parameters?

5. Construct a scenario where two `brms` models have nearly identical elpd (within 1 SE) but make meaningfully different predictions on an out-of-distribution test point. What does this illustrate about treating elpd as the sole comparison metric?

8.12. Further reading

- Gelman et al. (2020), *Bayesian Workflow*. The canonical argument for iterative model-criticism over one-shot fitting.
- McElreath (2020), *Statistical Rethinking* (2nd edition). Applied Bayesian regression with `brms` and `rethinking`, accessible to a first-time audience.
- Vehtari et al. (2017), *Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC*. The reference for `loo` and Pareto-$\hat{k}$ diagnostics.
- The `brms` documentation (https://paul-buerkner.github.io/brms/) and `loo` vignettes (https://mc-stan.org/loo/) are exemplary.

Part IV.

Scaling and Modelling

9. High-Performance and Distributed Computing

9.1. Learning objectives

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

- Choose between `parallel`, `future`, `furrr`, and `mirai`/`crew` for a given parallelisation problem and explain the cost model that governs the choice.
- Distinguish embarrassingly parallel work from work with inter-process communication, and recognise when the overhead of parallelism exceeds its benefit.
- Use Apache Arrow and `arrow` for out-of-memory tabular data, and DuckDB for SQL-style analytics on data larger than RAM.
- Decide when to move work to a GPU via `torch` or `gpuR`, and recognise the small set of statistical workloads where GPU acceleration actually pays off.
- Reason about cloud computation, instance selection, spot/preemptible pricing, storage tiers, in terms of total cost of ownership rather than just hourly rate.
- Profile and benchmark before parallelising; document the intended speedup and confirm it on representative data.

9.2. Orientation

Statistical computation often runs on a single laptop until suddenly it cannot. The bootstrap with 1000 replicates finishes in ten minutes; the same bootstrap on the production cohort with hundreds of strata stalls overnight. The posterior predictive check that ran fine on a 5% sample

crashes the R session at full sample. The Monte Carlo study that completed in an afternoon now needs a week of runs.

This chapter covers the next layer above the laptop. Most statistical work is **embarrassingly parallel**, independent replicates of the same computation, and the right tool for that case is rarely the most sophisticated one. We start with the parallelism stack in R, work through out-of-memory tabular data with Arrow and DuckDB, touch GPU computation where it matters, and end with cloud-cost reasoning.

The framing throughout: **profile before parallelising, parallelise the right loop, and verify the speedup is real.** Parallelisation is one of the few engineering moves that can both speed work up and silently corrupt results; treat it accordingly.

9.3. The statistician's contribution

GPUs and clusters do not solve statistical problems; they solve compute problems. The statistician's role is to ensure the compute problem being solved is the right one, and that the speedup is bought at a cost the project can afford.

(Judgement 1.) Parallelise the right level. A naive intuition is to parallelise the innermost loop. This is almost always wrong. The right level is the *outermost* embarrassingly parallel loop: the bootstrap replicate, the simulation iteration, the cross-validation fold. Parallel overhead, process startup, data serialisation, result collection, is paid once per dispatched task. Splitting a fast inner loop across workers can leave you slower than the serial version. The statistician identifies the right unit of parallelism by understanding where the natural independence lies; an LLM that says 'use `mclapply`' has elided the question.

(Judgement 2.) Random-number reproducibility is a correctness issue. Two parallel workers using the same seed will produce identical 'random' replicates, which is catastrophic for a Monte Carlo study and silent, the run completes, the output looks reasonable, the variance estimate is wrong. The defence is L'Ecuyer's parallel RNG (`set.seed(., kind =`

`"L'Ecuyer-CMRG"`) plus per-task seed advancement) or the `future` framework's `furrr_options(seed = TRUE)`. The statistician must specify reproducibility requirements before parallelising; LLMs frequently produce parallel code that is fast and wrong on RNG.

(Judgement 3.) Cloud cost is your problem. Spot instances are 60–90% cheaper than on-demand, but they can be reclaimed mid-run. A 12-hour Stan fit on a spot instance that gets reclaimed at hour 11 has cost the spot rate for 11 hours and produced no output. The statistician decides whether to use spot, on-demand, or reserved instances based on the value of a partial completion, the cost of a restart, and the deadline. These are not technical choices; they are project-management choices. Cloud cost surprises end careers and budgets; cloud cost discipline is part of the work.

These judgements are what distinguish defensible high-performance work from results that are merely fast.

9.4. The R parallelism stack

Four frameworks cover most R parallel workloads.

`parallel` (base R). `mclapply()` (forking, Unix only) and `parLapply()` (PSOCK clusters, cross-platform) are the oldest tools. They work; they are well-understood; they are also the most error-prone interface, particularly around RNG seeds, error propagation, and progress reporting. Use them when no dependency on `future` is acceptable.

`future` and `furrr`. `future` (Bengtsson, 2021) provides a uniform API for asynchronous computation across backends: sequential, multicore, multisession, cluster, and cloud. `furrr::future_map()` is the parallel counterpart of `purrr::map()` and handles RNG correctly with `furrr_options(seed = TRUE)`. This is the recommended default for applied parallelism: code written with `future_map()` runs on a laptop, a workstation, or a cluster with one `plan()` call.

`mirai` and `crew`. `mirai` is a lighter, lower-overhead alternative to `future` for high-throughput dispatch. `crew` builds on `mirai` to support persistent workers and auto-scaling, and is the engine behind `targets`'s parallelism.

For pipelines that dispatch many short tasks, `crew` is dramatically more efficient than `future`: seconds versus minutes of overhead at scale.

`batchtools`. For HPC clusters with schedulers (SLURM, PBS, LSF), `batchtools` is the established interface. It has a steeper learning curve and is overkill for laptop or workstation work, but it is the right tool when you need to submit thousands of jobs to a scheduler and harvest the results.

A worked example contrasts the styles. Suppose we want to fit 500 bootstrap replicates of a regression model.

```
library(furrr)
plan(multisession, workers = 8)

boot_fit <- function(i, data) {
  resample <- data[sample(nrow(data), replace = TRUE), ]
  coef(lm(y ~ x1 + x2, data = resample))
}

results <- future_map(
  1:500,
  ~boot_fit(.x, my_data),
  .options = furrr_options(seed = TRUE)
)
```

The `.options = furrr_options(seed = TRUE)` is non-negotiable. Without it, `future_map` will warn but proceed; the resulting 'replicates' may not be statistically independent.

The cost model, simplified:

- Forking (`mclapply` on Unix) shares memory copy-on-write; startup cost is microseconds.
- PSOCK clusters and `multisession` workers spawn fresh R sessions; startup cost is seconds and memory must be serialised across the boundary.
- Cluster backends pay scheduler queueing time on top.

The implication: if your task is sub-second and you have copy-on-write available, fork; if your task is seconds to minutes and you need cross-platform, use `multisession`; if your task is minutes to hours and you have scheduler access, batch.

9.5. Out-of-memory tabular data: Arrow and DuckDB

When a dataset will not fit in RAM, the right move is rarely 'rent a bigger machine.' It is to switch to columnar out-of-memory tools that read only the columns and rows required.

Apache Arrow is a columnar in-memory format and a set of tools for working with it. The R `arrow` package binds to the C++ implementation. Key idioms:

```
library(arrow)
library(dplyr)

ds <- open_dataset("data/large_cohort/", format = "parquet")
ds |>
  filter(year == 2024, site %in% c("A","B")) |>
  group_by(site) |>
  summarise(n = n(), mean_age = mean(age, na.rm = TRUE)) |>
  collect()
```

The query is built lazily, no data is read, until `collect()` materialises the result into an in-memory tibble. Arrow pushes the filter down to the file scan, so only the rows matching the predicate are decoded. On a 50 GB Parquet dataset this can be the difference between two seconds and two hours.

DuckDB (Raasveldt & Mühleisen, 2019) is an embedded analytical database, like SQLite, but column-oriented and tuned for OLAP queries. The `duckdb` and `duckplyr` R packages expose DuckDB through a `dplyr`-compatible interface. DuckDB excels at SQL-shaped analytics on data larger than RAM, joins, and window functions. For a project that already speaks SQL, DuckDB is often the cleanest path.

```
library(duckdb); library(DBI); library(dplyr)
con <- dbConnect(duckdb())
dbExecute(con, "CREATE VIEW cohort AS
  SELECT * FROM read_parquet('data/large_cohort/*.parquet')")
tbl(con, "cohort") |>
  filter(year == 2024) |>
  count(site) |>
  collect()
```

The decision between Arrow and DuckDB is largely about ergonomics. Arrow integrates more naturally with `dplyr` pipelines and the broader R ecosystem; DuckDB is faster on complex joins and excels at SQL-shaped work. Many projects use both, Arrow to read and filter, DuckDB to join.

Check your understanding: when not to parallelise

Question. A colleague has a `purrr::map()` call over 10 elements, each taking about 50 ms. They propose switching to `furrr::future_map()` to speed it up. Should they?
Answer.
No. The total serial time is about 500 ms. Spawning a multisession `future` plan typically costs more than a second per worker for R session startup, plus serialisation overhead per task. The parallel version will be slower than the serial version. Parallelisation is profitable when total serial time substantially exceeds parallel overhead; sub-second loops over a handful of items are below that threshold. Run the serial version, profile, and parallelise only the loops that dominate.

9.6. GPU computation: where it matters and where it does not

GPUs accelerate dense linear algebra and matrix-shaped computation. Most statistical workloads are not matrix-shaped in the way GPUs reward, and most attempts to move statistical code to GPU end up slower or no faster. The narrow class of problems where GPU pays off:

- **Deep learning and neural network training.** `torch` for R and `tensorflow/keras` are the established interfaces. These are GPU-native and benefit from order-of-magnitude speedups on appropriately sized models.
- **Large dense matrix algebra.** Models that hinge on Cholesky or eigendecomposition of dense matrices in the thousands-of-rows range can benefit from GPU BLAS. `gpuR` exposes this via OpenCL.
- **Stan and PyMC under specific configurations.** Stan can run gradient evaluations on GPU for some model classes; PyMC via JAX/NumPyro can target GPU and TPU. Speedups are model-dependent; benchmark before assuming benefit.

What does *not* benefit from GPU: looping over heterogeneous small models, data manipulation, most regression fitting on small-to-medium datasets, and any workload bottlenecked by data movement to and from GPU memory. The CPU GPU memory transfer is often the dominant cost; if your inner loop touches GPU memory many times, you have lost the gain.

The decision rule is empirical: benchmark a representative sub-problem on CPU and GPU before committing infrastructure to a GPU-based pipeline.

9.7. Cloud computation and cost discipline

Three observations dominate cloud-cost reasoning for statistics.

Spot instances are 60–90% cheaper but reclaimable. AWS spot, Azure low-priority, and GCP preemptible instances offer substantial discounts in exchange for the right of the provider to reclaim them on short notice. For fault-tolerant workloads, a Monte Carlo study where individual replicates can be retried, they are excellent. For long-running serial fits with no checkpointing, they are dangerous. Either save state at every iteration (so a reclaim is at most a small loss) or pay for on-demand.

Storage tiers compound. S3 standard storage is 10× cheaper than EBS attached to a running instance. Holding 50 TB of intermediate data on EBS for a six-month project is materially more expensive than holding it on S3 with on-demand access. The pattern that scales: keep cold data on S3, mount only the working subset to compute, write results back to S3.

`arrow::s3_bucket()` and `arrow::open_dataset()` make this idiomatic from R.

Right-size, do not over-size. The instance type with the most cores is rarely the right answer for a statistical workload. Memory-bandwidth-bound work (most R) benefits more from fewer fast cores with high memory bandwidth than from many slow cores; compute-bound matrix work may benefit from the opposite. Run a benchmark on a small instance, project the runtime to a large one, and pick the smallest instance that meets the deadline.

The framework that has emerged for managing this complexity is **infrastructure as code**, tools like `paws` and `googleCloudStorageR` to script provisioning, plus `targets` with cloud-aware backends to dispatch and harvest work. The pattern: a `_targets.R` pipeline runs locally on a laptop and on a cluster with the same code, parametrised only by the `crew` controller it uses.

9.8. Worked example: a parallel bootstrap with reproducible RNG

A 5000-replicate bootstrap of a Cox proportional hazards model fit, with explicit RNG handling and a `targets` pipeline that scales from laptop to cluster.

```
# _targets.R
library(targets)
library(crew)

tar_option_set(
  packages = c("survival", "dplyr"),
  controller = crew_controller_local(workers = 8)
)

boot_one <- function(i, data) {
  ix <- sample(nrow(data), replace = TRUE)
  fit <- coxph(Surv(time, event) ~ trt + age + stage,
```

```
                data = data[ix, ])
  coef(fit)
}

list(
  tar_target(cohort, readRDS("data/cohort.rds")),
  tar_target(
    boot_replicates,
    boot_one(seq_len(5000), cohort),
    pattern = map(seq_len(5000)),
    iteration = "list",
    cue = tar_cue(seed = TRUE)
  ),
  tar_target(
    boot_summary,
    do.call(rbind, boot_replicates) |>
      apply(2, function(x) c(
        est = mean(x), lo = quantile(x, 0.025),
        hi = quantile(x, 0.975)
      ))
  )
)
```

`crew_controller_local(workers = 8)` runs locally; swapping to `crew.cluster::crew_controller_slurm()` reuses the same pipeline on a SLURM cluster with one line changed. `cue = tar_cue(seed = TRUE)` ensures the per-replicate seed is deterministic, the same pipeline rerun produces the same bootstrap distribution.

A serial benchmark for comparison:

```
serial <- system.time({
  out <- lapply(1:500, boot_one, data = cohort)
})
# user.self  sys.self   elapsed
#   42.13     0.51       42.78
```

With 8 workers under `crew`, the same 500 replicates take about 7.4 seconds,

speedup of 5.8×, accounting for serialisation and dispatch overhead. The full 5000-replicate run completes in just over a minute, and the same pipeline deployed to a 64-worker cluster runs in under 10 seconds.

9.9. Collaborating with an LLM on HPC

Three patterns dominate. LLMs are useful for translating serial R code to a parallel form, for suggesting Arrow and DuckDB query rewrites, and for reasoning about cloud cost trade-offs. They are unreliable on parallel RNG, on diagnosing why a parallel run is slower than serial, and on GPU benchmarking advice that is not blanket optimism.

Prompt 1: 'Parallelise this bootstrap loop with reproducible RNG.' Provide the serial loop. Ask for a `furrr_map` version with proper seed handling.

What to watch for. The LLM may produce parallel code without `furrr_options(seed = TRUE)`, or with `set.seed()` inside the worker (which produces identical replicates across workers, a correctness bug). It may suggest `mclapply()` without the `mc.set.seed = TRUE` argument and the `RNGkind("L'Ecuyer-CMRG")` setup that makes it work.

Verification. Run the parallel version twice with the same seed; the bootstrap distribution should be identical. Run it once and inspect a small number of replicates for duplicate values across workers. Identical replicates from different worker indices are a definitive RNG-bug signature.

Prompt 2: 'Why is my parallel run slower than serial?' Paste the serial timing, the parallel timing, the worker count, the per-iteration work, and the data size.

What to watch for. The LLM will often suggest 'increase workers' or 'switch backends' without diagnosing the underlying issue. The common causes, too-fine-grained parallelism, large per-task data serialisation, GIL-style bottlenecks in BLAS-using code, require the LLM to ask clarifying questions about per-task work size and data movement. If the LLM does not ask, push back.

Verification. Profile a single worker iteration with `profvis`. Compute total work time, parallel overhead per task, and projected speedup with

W workers. If projected speedup is less than 2×, parallelisation is not the right intervention; restructure the loop instead.

Prompt 3: 'Should I use spot or on-demand instances for this 8-hour Stan fit?' Provide the workload, deadline, and checkpointing capability.

What to watch for. The LLM defaults to recommending spot for cost reasons. It typically does not weigh the cost of a mid-run reclaim, for an 8-hour fit with no checkpointing, a reclaim at hour 7 wastes the full instance cost and the elapsed time before a re-run. The right answer depends on your tolerance for re-runs and your deadline.

Verification. If the model can checkpoint (save Stan's warm-up state, resume), spot is fine. If it cannot, compute the expected cost including the probability of reclaim times the redo cost. A 5% reclaim probability on an 8-hour fit means an expected 24 minutes of waste per run; on a $10/hour instance that is $4 expected waste against roughly $48 in savings, spot still wins, but the calculation should be made.

The meta-pattern: LLMs are competent on the syntactic mechanics of parallelism (which package, which function) and weak on the operational reasoning (will it actually be faster, will it be correct, will it cost less). Use them for code translation; do the cost and correctness reasoning yourself.

9.10. Principle in use

Three habits define defensible HPC work:

1. **Profile first, parallelise second.** A serial run with `profvis` identifies the bottleneck. Parallelising a non-bottleneck wastes engineering time and adds bugs. The 80/20 rule: most of the runtime is in 20% of the code; parallelise that 20%.

2. **Make RNG explicit and tested.** Every parallel computation that uses random numbers must specify `furrr_options(seed = TRUE)` or its equivalent. Test by running the pipeline twice with the same seed; the results must match exactly. Anything else is a correctness bug masquerading as a feature.

3. **Cost-budget the run.** Before submitting a long cloud job, write down the expected runtime, the per-hour cost, the total budget, and the kill criterion. A $400/day instance left running over a forgotten weekend has cost real projects real money. Set up billing alerts; check them.

9.11. Exercises

1. Take a serial Monte Carlo study of your choice (1000 replicates, each ~1 second). Implement it three ways: `mclapply`, `furrr::future_map` with multisession, and `crew` via `targets`. Benchmark each. Account for the overhead difference; does the cheapest framework give the lowest total time on this workload?

2. Convert a `dplyr` pipeline that operates on a 5 GB CSV to operate on a Parquet dataset via Arrow. Measure the speedup on (a) a column-subsetting query, (b) a `filter`

   - `summarise` query, and (c) a self-join. Identify which query benefits most from columnar layout and explain why.

3. Set up a SLURM (or PBS) job submission for a 100-replicate simulation using `targets` with `crew.cluster`. Configure it to run on the cluster but also to run locally with no code change. Document the configuration switch.

4. Profile a `brms` fit with `profvis`. Identify the three most expensive steps. Are any of them parallelisable without modifying Stan? (Hint: most are not.)

5. Cost-model a 200-iteration MCMC study where each chain takes 4 hours. Assume an on-demand `c5.4xlarge` is $0.68/hour, spot is $0.20/hour, and spot reclaim probability is 8%. With 200 chains needed and four chains per instance, compute the total expected cost under each strategy. Document your assumptions.

9.12. Further reading

- Bengtsson (2021), *A Unifying Framework for Parallel and Distributed Processing in R using Futures*. The reference for the `future` ecosystem.
- Raasveldt & Mühleisen (2019), *DuckDB: an Embeddable Analytical Database*. The original DuckDB paper.
- The Apache Arrow project documentation for R (https://arrow.apache.org/docs/r/) is current and comprehensive.
- The `targets` user manual (https://books.ropensci.org/targets/) is the canonical reference for reproducible pipelines, with extensive HPC backend coverage.

10. High-Dimensional and Sparse Methods

10.1. Learning objectives

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

- Distinguish the lasso, ridge, elastic net, group lasso, and adaptive lasso, and choose between them based on the scientific question and the structure of the predictors.
- Use cross-validation in `glmnet` for tuning λ, and recognise its weaknesses for selection-stable inference.
- Apply screening rules (SAFE, strong, EDPP) to scale lasso fits to p in the millions, and use `biglasso` for out-of-memory designs.
- Use the knockoff filter (Candès et al., 2018) to control the false-discovery rate among selected variables, and explain what guarantees it provides and when those guarantees fail.
- Reason about post-selection inference: why naive confidence intervals after lasso are wrong, and what `selectiveInference`, `hdi`, and debiased lasso offer instead.
- Recognise when sparse methods are the right tool, and when they are an overfit solution to a problem that calls for hierarchical Bayesian shrinkage or principal components.

10.2. Orientation

When the number of candidate predictors p approaches or exceeds the sample size n, ordinary least squares no longer identifies a unique solution and overfitting becomes the dominant statistical risk. Modern high-dimensional methods solve this by **structural assumption**, the truth is sparse (few

non-zero coefficients), or grouped, or low-rank, and **regularisation**, a penalty that encodes that assumption into the optimisation.

The lasso (Tibshirani, 1996) launched this field by combining sparsity with computational tractability. The twenty years since have produced refinements (elastic net, group lasso, adaptive lasso), scalability advances (coordinate descent, screening rules, biglasso), and a post-selection inference apparatus (knockoffs, debiased lasso, selective inference) that gives back something resembling confidence intervals.

This chapter covers all three. We start with the penalty zoo and tuning, work through scaling, and finish with selection inference and the knockoff filter. The framing throughout: **sparsity is an assumption, not a fact**. When the assumption holds, sparse methods are dramatically effective. When it does not, they produce confidently wrong answers.

10.3. The statistician's contribution

LLMs can fit a `glmnet` model and call `cv.glmnet`. They cannot decide whether the lasso assumption, that few predictors are truly associated with the outcome, is appropriate for the problem at hand, nor whether the selected variables warrant the causal language a collaborator wants to attach to them.

(Judgement 1.) Sparsity is a modelling commitment. The lasso assumes a sparse truth, most coefficients are exactly zero. In some applications this is a reasonable prior: a genome-wide association study where most SNPs are unrelated to the phenotype. In others it is implausible: a clinical risk model where many demographic and clinical factors plausibly contribute small, partially correlated effects. Forcing sparsity on a non-sparse truth produces a biased model that is over-confident about which variables matter. Ridge regression or hierarchical Bayesian shrinkage would be the better tool. The statistician decides whether sparsity matches the science.

(Judgement 2.) The selected set is not the active set. The lasso selects predictors based on their conditional contribution given the others. Highly correlated predictors get arbitrarily assigned: lasso picks one and zeros the others, even when biologically the others are equally involved.

The selected set is *one valid sparse representation* of the data, not the set of true causal contributors. Reporting the selected variables as 'the variables associated with the outcome' overstates the guarantee. The statistician explains what the selection does and does not show.

(Judgement 3.) Post-selection inference is not optional window-dressing. Naive standard errors and p-values computed on the selected variables ignore the selection event and are anti-conservative. The fix, knockoffs, debiased lasso, sample splitting, or stability selection: is part of the analysis, not a refinement. An LLM that returns a `glmnet` fit and the selected variables without addressing inference has produced an exploration, not a result. The statistician owns this distinction.

These judgements decide whether a high-dimensional analysis informs a scientific claim or merely produces a list of covariates.

10.4. The penalty zoo

The lasso minimises

$$\frac{1}{2n}\|y - X\beta\|_2^2 + \lambda\|\beta\|_1$$

for tuning parameter $\lambda \geq 0$. The ℓ_1 penalty shrinks coefficients toward zero and sets some exactly to zero, performing simultaneous shrinkage and selection.

Ridge uses $\|\beta\|_2^2$ instead. It shrinks but does not select, and it is the right choice when many small contributions are expected.

Elastic net (Zou & Hastie, 2005) mixes the two: $\alpha\|\beta\|_1 + (1-\alpha)\|\beta\|_2^2$. The combination retains the lasso's selection while sharing information across correlated predictors, neither lasso's arbitrary single-pick nor ridge's no-selection.

Group lasso (Yuan & Lin, 2006) applies the penalty to groups of coefficients (e.g., all dummy levels of a categorical predictor), so a group is selected or zeroed as a unit. The right tool whenever predictors come in natural groups: dummy-coded factors, splines, gene pathways.

Adaptive lasso (Zou, 2006) applies different weights to different coefficients, with the weights derived from a preliminary estimator. Under regularity

conditions it has the **oracle property**, it consistently selects the true active set and gives asymptotically normal estimates on that set, properties the plain lasso lacks.

SCAD (Fan & Li, 2001) and **MCP** (C.-H. Zhang, 2010) are non-convex penalties that further reduce the bias toward zero on large coefficients. They are implemented in `ncvreg` and are standard in genetics applications. They lose the convex-optimisation guarantees of the lasso but often perform better in practice on truly sparse problems.

A practical default: **start with elastic net**. The lasso is a corner case ($\alpha = 1$); ridge is the other ($\alpha = 0$); the elastic net interpolates and is robust to the mis-specification of either extreme. Use the group lasso when groups are natural; use SCAD or MCP when the bias of the lasso is the binding constraint.

10.5. Tuning λ via cross-validation

`cv.glmnet` is the standard tool. Two choices of λ are returned: `lambda.min` (the value minimising mean CV error) and `lambda.1se` (the largest λ within one standard error of the minimum). The 1-SE rule produces sparser, more interpretable solutions at modest predictive cost; it is the conventional default for *selection*. For prediction-oriented work, `lambda.min` typically wins on out-of-sample loss.

```
library(glmnet)
fit_cv <- cv.glmnet(
  x = X, y = y,
  family = "binomial", alpha = 0.5,
  nfolds = 10
)
plot(fit_cv)
coef(fit_cv, s = "lambda.1se")
```

Three points deserve emphasis.

Standardise predictors. `glmnet` standardises by default; this is correct and should not be turned off. The lasso penalty is not scale-invariant,

so unstandardised predictors on different scales receive different effective penalties.

The CV loss is not the inferential target. CV measures predictive accuracy. If your question is which variables matter, CV optimisation is a means to an end. If your question is which variables are stable across resamples, **stability selection** (Meinshausen & Bühlmann, 2010) is a better tool: refit on many subsamples, retain variables that are selected with high probability.

Repeated CV reduces variance. Single 10-fold CV gives noisy λ estimates. Repeating CV with different fold assignments and averaging is standard. `glmnet`'s `cv.glmnet` does not do this directly; wrap it in a loop or use `caret`/`tidymodels` for repeated CV.

Check your understanding: when ridge beats lasso

Question. A clinical risk model has 50 candidate predictors, all expected to contribute small but non-zero effects (demographics, comorbidities, lab values). The investigator asks whether to use lasso or ridge. The dataset has $n = 5000$. Which is appropriate?
Answer.
Ridge (or elastic net with low α). The scientific prior is that most predictors contribute small effects, which is the ridge regime. Lasso would zero many true (small) effects, biasing the resulting model. With $n = 5000$ and $p = 50$, OLS would also work but variance can still be reduced by mild shrinkage. Elastic net with $\alpha \leq 0.2$ is a defensible default; pure ridge ($\alpha = 0$) is the appropriate choice if no selection is desired.

10.6. Scaling: screening rules and `biglasso`

When p is in the millions, typical for genome-wide analyses, even coordinate descent on the full design is too slow. Two ideas extend the lasso's reach.

Screening rules identify predictors that are guaranteed or almost-guaranteed to have zero coefficient at the target λ, and exclude them from the active-set computation.

- **SAFE** (El Ghaoui et al., 2012) gives a provably safe exclusion: predictors that fail the SAFE bound have zero coefficient at λ, exactly.
- **Strong rules** (Tibshirani et al., 2012) are a near-safe heuristic: provably correct at the first λ on the path, then progressively heuristic. In practice they exclude almost all truly inactive predictors and require only an active-set check after fitting to confirm correctness.
- **EDPP** (Wang et al., 2015) (Enhanced Dual Polytope Projection) is a more aggressive safe rule that often excludes 99% or more of predictors at the first λ.

`glmnet` uses strong rules internally. For very large problems where even strong rules are insufficient, `biglasso` (Zeng & Breheny, 2017) adds out-of-memory support via memory-mapped files, parallelism, and a more aggressive EDPP-derived screening, achieving order-of-magnitude speedups over `glmnet` on p in the millions.

```
library(biglasso)
X.bm <- as.big.matrix(X)
fit <- biglasso(X.bm, y, family = "binomial",
                ncores = 4)
plot(fit)
```

The `bigmemory` design avoids loading the full matrix into R's heap, allowing fits on designs that exceed RAM.

10.7. Selection inference: knockoffs and beyond

Once you have a selected variable set, the natural next question, 'are these variables really associated, or is some of the selection a chance artefact?', is the question naive lasso output cannot answer.

The knockoff filter (Candès et al., 2018) constructs artificial 'knockoff' variables that mimic the joint distribution of the original predictors but are conditionally independent of the outcome given the originals. Running the lasso on the augmented design (originals plus knockoffs) and comparing the order of entry of originals versus knockoffs gives a finite-sample false discovery rate (FDR) guarantee.

The pseudocode is straightforward:

```
library(knockoff)
result <- knockoff.filter(
  X, y, fdr = 0.10,
  knockoffs = create.gaussian,
  statistic = stat.glmnet_coefdiff
)
result$selected
```

The result is a set of variables for which the FDR, the expected proportion of false discoveries among the selected: is at most 0.10. This is a rigorous guarantee, finite-sample, under one assumption: the knockoff construction satisfies the exchangeability condition. For Gaussian designs with known covariance the construction is exact; for unknown covariance it is approximate (`create.gaussian` estimates). For non-Gaussian designs, the **model-X knockoff** version (Candès et al., 2018) requires only that the joint distribution of X is known or estimable.

The framework's strengths and limits:

- **Strength.** A finite-sample FDR guarantee is rare in high-dimensional inference. The knockoff filter is the cleanest available answer for genome-wide selection.
- **Limit.** Power can be modest, especially when predictors are highly correlated. The knockoffs and the originals are by construction similar, so the lasso may enter them in similar order.
- **Limit.** The construction depends on knowing the joint distribution of X. For a designed experiment or GWAS where X has known structure, this is fine. For observational data with arbitrary correlations and missing data, the assumptions become heavier.

Other approaches. Debiased lasso (Geer et al., 2014; C.-H. Zhang & Zhang, 2014) removes the bias from the lasso estimate by an explicit correction, producing asymptotically normal estimators amenable to confidence intervals. **Selective inference** (Lee et al., 2016; Taylor & Tibshirani, 2018) conditions on the selection event and produces p-values valid given that selection. **Sample splitting** (Meinshausen et al., 2009) uses one half of the data to select and the other to compute classical inference on

the selected set. **Stability selection** (Meinshausen & Bühlmann, 2010) runs many lasso fits on subsamples and retains variables selected with high probability.

For a typical analysis the recommendation is: **use knockoffs for FDR-controlled selection in high-dimensional discovery; use sample splitting or debiased lasso when classical confidence intervals on the selected set are required; use stability selection as a sanity check on any selected set.**

10.8. Worked example: a genome-wide-style analysis

We illustrate on a simulated design with $n = 1000$, $p = 20{,}000$, and 30 truly active predictors.

```
library(glmnet)
library(knockoff)
set.seed(228)

n <- 1000; p <- 20000
X <- matrix(rnorm(n * p), n, p)
beta <- numeric(p)
active <- sample(p, 30)
beta[active] <- rnorm(30, 0, 1.5)
y <- X %*% beta + rnorm(n)

fit_cv <- cv.glmnet(X, y, alpha = 0.5)
selected_glmnet <- which(coef(fit_cv,
                              s = "lambda.1se")[-1] != 0)
length(selected_glmnet)
length(intersect(selected_glmnet, active))
```

The elastic net at `lambda.1se` selects roughly 60–80 variables; about 22 of the 30 truly active are recovered; the rest are false discoveries, a false-discovery rate near 60%. This is what naive selection looks like in this regime.

```
result <- knockoff.filter(
  X, y, fdr = 0.10,
  knockoffs = create.second_order,
  statistic = stat.glmnet_coefdiff
)
length(result$selected)
length(intersect(result$selected, active))
```

The knockoff filter at FDR 0.10 selects about 18 variables; about 17 are truly active. The realised FDR is roughly 0.06, well within the 0.10 target. The price is reduced power: the knockoff selection misses 13 of the 30 active predictors while the naive lasso would have caught more, but the false-discovery rate is now controlled.

For a genome-wide study where the cost of follow-up experiments on false discoveries is high, the knockoff filter's controlled FDR is usually the better trade.

10.9. Collaborating with an LLM on high-dimensional methods

Three patterns dominate. LLMs are reliable on the syntax of `glmnet`, `biglasso`, `knockoff`, and `ncvreg`, and on explaining the lasso–ridge–elastic net distinction. They are unreliable on the choice of method for a given scientific question and on selection inference.

Prompt 1: 'Fit a lasso to this binary outcome and report the selected variables.' Provide the data and the formula.

What to watch for. The LLM will return a `cv.glmnet` fit and a list of selected variables, often without warning about the absence of selection inference. It will not flag that the selected variables are not the same as 'the variables associated with the outcome' and that classical intervals on those variables are invalid. It will not mention stability selection or knockoffs.

Verification. Always supplement a lasso selection with at least one of (a) stability selection, (b) knockoffs for FDR control, or (c) sample splitting for

classical inference. The LLM-default 'fit and report selected' is the start of an analysis, not its end.

Prompt 2: 'Should I use lasso, ridge, or elastic net for this problem?' Provide the scientific question, the expected sparsity, and the predictor structure.

What to watch for. The LLM will often default to the lasso on the basis of interpretability and sparsity arguments without checking whether sparsity is a defensible assumption for the problem. It may not raise the correlated-predictor issue (lasso picks arbitrarily among correlated predictors).

Verification. Ask explicitly: is a sparse truth plausible? Are predictors correlated? If the answer to sparsity is 'no' or unclear, ridge or elastic net with small α is the safer default. If the answer to correlation is 'yes,' the elastic net's sharing of correlated predictors is a feature.

Prompt 3: 'Implement a knockoff filter for this design and report the selected variables at FDR 0.05.' Provide the design and outcome.

What to watch for. The LLM will produce a working `knockoff::knockoff.filter` call. It will not always discuss the assumption that the joint distribution of X is appropriately captured, and it may default to `create.gaussian` even when the design is non-Gaussian. It will rarely run a power-loss diagnostic comparing knockoff selection to plain lasso selection.

Verification. Check that the knockoff construction is appropriate for the design (Gaussian, second-order, model-X with conditional density estimation). Compare the knockoff selection size and overlap with the plain lasso selection; a knockoff filter that selects far fewer variables than the lasso may be reasonable (FDR control trades power) or may signal a knockoff construction problem.

The meta-pattern: LLMs handle the *fitting* of high-dimensional methods well and the *adjudication*: which method matches the science, what selection inference is required, poorly. Frame the scientific question before asking for a fit, and never accept a selected-variable list without an inference plan attached.

10.10. Principle in use

Three habits define defensible high-dimensional work:

1. **State the sparsity assumption.** Before fitting a lasso, write down why you expect the truth to be sparse and what proportion of predictors you expect to be active. If the answer is 'I have no reason to expect sparsity,' use ridge or elastic net with small α instead.
2. **Pair selection with inference.** Every selected-variable list should be reported with one of: stability selection probabilities, knockoff FDR-controlled selection, debiased lasso intervals, or sample-split classical intervals. Naked selection lists invite over-interpretation.
3. **Stress-test correlated predictors.** When variables are highly correlated, lasso selection is unstable; small changes in the data can swap which correlated variable is retained. Run the analysis on bootstrap resamples and report the proportion of resamples on which each variable is selected. Variables selected on $< 60\%$ of resamples should be reported as candidate rather than confirmed.

10.11. Exercises

1. Simulate a regression with $n = 500$, $p = 5000$, 50 truly active predictors with effects $N(0, 1)$, and correlation $\rho = 0.7$ within blocks of 5 predictors. Fit lasso, ridge, and elastic net at $\alpha = 0.5$. Compare predictive MSE and the count of true positives among selected. Which method dominates?
2. Repeat exercise 1 but with effects $N(0, 0.2)$ on *all* predictors (no truly zero coefficients). Which method dominates now? Explain why.
3. Apply the knockoff filter to the design from exercise 1 at FDR 0.10. Report the realised FDR and the power. Is the realised FDR within the target?
4. Use stability selection on the design from exercise 1. Plot the selection probability as a function of λ for the truly active variables and for a random sample of inactive ones.

5. Compare the running time of `glmnet` and `biglasso` on a design with $n = 5000$, $p = 100{,}000$. Document the speedup and identify under what conditions `biglasso` becomes worth the additional setup cost.

10.12. Further reading

- Hastie et al. (2015), *Statistical Learning with Sparsity*. The textbook treatment of the lasso and its extensions, by the authors of `glmnet`.
- Tibshirani (1996), *Regression Shrinkage and Selection via the Lasso*. The original paper.
- Candès et al. (2018), *Panning for Gold: Model-X Knockoffs for High-Dimensional Controlled Variable Selection*. The knockoff filter reference.
- Meinshausen & Bühlmann (2010), *Stability Selection*. The reference for resampling-based selection.
- The `glmnet` vignette (https://glmnet.stanford.edu/) is exemplary; the `knockoff` package documentation (https://web.stanford.edu/group/candes/knockoffs/) is the practical entry point.

11. Machine Learning for Biostatistics

11.1. Learning objectives

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

- Use `tidymodels` to specify, fit, tune, and evaluate predictive models with a consistent workflow.
- Choose between gradient-boosted trees (`xgboost`, `lightgbm`), random forests (`ranger`), and regularised GLMs based on dataset characteristics, interpretability needs, and the role of the model in the analysis.
- Implement nested cross-validation for hyperparameter tuning and unbiased performance estimation, and recognise when single-level CV is misleadingly optimistic.
- Apply calibration assessment and recalibration to probabilistic classifiers, and explain why area under the ROC curve does not measure calibration.
- Use SHAP values (Lundberg & Lee, 2017) for model interpretation, and articulate what they show and what they do not (association, not causation).
- Recognise when a 'machine learning' framing is hiding a classical statistical question, and when the reverse is true.

11.2. Orientation

The boundary between machine learning and biostatistics is porous and confused. Both fit predictive models from data; both quantify uncertainty; both worry about overfitting and generalisation. The cultural distinctions,

ML emphasises black-box prediction, biostatistics emphasises interpretable inference, were never as clean as the caricatures and have eroded further as both fields adopt each other's tools.

This chapter is not an introduction to machine learning. The goal is narrower: how to *use* the modern ML toolkit in biostatistical work where the question is genuinely predictive, and how to recognise when the toolkit is being mis-applied to a question that calls for classical inference instead.

We anchor on `tidymodels` (Kuhn & Silge, 2022) as the contemporary R interface, a coherent, principled stack that handles preprocessing, resampling, tuning, and evaluation with consistent verbs. We then walk through the algorithm choices that matter most in clinical and epidemiological work, the diagnostic and validation machinery, and the interpretability tools that have made black-box models more defensible than they used to be.

11.3. The statistician's contribution

LLMs can fit a gradient-boosted tree, run cross-validation, and produce a calibration plot. They cannot decide whether prediction or inference is the right framing, whether the deployed model will actually improve the decision it informs, or whether the data resemble those in which the model will be used.

(Judgement 1.) Prediction is not inference. A model that achieves high AUC on a held-out set is a good predictor. It is *not* a tool for asking which variables cause the outcome, what the conditional effect of treatment is, or how a policy intervention would change outcomes. Conflating prediction performance with causal or counterfactual claims is the most common error in applied ML in medicine. The statistician's job is to keep this distinction visible, in the analysis, the report, and the conversation with collaborators.

(Judgement 2.) The held-out set is the deployment distribution, not the design distribution. Models trained on data from one hospital and evaluated on held-out cases from the same hospital are evaluated on the *training* distribution. Models that will be deployed to a different hospital, or to a future cohort, must be evaluated under that shift, temporal hold-out,

geographic hold-out, or prospective validation. The statistician identifies the deployment scenario and designs the evaluation to match.

(Judgement 3.) Calibration is the metric clinicians need. Discrimination, AUC, accuracy, tells you whether the model orders patients correctly. It does not tell you whether 'the model says 30%' means 30% of those patients will have the event. For decisions that require absolute risk thresholds (treat / do not treat above 10% predicted risk), calibration is the binding constraint, and it is not what tuning typically optimises. The statistician checks calibration explicitly and recalibrates if needed.

These judgements decide whether an ML pipeline produces a deployable tool or a benchmark on a private dataset.

11.4. The `tidymodels` workflow

`tidymodels` (Kuhn & Silge, 2022) is a meta-package covering the steps of a modelling pipeline:

- `rsample` for resampling (CV, bootstrap, time-based splits).
- `recipes` for preprocessing (centring, encoding, missing data handling).
- `parsnip` for model specification (uniform interface across `glmnet`, `ranger`, `xgboost`, `keras`, etc.).
- `tune` for hyperparameter tuning.
- `yardstick` for performance metrics.
- `workflows` for binding preprocessing + model into one object.

The goal is consistency: switching from logistic regression to gradient boosting changes one line. A typical pipeline:

```
library(tidymodels)

split <- initial_split(d, strata = outcome, prop = 0.75)
train <- training(split)
test <- testing(split)

rec <- recipe(outcome ~ ., data = train) |>
```

```
  step_rm(id) |>
  step_impute_median(all_numeric_predictors()) |>
  step_dummy(all_nominal_predictors()) |>
  step_normalize(all_numeric_predictors())

xgb_spec <- boost_tree(
  trees = 1000, tree_depth = tune(),
  learn_rate = tune(), min_n = tune()
) |>
  set_engine("xgboost") |>
  set_mode("classification")

wf <- workflow() |>
  add_recipe(rec) |>
  add_model(xgb_spec)

folds <- vfold_cv(train, v = 5, strata = outcome)
grid <- grid_latin_hypercube(
  tree_depth(), learn_rate(), min_n(),
  size = 30
)

tuned <- tune_grid(
  wf, resamples = folds, grid = grid,
  metrics = metric_set(roc_auc, brier_class)
)

best <- select_best(tuned, metric = "brier_class")
final_fit <- finalize_workflow(wf, best) |>
  last_fit(split, metrics = metric_set(
    roc_auc, brier_class, accuracy))
```

Two design choices to highlight. First, the metric used to select hyperparameters matters. Choosing on `roc_auc` optimises ranking; choosing on `brier_class` (or `mn_log_loss`) optimises probabilistic calibration. Choose the metric your downstream decision actually uses.

Second, `last_fit()` deliberately uses the test split *only* once, after tuning

is complete. The temptation to peek and re-tune is the single largest source of optimistic performance estimates in applied ML.

11.5. Algorithm choices that matter

For tabular biomedical data, four model families do most of the work:

Regularised generalised linear models (`glmnet`). The elastic net from chapter 9 fits cleanly into `tidymodels` via `parsnip::logistic_reg(penalty = tune(), mixture = tune())`. It is interpretable, well-calibrated by construction (with appropriate penalty), and the right baseline for any classification problem with sample sizes in the thousands or fewer.

Random forests (`ranger`). Robust, low-tuning, performant on tabular data with non-linear interactions. Less prone to overfitting than boosted trees but typically outperformed by them when tuning is done well.

Gradient-boosted trees (`xgboost`, `lightgbm`). The default for high-performance prediction on tabular data. `xgboost` (Chen & Guestrin, 2016) is the established choice; `lightgbm` is faster on very large datasets and handles categorical predictors more naturally. Both require careful tuning of `learn_rate`, `tree_depth`, `min_n`, and the number of trees; under-tuned, they overfit; over-tuned, they generalise poorly.

Neural networks (`torch`, `keras`). For tabular biomedical data with n in the thousands, neural networks rarely outperform gradient boosting and require considerably more engineering effort. They earn their place on high-dimensional sequence, image, or text data, clinical notes via transformers, medical imaging via convolutional networks, time series via recurrent or attention architectures. Outside those settings, prefer trees.

A practical decision rule:

Setting	Default model
$p < n$, interpretability matters	Regularised GLM
$p < n$, raw prediction matters	Gradient boosting
$p \approx n$ or $p > n$, sparse	Lasso (chapter 9)
Tabular with non-linear interactions	Gradient boosting
Imaging, text, sequence	Deep learning

Setting	Default model

Check your understanding: the right metric

Question. A clinical decision support tool will use a risk model output to display 'high', 'medium', or 'low' risk to clinicians. The thresholds are predicted probability above 30% (high) and below 10% (low). Should hyperparameters be tuned on AUC or on Brier score?
Answer.
Brier score (or log loss). The decision rule uses absolute predicted probabilities; calibration matters for the decision. AUC measures discrimination, does the model correctly *order* patients, and is invariant to monotonic transformations of the score. A model with excellent AUC can have terrible calibration: it ranks correctly but its predicted probabilities are systematically off. Tuning on AUC and then deploying with absolute-threshold decisions is a common error. Tune on Brier score; report AUC for context.

11.6. Validation: nested CV and external validation

Single-level cross-validation gives an optimistic estimate of out-of-sample performance when CV is also used for hyperparameter tuning, because the tuning has implicitly seen all of the data.

Nested CV corrects this: an inner CV loop chooses hyperparameters, an outer CV loop estimates performance. The outer loop never sees the tuning, so its performance estimate is unbiased.

```
nested <- nested_cv(
  train,
  outside = vfold_cv(v = 10),
  inside = vfold_cv(v = 5)
)
```

Nested CV is computationally expensive, 50 fits in the example above, plus a final fit on the full training set , but it is the correct estimator when

the question is 'what performance should we expect when we deploy this pipeline?'

For models intended for deployment, **external validation** on data the model has never seen is more important than any internal CV estimate. The relevant external set depends on the deployment scenario:

- **Temporal validation:** train on cases through year T, evaluate on year $T+1$. Catches drift in case mix, coding practices, and population.
- **Geographic validation:** train on hospital A, evaluate on hospital B. Catches site-specific feature distributions.
- **Prospective validation:** train on retrospective data, evaluate on prospectively collected cases under the actual deployment workflow. The gold standard for high-stakes deployments.

A model that drops 0.05 AUC from internal CV to external validation has a deployment problem; one that drops 0.15 has a fundamentally different distribution from the training set, and the analyst should investigate before deployment.

11.7. Calibration and recalibration

Calibration is the property that 'the model says 20%' means 20% of those cases have the event. It is checked by plotting predicted probability (binned) against observed event rate. A perfectly calibrated model lies on the diagonal.

```
library(probably)
cal_plot_breaks(test_with_preds,
                truth = outcome, estimate = .pred_yes,
                num_breaks = 10)
```

Common patterns:

- **Over-confident** model: predicts probabilities too close to 0 and 1; the curve is steeper than the diagonal. Common with deep learning models trained on small data.

- **Under-confident** model: predicts probabilities too close to the base rate; the curve is shallower than the diagonal. Common with heavy regularisation or with log-loss-suboptimal training.

Recalibration post-fits a one-dimensional function mapping raw scores to calibrated probabilities. Two methods:

- **Platt scaling**: fits a logistic regression of the outcome on the raw score.
- **Isotonic regression**: fits a non-decreasing function; more flexible but requires more recalibration data.

`probably::cal_apply()` integrates these into the `tidymodels` workflow. Recalibration is essential when deploying a model whose raw outputs do not match the deployment distribution; it is also a free improvement on most ML models, since the cost of training a calibration layer is trivial.

11.8. Interpretability: SHAP and beyond

For a deployed model, the relevant interpretability question is rarely 'which features are most important overall' (the model permutation importance answers that) but 'why did the model give *this* patient *this* prediction'. **SHAP values** (Lundberg & Lee, 2017): Shapley values from cooperative game theory, provide a principled per-prediction decomposition.

```
library(SHAPforxgboost)
shap_long <- shap.prep(xgb_model = final_xgb,
                       X_train = X_matrix)
shap.plot.summary(shap_long)
shap.plot.dependence(shap_long, x = "age")
```

SHAP values have two strong properties: they sum to the prediction (consistency), and they distribute credit fairly across features. They have one persistent limitation: they describe the *model's* dependence on a feature, not the world's. If the model has learned a spurious association, sex is associated with outcome because of unmeasured confounders, SHAP will faithfully describe that association as the model uses it. SHAP does not

provide causal inference, and a model interpreter who treats SHAP values as causal effects has crossed an important line.

The cleaner framing: SHAP explains *the model*, not the world. The model explains the data only to the extent that the analyst trusts the model. If a clinician asks 'why does the model predict high risk for this patient,' SHAP gives a defensible answer. If the same clinician asks 'so if we intervene on age, will risk drop,' SHAP cannot answer that question.

11.9. Worked example: 30-day readmission risk model

We extend the readmission example from chapter 7 to a machine-learning workflow with proper validation and calibration.

```
library(tidymodels)
library(probably)

set.seed(2026)
n <- 8000
d <- tibble(
  age = rnorm(n, 65, 12),
  cmi = rnorm(n, 1.5, 0.4),
  prior_admits = rpois(n, 0.6),
  los = rgamma(n, shape = 2, rate = 0.4),
  service = sample(c("med","surg","cards"), n,
                   replace = TRUE)
)
d$readmit <- factor(rbinom(
  n, 1,
  plogis(-2 + 0.02 * (d$age - 65) +
         0.5 * (d$cmi - 1.5) +
         0.3 * d$prior_admits +
         0.05 * d$los)
), labels = c("no","yes"))

split <- initial_split(d, strata = readmit, prop = 0.75)
train <- training(split); test <- testing(split)
```

```
rec <- recipe(readmit ~ ., data = train) |>
  step_dummy(all_nominal_predictors()) |>
  step_normalize(all_numeric_predictors())

xgb_spec <- boost_tree(
  trees = 500, tree_depth = tune(),
  learn_rate = tune(), min_n = tune()
) |> set_engine("xgboost") |>
     set_mode("classification")

wf <- workflow() |> add_recipe(rec) |> add_model(xgb_spec)

folds <- vfold_cv(train, v = 5, strata = readmit)
grid <- grid_latin_hypercube(
  tree_depth(range = c(2, 6)),
  learn_rate(range = c(-3, -1)),
  min_n(),
  size = 25
)

tuned <- tune_grid(
  wf, resamples = folds, grid = grid,
  metrics = metric_set(roc_auc, brier_class)
)

best <- select_best(tuned, metric = "brier_class")
final_fit <- finalize_workflow(wf, best) |>
  last_fit(split,
           metrics = metric_set(roc_auc, brier_class))

collect_metrics(final_fit)
```

The held-out AUC is around 0.74 and the Brier score around 0.16, competent for a heavily simulated example. Using `probably::cal_plot_breaks()` reveals a mild over-confidence at high predicted probabilities; a Platt recalibration narrows the gap.

A logistic regression on the same problem with elastic-net regularisation gives AUC around 0.73 and Brier around 0.16 , effectively indistinguishable. For this dataset, the gradient-boosted tree's flexibility yields no benefit over the regularised GLM. The simpler model is the right deployment.

This is the typical finding in applied biomedical ML: on tabular data with thousands of cases, regularised GLMs and gradient-boosted trees are within a percent of each other. The gradient-boosted tree wins when there are strong non-linear interactions; otherwise the simpler model wins on parsimony, calibration, and ease of explanation.

11.10. Collaborating with an LLM on machine learning

Three patterns dominate. LLMs handle the syntax of `tidymodels`, `xgboost`, and SHAP fluently. They are unreliable on the choice between prediction and inference, on validation strategy, and on the interpretation of model explanations.

Prompt 1: 'Build me a tidymodels pipeline for this binary outcome.' Provide the data and the outcome.

What to watch for. The LLM will produce a working pipeline. It will rarely ask whether the goal is prediction or inference. It will rarely propose external validation. It will default to AUC as the tuning metric even when absolute calibration is the binding constraint.

Verification. Before accepting the pipeline, articulate the decision the model will inform. If the decision is a threshold-based action ('treat above 10% predicted risk'), override the metric to Brier score and add a calibration plot. If the decision is to deploy at another site, plan external validation now, not after CV is complete.

Prompt 2: 'Why is the model giving this prediction for this patient?' Provide the patient's features and the model.

What to watch for. The LLM will produce a SHAP-style explanation: 'age contributes +0.4 to the log-odds, low CMI contributes -0.2,' etc. It will rarely flag that this explains the model's behaviour, not the underlying biology. It may slip into language that sounds causal: 'increasing age increases risk.'

Verification. The model explanation is a description of how the trained model uses the features in this case. It is not a causal effect. Translate any LLM explanation into explicit 'the model attributes' language before showing it to a clinician.

Prompt 3: 'Compare gradient boosting and logistic regression on this dataset.' Provide the data.

What to watch for. The LLM will fit both, report metrics, and pick the higher-AUC model. It will rarely test for significance of the difference, run nested CV to estimate generalisation honestly, or check calibration on the selected model. It will often miss that an apparent 0.005-AUC advantage is well within sampling noise.

Verification. Estimate uncertainty in the AUC difference via paired bootstrap on the test set. Run nested CV to confirm the advantage is not an artefact of single-level tuning. If the difference is small, report both models and let the parsimony argument decide.

The meta-pattern: LLMs are competent at the *mechanics* (pipeline construction, metric calculation, SHAP plots) and weak at the *adjudication* (does this model match the deployment scenario, is the apparent advantage real, what does the explanation actually mean). Treat the LLM as a fluent technician.

11.11. Principle in use

Three habits define defensible ML work:

1. **State the deployment scenario before tuning.** Who will use the model, where, on what distribution? Tune the validation strategy and the metric to match. A model evaluated on the same distribution it was trained on is a benchmark, not a deployable artefact.

2. **Calibrate before deployment.** Most ML models are poorly calibrated out of the box. Recalibration via Platt or isotonic on a held-out set is cheap insurance. Threshold-based decisions on uncalibrated scores are a common source of clinical-deployment failures.

3. **Distinguish model explanation from causal claim.** SHAP, permutation importance, and partial-dependence plots describe the model. They do not describe the world. Reports that conflate the two mislead clinicians and reviewers; reports that keep them separate are defensible.

11.12. Exercises

1. Repeat the readmission worked example with $n = 800$ instead of $n = 8000$. Compare the AUC and Brier improvements of the gradient-boosted tree over the logistic regression at small sample size.

2. Implement nested CV for the readmission pipeline. Compare the nested-CV AUC estimate to the simple train/test estimate. How large is the optimism?

3. Train the model on the first 70% of the data sorted by date, evaluate on the last 30%. Compare the temporal-validation AUC to the random-split AUC. Discuss any differences.

4. Apply Platt recalibration to a deliberately over-confident model (e.g., a deep tree with no regularisation). Plot calibration before and after.

5. Use SHAP on the readmission gradient-boosted tree. Identify a patient where the model's prediction is surprising (high predicted risk despite a low-risk profile). Investigate the SHAP attribution and discuss whether it would be defensible language to use with a clinician.

11.13. Further reading

- Kuhn & Silge (2022), the `tidymodels` book (https://www.tmwr.org/). The canonical applied reference.
- Hastie et al. (2009), *The Elements of Statistical Learning*. The textbook treatment of the underlying methods.
- Lundberg & Lee (2017), the SHAP paper. Best read alongside the `shap` documentation (https://shap.readthedocs.io/).

- Van Calster et al. (2019), *Calibration: the Achilles heel of predictive analytics*. The essential applied reference for clinical prediction model calibration.

Part V.

Software Engineering and Communication

12. Software Engineering for Statisticians

12.1. Learning objectives

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

- Profile R code with `profvis` and `bench` and identify the small set of bottlenecks worth optimising.
- Write performance-critical inner loops in C++ via `Rcpp` and integrate them into an R package.
- Build a defensible R package with `usethis` and `devtools`, including documentation, tests, continuous integration, and `pkgdown` documentation.
- Design a `testthat` test suite that covers correctness, numerical stability, and regression, and explain why each layer is needed.
- Use AI-assisted coding tools (LLM in IDE, Claude Code, GH Copilot) productively without ceding the engineering judgement that distinguishes maintainable code from syntactically correct code.
- Recognise the warning signs of unmaintainable research code and apply the small set of practices that prevent them.

12.2. Orientation

Statisticians produce code as a research artefact. Most of that code is throwaway: a one-off analysis, a simulation study, a figure script. Some of it becomes a library that collaborators use, a package on CRAN, or the computational core of a clinical decision tool. The skills that produce maintainable, performant, defensible software are not the same as the skills

that produce a clean Quarto report: and they are routinely under-taught in statistics curricula.

This chapter is a practical foundation in those skills. We cover profiling and performance, the C++/R interface through Rcpp, R-package authorship, testing strategy, and the working pattern of using AI assistance well. The running thread is that **the discipline of statistical software engineering is the same discipline as the rest of software engineering, with one difference: numerical correctness is harder to verify, and easier to compromise silently, than typical software correctness.**

12.3. The statistician's contribution

LLMs are competent at writing R code, structuring tests, and refactoring functions. They do not understand which parts of a numerical algorithm are correctness-critical, which optimisations preserve numerical equivalence, or when an apparent speedup hides a regression. The statistician owns these.

(Judgement 1.) Profile, then optimise. The most common engineering error in research code is optimising the wrong thing. A one-line change to the algorithm can recover 50× speedups; rewriting an irrelevant inner loop in C++ can buy 1.05× and three days of debugging. The statistician's role is to insist on profiling before optimising and to keep the engineering effort proportional to the bottleneck. LLMs are happy to rewrite anything in Rcpp; that does not mean they should.

(Judgement 2.) Numerical equivalence is not optional. A 'speedup' that changes outputs by 10^{-12} might be fine; one that changes outputs by 10^{-4} might be a broken algorithm. The statistician verifies that an optimised version produces *identical* results to the reference implementation on a battery of test cases: identical up to machine precision, with explicit tolerances. Skipping this check has produced silently wrong published results in the literature.

(Judgement 3.) Tests are the documentation that runs. Code without tests rots. Code with comprehensive tests can be refactored aggressively, ported to new platforms, and trusted by collaborators. The work of writing tests is also the work of figuring out the boundaries of the algorithm,

what inputs are valid, what edge cases must be handled, what numerical regime is supported. The statistician treats this as part of the work, not as overhead added at the end.

These judgements decide whether software is a research asset or a slowly accumulating liability.

12.4. Profiling: profvis and bench

R is famously slow at loops; it is also famously fast at vectorised operations on the right primitives. The question is rarely 'is R slow' but 'where is *my* code slow.' Profiling answers that.

profvis (Chang, Luraschi, et al., 2024) wraps R's sampling profiler in an interactive flame graph. The workflow:

```
library(profvis)
profvis({
  result <- run_my_analysis(data)
})
```

The output identifies which functions consume the most time and how much of that time is in their own bodies versus delegated to children. The pattern that recurs: 80% of the time is in 20% of the code, and within that 20%, a single line, a `for` loop over rows of a data frame, an unnecessary `rbind()` in a loop, a poorly configured `lm.fit()`, accounts for most of the budget.

bench (Hester, 2024) is the right tool for microbenchmarks. Comparing two implementations:

```
library(bench)
mark(
  base = sapply(x, sqrt),
  vec = sqrt(x),
  iterations = 100,
  check = TRUE
)
```

`check = TRUE` confirms the implementations produce equal results. This is not optional. Many proposed 'faster' implementations turn out to be both faster and wrong; the check catches them.

`bench::press()` sweeps a parameter and produces a performance-by-size table, the right tool when the question is 'how does runtime scale with n' rather than 'which is faster on this fixed problem.'

The two practical patterns that recover most R speedups:

1. **Vectorise.** Replace explicit `for` loops with element-wise operations on whole vectors or matrices.
2. **Avoid copy-on-modify in tight loops.** Pre-allocate output containers; use `data.table`'s in-place modification or `dplyr`'s reference semantics where helpful; never grow a vector with `c(x, new_value)` in a loop.

When neither applies, a genuinely sequential algorithm with state that depends on earlier iterations, the next move is C++.

12.5. Rcpp: the C++ escape hatch

`Rcpp` (Eddelbuettel & François, 2011) provides a clean interface between R and C++. The modal workflow:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector cumsum_cpp(NumericVector x) {
  int n = x.size();
  if (n == 0) return NumericVector(0);
  NumericVector out(n);
  out[0] = x[0];
  for (int i = 1; i < n; i++) {
    out[i] = out[i-1] + x[i];
  }
```

```
  return out;
}
```

`Rcpp::sourceCpp("cumsum.cpp")` compiles and exposes the function. For C++ code that lives inside an R package, `devtools::document()` regenerates the `RcppExports.cpp` shim and the C++ functions become callable from R like any other package function.

Rcpp is the right tool when:

- The algorithm is genuinely sequential (cannot be vectorised).
- The R version is order-of-magnitude slower than needed.
- The C++ version will be reused, justifying the engineering investment.

Rcpp is the wrong tool when the speedup is small, when an existing R-side primitive does the job, or when the correctness of the C++ version cannot be verified against a reference.

RcppArmadillo (Eddelbuettel & Sanderson, 2014) adds linear-algebra primitives via Armadillo; it is the standard for moving matrix-shaped algorithms into C++. **RcppEigen** does the same with Eigen and is preferred when interoperability with C++ libraries that use Eigen is needed.

Check your understanding: when not to use Rcpp

Question. A colleague has a function that loops over 1000 rows of a data frame, fits a small regression model to each, and stores the coefficient. The loop takes 30 seconds. They propose rewriting it in Rcpp. Is that the right next step?
Answer.
No. Rcpp would not help here: the bottleneck is the cost of fitting 1000 regression models, which lives in already-compiled C code (`lm.fit` or whichever solver). Rewriting the wrapping loop in Rcpp would buy nothing. The right next step is either to parallelise (chapter 8) or to consolidate the 1000 small fits into a single large fit using a per-row factor, a one-line `lm` call with a group factor produces the 1000 coefficients in milliseconds.

12.6. R-package authorship

A function that grows beyond a single script wants to be a package. The mechanics, with `usethis` and `devtools`, are mature enough that the engineering overhead is small.

```
usethis::create_package("~/projects/mypkg")
usethis::use_mit_license("Glenn Thomas")
usethis::use_testthat()
usethis::use_github_action_check_standard()
usethis::use_roxygen_md()
usethis::use_pkgdown()
```

A defensible package has, at minimum:

- **`DESCRIPTION`** declaring dependencies, license, and version.
- **`R/` source files** with `roxygen2` documentation on every exported function.
- **`tests/testthat/`** with tests for every exported function.
- **A `NEWS.md`** logging changes.
- **A `README.md`** with a minimal worked example.
- **CI configured** so every push runs `R CMD check` on multiple platforms.

The `usethis` calls above set all of this up. The remaining work is writing the code and the tests.

For packages that will be released widely:

- **CRAN release** through `devtools::release()`. The CRAN policies are strict; submission is iterative; allow time.
- **`pkgdown` site** built from `roxygen2` + vignettes, hosted on GitHub Pages. The de-facto standard for R package documentation; users expect it.
- **Versioning** follows semantic versioning conventions (`MAJOR.MINOR.PATCH`). Breaking changes increment `MAJOR`; new features increment `MINOR`; bug fixes increment `PATCH`.

12.7. Testing strategy

A typical research package needs three layers of tests.

Correctness tests verify the function returns the right answer on inputs with a known answer.

```
test_that("ols_fit recovers known coefficients", {
  set.seed(228)
  X <- cbind(1, rnorm(100))
  beta_true <- c(0.5, 1.2)
  y <- X %*% beta_true + rnorm(100, 0, 0.1)
  fit <- ols_fit(y, X)
  expect_equal(fit$coef, beta_true, tolerance = 0.05)
})
```

Numerical stability tests verify the function behaves correctly on inputs that stress the algorithm.

```
test_that("ols_fit handles near-collinear design", {
  X <- cbind(1, runif(100), runif(100) + 1e-8)
  expect_warning(ols_fit(rnorm(100), X), "collinear")
})
```

Regression tests verify that a known input produces a specific known output, byte-for-byte. They catch silent behaviour changes from refactors.

```
test_that("ols_fit output has not changed", {
  expect_snapshot_value(
    ols_fit(testdata$y, testdata$X)$coef,
    style = "deparse"
  )
})
```

`testthat::expect_snapshot()` and `expect_snapshot_value()` (Wickham, 2024) make regression tests inexpensive: the first run records the output; subsequent runs compare.

For packages that wrap numerical algorithms, a fourth layer is **reference-implementation tests**: the package output is compared to a slower but trusted reference (e.g., the closed-form solution, or the output from a well-established package). These are the tests that catch subtle algorithmic errors that pass the correctness and numerical-stability tests but produce wrong answers in specific regimes.

12.8. Working with AI assistants

Coding assistants, Copilot, Cursor, Claude Code, or similar, are now standard tools. Used well, they speed up boilerplate, accelerate refactors, and surface documentation. Used badly, they produce subtly wrong code that passes the eye test and fails in production.

Three patterns of productive use:

Boilerplate. Roxygen documentation skeletons, package file templates, ggplot themes, recipe steps in `tidymodels`. The LLM is producing code that any sufficiently experienced developer would write the same way; the verification cost is low because the code is boilerplate.

Refactor proposals. 'Rewrite this function to avoid the nested loop' is a request the LLM does well, *if* you verify equivalence. The verification cost is moderate (usually a `bench::mark()` with `check = TRUE` and a testthat run); the time saved is substantial.

Documentation explanations. 'What does this `dplyr` expression do' or 'why does my Stan model produce this warning' often gets a useful answer the LLM can produce from internalised documentation. Lower stakes; the verification is reading the result and confirming it matches your understanding.

Three patterns to avoid:

Numerical-algorithm authorship without verification. Asking the LLM to 'implement the L-BFGS update' or 'write a sparse Cholesky' is asking for code that is plausibly correct but very likely wrong in the corners. Use existing libraries (`optim`, `Matrix`, `RcppEigen`); reach for an LLM-authored numerical kernel only when no library exists, and verify exhaustively against a reference.

Test-skipping. LLMs will gladly remove tests that fail to pass after a refactor. The right move is the opposite: if a test fails, the test is your friend; investigate the behaviour change. An LLM that suggests removing a test is solving the wrong problem.

Wholesale generation of unfamiliar code. Asking the LLM for 800 lines of code in a domain you have never worked in is asking to deploy code you cannot review. The defence is to break the work into small enough pieces that you can review each.

The framing that has emerged: **the LLM is a fast, fluent, tireless junior collaborator.** It will type faster than you and produce more lines of code per hour. It will not catch the numerical edge cases, judge the right level of abstraction, or know what to test. Those remain yours.

12.9. Worked example: optimising a sequential algorithm

We illustrate the profile → optimise → verify cycle on a sequential filter, a one-step Kalman filter for a state-space model, that starts as a slow R loop.

```
kalman_r <- function(y, x0, P0, A, H, Q, R) {
  n <- length(y)
  x_hat <- numeric(n)
  P <- numeric(n)
  x_pred <- x0; P_pred <- P0
  for (t in seq_len(n)) {
    K <- P_pred * H / (H * P_pred * H + R)
    x_hat[t] <- x_pred + K * (y[t] - H * x_pred)
    P[t] <- (1 - K * H) * P_pred
    x_pred <- A * x_hat[t]
    P_pred <- A * P[t] * A + Q
  }
  list(x = x_hat, P = P)
}
```

Profile on a 100,000-step series:

```
library(profvis)
y <- rnorm(1e5)
profvis({
  out <- kalman_r(y, 0, 1, 1, 1, 0.1, 0.5)
})
# 1.4 seconds; 95% in the for loop body
```

Vectorisation is not available, each iteration depends on the previous. The remaining options are Rcpp or accepting the runtime. The Rcpp version:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
List kalman_cpp(NumericVector y, double x0, double P0,
                double A, double H, double Q, double R) {
  int n = y.size();
  NumericVector x_hat(n), P(n);
  double x_pred = x0, P_pred = P0;
  for (int t = 0; t < n; t++) {
    double K = P_pred * H / (H * P_pred * H + R);
    x_hat[t] = x_pred + K * (y[t] - H * x_pred);
    P[t] = (1.0 - K * H) * P_pred;
    x_pred = A * x_hat[t];
    P_pred = A * P[t] * A + Q;
  }
  return List::create(_["x"] = x_hat, _["P"] = P);
}
```

Benchmark with verification:

```
library(bench)
mark(
  R = kalman_r(y, 0, 1, 1, 1, 0.1, 0.5),
  cpp = kalman_cpp(y, 0, 1, 1, 1, 0.1, 0.5),
```

```
  iterations = 50, check = TRUE
)
# R    1.4 s
# cpp  3.2 ms
# check passed: outputs identical
```

The C++ version is about 400× faster and exactly identical in output (the implementations use the same algorithm with the same arithmetic order). The `check = TRUE` is the non-negotiable part: an Rcpp implementation that is fast but produces different output is a regression that must be caught at compile time, not in production.

The rule of thumb that emerges: **when the loop is genuinely sequential and dominates runtime, Rcpp is worth it; verify equivalence with `bench::mark(check = TRUE)` or a snapshot test.**

12.10. Collaborating with an LLM on software engineering

Three patterns dominate. LLMs are productive on boilerplate, refactors, and documentation. They are unreliable on performance reasoning, on numerical correctness in custom algorithms, and on test design.

Prompt 1: 'Refactor this 200-line function for clarity.' Provide the function and any test cases.

What to watch for. The LLM will produce a refactored version with smaller helpers and clearer naming. It may quietly change behaviour, a default argument, a sort order, an edge-case branch. It rarely confirms by running the test suite.

Verification. Run the test suite before and after. Diff the output of the function on a representative input before and after. Either a passing test suite or a byte-identical output diff is acceptable evidence; both are better.

Prompt 2: 'Write tests for this exported function.' Provide the function.

What to watch for. The LLM will produce a small set of correctness tests on happy-path inputs. It will rarely generate edge-case tests (empty inputs, NA values, boundary values) without prompting. It will not write regression tests against the current behaviour because it does not know what the current behaviour produces.

Verification. Use the LLM-generated tests as a starting point. Add edge-case tests yourself: empty input, NA input, single-element input, very large input. Add a snapshot regression test on a representative case. Then mutation-test: deliberately break the function and check that at least one test fails.

Prompt 3: 'Optimise this slow function.' Provide the function and the profiling output.

What to watch for. The LLM will propose vectorisation, preallocation, or Rcpp. It will sometimes propose all three at once. It will rarely run a benchmark to confirm the proposed optimisation actually helps; it is happy to ship a 'faster' version that is the same speed or slower.

Verification. `bench::mark(check = TRUE)` against the original. If the optimised version is no faster, revert. If it is faster but `check = FALSE`, you have a correctness regression masquerading as a speedup; revert and investigate.

The meta-pattern: LLMs accelerate the keystrokes; they do not accelerate the engineering. Treat the LLM as a fast typist with no judgement; supply the judgement yourself.

12.11. Principle in use

Three habits define defensible software engineering work:

1. **Profile before optimising; verify after.** Optimisation work without a profile is guesswork. Optimisation results without a verification of numerical equivalence are claims, not facts. Both habits are inexpensive; both are routinely skipped.

2. **Treat the test suite as the spec.** Write tests as you write the function, not after. The tests are how you discover the boundary cases the function must handle; they are also how you prove the function works. A function without tests is undocumented and un-refactorable.

3. **Match engineering effort to artefact lifespan.** A one-off analysis script does not need a package structure or CI. A library that collaborators will use does. Apply the right level of engineering to the right artefact; under-engineering one and over-engineering the other are equally costly.

12.12. Exercises

1. Take a research script of your own and profile it with `profvis`. Identify the slowest function. Re-profile after vectorising it. Document the speedup.

2. Implement a sequential algorithm of your choice (e.g., the Viterbi algorithm for HMMs) in pure R and in Rcpp. Benchmark with `check = TRUE`. Document the speedup and the threshold problem size at which the Rcpp version becomes worth using.

3. Convert a research script into a package using `usethis::create_package()`. Add at least one exported function with `roxygen2` documentation, three testthat tests (correctness, edge case, snapshot), and a CI workflow with `usethis::use_github_action_check_standard()`.

4. Write a regression test against a non-trivial function in your codebase using `expect_snapshot_value()`. Modify the function to produce a different output; confirm the test fails. Restore the function; confirm the test passes.

5. Pick a function written entirely by an LLM. Write a property-based test (e.g., 'the function is monotone in its first argument') without consulting the LLM. Did the LLM-authored function pass?

12.13. Further reading

- Wickham & Bryan (2023), *R Packages*, 2nd edition (https://r-pkgs.org/). The canonical reference for R-package authorship.
- Eddelbuettel (2013), *Seamless R and C++ Integration with Rcpp*. The book-length treatment of the C++/R interface.
- Wickham (2024), the `testthat` documentation and the chapter in *R Packages*.
- The `profvis` and `bench` package vignettes are exemplary applied references.

13. Advanced Interactive Visualisation and Dashboards

13.1. Learning objectives

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

- Use `plotly` and `htmlwidgets` to convert static visualisations into interactive web graphics, and decide when interactivity adds value versus when it adds distraction.
- Architect a `shiny` application that scales beyond a single user and a single session, using `bslib`, `shinyloadtest`, and reactive-graph discipline.
- Build dashboards with `flexdashboard`, Quarto dashboards, and `shinydashboard`/`bslib::page_navbar()`, and choose between them based on the audience and update cadence.
- Use Observable JS and `ojs-define` blocks in Quarto to embed reactive analytics in static documents.
- Apply principles of effective dashboard design, hierarchy of information, calibrated colour, semantic accessibility, to an analytics dashboard for a clinical audience.

13.2. Orientation

A static plot answers a question. A dashboard mediates a conversation: the user asks something the designer anticipated, the dashboard responds, the user asks the follow-up. The shift from artefact to conversation changes the engineering and the design.

This chapter is about the engineering. The design half: how to compose a dashboard that supports decision-making rather than confusing the user, gets a section here but is the subject of book-length treatments (Few, 2013; Munzner, 2014). We focus on the toolkit: `plotly` for interactive graphics, `shiny` for reactive applications, dashboard frameworks for at-a-glance displays, and Observable JS for in-document reactivity.

The framing throughout: **interactivity is a tool, not a goal.** Most analytic questions are better answered by a well-designed static figure than by a dashboard. The case for interactivity should be made, and the case for static should be the default, every time.

13.3. The statistician's contribution

LLMs can write `shiny` code, configure `plotly` tooltips, and lay out dashboards. They cannot decide which interactions a clinician actually wants, what default views inform decisions, or whether the dashboard is delivering information or merely flattering the user with graphics.

(Judgement 1.) Interactivity that does not help is distraction. Users routinely ask for filters and toggles that they will not use. Building them costs engineering time, complicates testing, slows page load, and obscures the data behind layers of UI. The statistician's role is to push back on requested interactions until the user can articulate what decision the interaction informs. 'I want to filter by year' is not yet a decision; 'I want to see this year's quality measure trends to decide where to intervene' is.

(Judgement 2.) Defaults are the dashboard. Most users look at a dashboard's default state and never change it. The default view, default filters, default time range, and default sort order *are* the dashboard for those users. Building elaborate alternative views that no one will discover is wasted work. The statistician decides what the default state should communicate; the secondary views are optional refinements.

(Judgement 3.) Calibration applies to dashboards. Numbers reported on dashboards are taken at face value. A dashboard that displays predicted readmission risk to clinicians is making a calibration claim by displaying the number. If the underlying model is poorly calibrated, the dashboard is misleading, and 'we just display the model output' is not a

defence. The statistician is responsible for the accuracy of every quantity the dashboard shows, including the ones the user did not ask about.

These judgements decide whether a dashboard is a useful tool or an attractive façade.

13.4. Interactive plots: plotly and htmlwidgets

The shortest path from a ggplot2 figure to an interactive version is plotly::ggplotly():

```
library(ggplot2)
library(plotly)

p <- ggplot(d, aes(x = age, y = los, colour = service)) +
  geom_point(alpha = 0.4) +
  facet_wrap(~ quarter)

ggplotly(p)
```

ggplotly() translates the ggplot to a Plotly graph, preserving most aesthetics and adding hover tooltips, zoom, pan, and a legend toggle. For most exploratory work this is enough.

For dashboard or report contexts that need more control: custom hover content, click events, animation, Plotly's native syntax via plot_ly() is preferable. The native syntax is also faster and produces smaller HTML output for large data.

Other htmlwidgets-based packages cover specialised needs:

- **leaflet** for interactive maps.
- **DT** for searchable, sortable, paginated tables: the de-facto standard for data tables in dashboards.
- **reactable** for tables with conditional formatting and inline interactivity, more flexible than DT for custom layouts.
- **networkD3** / **visNetwork** for graph and network visualisation.

- **echarts4r** for richer chart types than **plotly** (gauges, sankey, treemaps).

A practical performance note: **htmlwidgets** embed all data in the rendered HTML by default. A dashboard with 100,000 points displayed in **plotly** produces a large HTML file and can be slow to render. For large data either subsample before rendering or use server-side rendering via **shiny::renderPlotly()**.

13.5. Reactive applications with shiny

shiny (Chang, Cheng, et al., 2024) is the workhorse for serious analytic applications in R. The architecture is reactive: the UI declares inputs and outputs, the server defines how outputs depend on inputs, and **shiny** wires them together.

A minimum viable application:

```
library(shiny)
library(bslib)

ui <- page_sidebar(
  title = "Cohort explorer",
  sidebar = sidebar(
    selectInput("service", "Service",
                choices = c("med", "surg", "cards")),
    sliderInput("year", "Year", 2018, 2024,
                value = c(2022, 2024), sep = "")
  ),
  card(card_header("Outcomes by service"),
       plotOutput("plot")),
  card(card_header("Cohort summary"),
       tableOutput("summary"))
)

server <- function(input, output, session) {
  filtered <- reactive({
```

```
    d |>
      filter(service == input$service,
             year >= input$year[1],
             year <= input$year[2])
  })
  output$plot <- renderPlot({
    ggplot(filtered(), aes(x = los, y = readmit)) +
      geom_smooth(method = "lm") + geom_point()
  })
  output$summary <- renderTable({
    filtered() |>
      summarise(n = n(), mean_los = mean(los),
                readmit_rate = mean(readmit == "yes"))
  })
}

shinyApp(ui, server)
```

`bslib` (Aden-Buie et al., 2024) is the modern UI framework for `shiny`, replacing the older `shinydashboard` look with a Bootstrap 5 design system that supports themeing, responsive layouts, and dark mode. Most new applications should start with `bslib`.

13.5.1. Reactive-graph discipline

The most common cause of slow `shiny` applications is unintended re-execution. A `reactive({ ... })` runs once per change to its inputs and caches the result; a `renderPlot({ ... })` runs once per change to *any* reactive it reads. Without discipline, a single input change can trigger a cascade of re-renders.

Three rules:

1. **Compute once, use many times.** If two outputs depend on the same filtered data, compute the filter in a `reactive()` and read it from both. Do not re-filter in each `renderX()`.

2. **Defer expensive operations.** `bindCache()` caches reactive computations across sessions; `bindEvent()` and `eventReactive()` decouple expensive computations from input changes that should not trigger them.

3. **Respect the `req()` boundary.** A reactive that uses `req(input$x)` halts the reactive chain if `input$x` is `NULL`. This prevents unnecessary computation during initial render.

For applications with many users, the question is not just 'does it work' but 'does it scale.' `shinyloadtest` (Dipert, 2024) simulates concurrent sessions and measures response times. The results often reveal that a single-server application supports 10–20 concurrent users before degrading. Scaling beyond that requires either horizontal scaling (multiple R processes behind a load balancer, via `shinyapps.io`, Posit Connect, or `shiny-server-pro`) or aggressive caching of the expensive operations.

Check your understanding: the right framework

Question. A collaborator asks for a dashboard displaying weekly clinical-quality metrics. The data update once a week, all users see the same view, no filtering or interaction is needed beyond viewing the plots. Should this be a `shiny` application?
Answer.
No. The use case is a static report, not a reactive application. A Quarto document parameterised by week, rendered to HTML and republished weekly via cron or GitHub Actions, is the right tool. It is simpler, cheaper to host, faster to load, and accessible without server infrastructure. `shiny` adds value only when the user interacts; if the only 'interaction' is viewing a fixed display, the static report is the better choice.

13.6. Dashboard frameworks

Three dashboard frameworks dominate in 2026:

Quarto dashboards (Posit Software, PBC, 2024), added in Quarto 1.4, let you author a multi-pane dashboard in Markdown with code chunks.

They render to a static HTML file with no server required. The right tool when the data updates on a schedule and the dashboard does not need user-driven filtering.

`flexdashboard` is the older, still-supported predecessor with a similar mental model and slightly different layout primitives. New work should prefer Quarto dashboards.

`shiny` dashboards via `bslib::page_navbar()` or `bs4Dash` are the choice when reactive interactivity is required. They are heavier, they need a server, but they support the full reactive programming model.

The decision tree:

Need	Tool
Static, scheduled refresh	Quarto dashboard
Reactive filtering, single user	`shiny` + `bslib`
Reactive, many concurrent users	`shiny` + caching + horizontal scaling
Embedded reactive in a report	Observable JS in Quarto

13.7. Observable JS in Quarto documents

Observable JS (OJS) is JavaScript-based reactive notebook syntax adapted for Quarto. It provides reactive interactivity without a server: the computation runs in the user's browser. The trade-off: OJS reactives must be written in JavaScript (or in R/Python with data passed via `ojs_define()`), and the data must be small enough to ship in the rendered HTML.

The pattern:

````
```{r}
ojs_define(d = readmissions)
```


```{ojs}
viewof service = Inputs.select(
  ["med", "surg", "cards"],
````
````

```
  {label: "Service"}
)

filtered = d.filter(r => r.service == service)

Plot.plot({
  marks: [
    Plot.dot(filtered, {x: "los", y: "readmit_prob"})
  ]
})
```

(The double-braced `{r}` and `{ojs}` are Quarto's escape syntax for showing chunk-syntax literally inside another code block; in your real document, write single braces.)

OJS is the right tool when:

- The data is small enough to embed (typically under a few MB).
- The reactivity is genuinely client-side, filter, sort, summarise, plot, and does not require server computation.
- The deliverable is a self-contained HTML file that works without a server.

For large datasets, server computation, or full reactive applications, `shiny` remains the answer.

13.8. Worked example: a clinical quality dashboard

A monthly-refreshed dashboard displaying readmission rates, length of stay, and patient satisfaction by service line, built as a Quarto dashboard. The data refreshes on the first of each month via a cron-driven render.

```
---
title: "Quality metrics: April 2026"
format: dashboard
---
```

````
```{r}
#| include: false
library(tidyverse); library(plotly); library(DT)
d <- readRDS("data/quality_metrics_apr2026.rds")
```

# Overview

## Row {height=20%}

```{r}
#| content: valuebox
list(
  icon = "hospital",
  color = "primary",
  value = nrow(d),
  title = "Discharges (month)"
)
```

```{r}
#| content: valuebox
list(
  icon = "arrow-repeat",
  color = if (mean(d$readmit) > 0.12) "danger"
          else "success",
  value = sprintf("%.1f%%", 100 * mean(d$readmit)),
  title = "Readmission rate"
)
```

## Row {height=80%}

### Column

```{r}
p <- d |>
````
````

```
  group_by(service, month) |>
  summarise(rate = mean(readmit), .groups = "drop") |>
  ggplot(aes(month, rate, colour = service)) +
  geom_line() + geom_point() +
  labs(y = "Readmission rate", x = NULL)
ggplotly(p)
```

Column

```{r}
d |>
  group_by(service) |>
  summarise(n = n(), mean_los = mean(los),
            readmit = mean(readmit)) |>
  datatable(options = list(dom = "t"),
            rownames = FALSE)
```

Data quality

Row

(...further pages with data-quality checks)
```

The `valuebox` cells communicate the headline numbers in the way clinicians scan first. The line chart provides trend context. The data table at the right provides drill detail. The conditional colour on the readmission valuebox (red above 12%, green below) is a deliberate decision: the dashboard is communicating not only the value but a judgement about it. That judgement should be reviewed by clinical stakeholders, not invented by the analyst.

The dashboard renders to a self-contained HTML file. A GitHub Actions workflow runs `quarto render` against fresh data on the first of every month and publishes the result. No `shiny` server is needed; the cost is near zero; the output is fast to load and trivially shareable.
```

13.9. Collaborating with an LLM on interactive visualisation

Three patterns dominate. LLMs handle `shiny` and Quarto dashboard syntax fluently. They are weaker on dashboard design judgement and on performance reasoning.

Prompt 1: 'Convert this ggplot to plotly with hover tooltips showing patient ID and discharge date.'

What to watch for. The LLM will produce working code quickly. It may default to `ggplotly()` even when `plot_ly()` would render faster on the data size. It may embed sensitive identifiers (patient ID) in the rendered HTML; if the dashboard is shared, this is a privacy violation.

Verification. Inspect the rendered HTML for any identifying information that should not be shared. Check rendering time on representative data; if it is slow, switch to `plot_ly()` or aggregate the data before plotting.

Prompt 2: 'Add a filter for service line and a date range to this shiny app.'

What to watch for. The LLM will produce working reactive code. It may not factor the filter into a single `reactive()`, leading to redundant computation. It may not add `req(input$service)` to handle initial state. It will rarely test the application under any meaningful load.

Verification. Read the resulting reactive graph in `reactlog`. Confirm filters compute once, not in each `renderX()`. Run `shinyloadtest` on a target concurrency level before deploying to a multi-user environment.

Prompt 3: 'Design a dashboard for clinical leadership to monitor readmissions.'

What to watch for. The LLM will produce a layout: typically with too many panels and too many chart types. It will not ask what decisions clinical leadership makes or what view should be the default. It is happy to populate a dashboard with everything available rather than the few quantities that matter.

Verification. Before implementing, articulate the two or three decisions the dashboard supports. Build for those. Treat additional panels as optional.

The dashboard that fits on a single screen with three carefully chosen panels is almost always more useful than the eight-panel sprawl the LLM produces.

The meta-pattern: LLMs accelerate the building of UI; they do not improve the design of UI. Treat the LLM as fast hands; bring your own taste and your own contact with the end user.

13.10. Principle in use

Three habits define defensible interactive-visualisation work:

1. **Default to static.** A static figure or a static parameterised report should be the working assumption. Interactivity is a feature you add when you can name the decision it informs, not a default.

2. **Engineer for the default state.** Most users look at the dashboard you ship without changing anything. The default view, default filters, and default sort are the dashboard. Make them informative.

3. **Test under load before deploying.** A `shiny` application that works on your laptop may collapse under five concurrent users. Run `shinyloadtest` before deploying to a multi-user setting; cache and scale the bottlenecks before users find them.

13.11. Exercises

1. Take a static `ggplot2` visualisation from a previous chapter. Convert it to `plotly`, then to OJS. Compare the file sizes, load times, and user experience for a colleague who has not seen the figure before.

2. Build a `shiny` application with a single filter and two dependent outputs. Implement it once with the filter inside both `renderX` calls and once with the filter in a shared `reactive()`. Use `reactlog` to confirm the difference in re-execution.

3. Convert a quarterly report you produce as a static document to a Quarto dashboard. Identify which elements gain from being on a dashboard and which lose from the constraint of a single screen.
4. Run `shinyloadtest` against a `shiny` application of your choice with simulated concurrency of 10, 50, and

    100. Document the response-time degradation. Implement `bindCache()` on the slowest reactive and re-test.

5. Critique a clinical or epidemiological dashboard you have seen in production. Identify three design choices that aid decision-making and three that obscure it. Propose specific changes.

13.12. Further reading

- Wickham (2021), *Mastering Shiny* (https://mastering-shiny.org/). The canonical applied reference for `shiny`.
- Munzner (2014), *Visualization Analysis & Design*. The textbook treatment of visualisation principles, applied broadly.
- Few (2013), *Information Dashboard Design*. The applied reference for dashboard design specifically.
- The Quarto dashboard documentation (https://quarto.org/docs/dashboards/) and the Observable JS guide (https://quarto.org/docs/interactive/ojs/) are exemplary practical references.

References

Aden-Buie, G., Sievert, C., Cheng, J., & Schloerke, B. (2024). *bslib: Custom Bootstrap Sass themes for shiny and rmarkdown*. https://rstudio.github.io/bslib/

Bates, D., & Maechler, M. (2010). Matrix: Sparse and dense matrix classes and methods. *R Package*. https://cran.r-project.org/package=Matrix

Beaumont, M. A. (2010). Approximate Bayesian computation in evolution and ecology. *Annual Review of Ecology, Evolution, and Systematics*, *41*, 379–406.

Bengtsson, H. (2021). A unifying framework for parallel and distributed processing in R using futures. *The R Journal*, *13*(2), 208–227.

Betancourt, M. (2017). A conceptual introduction to Hamiltonian Monte Carlo. *arXiv:1701.02434*.

Booth, J. G., & Hobert, J. P. (1999). Maximizing generalized linear mixed model likelihoods with an automated Monte Carlo EM algorithm. *Journal of the Royal Statistical Society, Series B*, *61*(1), 265–285.

Boyd, S., Parikh, N., Chu, E., Peleato, B., & Eckstein, J. (2011). Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, *3*(1), 1–122.

Boyd, S., & Vandenberghe, L. (2004). *Convex optimization*. Cambridge University Press. https://web.stanford.edu/~boyd/cvxbook

Bürkner, P.-C. (2017). brms: An R package for Bayesian multilevel models using Stan. In *Journal of Statistical Software* (No. 1; Vol. 80).

Candès, E., Fan, Y., Janson, L., & Lv, J. (2018). Panning for gold: Model-X knockoffs for high dimensional controlled variable selection. *Journal of the Royal Statistical Society, Series B*, *80*(3), 551–577.

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., & Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of Statistical Software*, *76*(1).

Chang, W., Cheng, J., Allaire, J., Sievert, C., Schloerke, B., Xie, Y., Allen, J., McPherson, J., Dipert, A., & Borges, B. (2024). *shiny: Web application framework for R*. https://shiny.posit.co/

Chang, W., Luraschi, J., & Mastny, T. (2024). *profvis: Interactive visualizations for profiling R code*. https://rstudio.github.io/profvis/

Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 785–794.

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B*, *39*(1), 1–38.

Dipert, A. (2024). *shinyloadtest: Load test Shiny applications*. https://rstudio.github.io/shinyloadtest/

Eddelbuettel, D. (2013). *Seamless R and C++ integration with Rcpp*. Springer.

Eddelbuettel, D., & François, R. (2011). Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, *40*(8), 1–18.

Eddelbuettel, D., & Sanderson, C. (2014). RcppArmadillo: Accelerating R with high-performance C++ linear algebra. *Computational Statistics and Data Analysis*, *71*, 1054–1063.

El Ghaoui, L., Viallon, V., & Rabbani, T. (2012). Safe feature elimination for the LASSO and sparse supervised learning problems. *Pacific Journal of Optimization*, *8*(4), 667–698.

Fan, J., & Li, R. (2001). Variable selection via nonconcave penalized likelihood and its oracle properties. *Journal of the American Statistical Association*, *96*(456), 1348–1360.

Few, S. (2013). *Information dashboard design: Displaying data for at-a-glance monitoring* (2nd ed.). Analytics Press.

Ge, H., Xu, K., & Ghahramani, Z. (2018). Turing: A language for flexible probabilistic inference. *AISTATS*.

Geer, S. van de, Bühlmann, P., Ritov, Y., & Dezeure, R. (2014). On asymptotically optimal confidence regions and tests for high-dimensional models. *Annals of Statistics*, *42*(3), 1166–1202.

Gelman, A., Vehtari, A., Simpson, D., Margossian, C. C., Carpenter, B., Yao, Y., Kennedy, L., Gabry, J., Bürkner, P.-C., & Modrák, M. (2020). Bayesian workflow. *arXiv:2011.01808*.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, *23*(1), 5–48.

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press.

Goodrich, B., Gabry, J., Ali, I., & Brilleman, S. (2024). *rstanarm: Bayesian applied regression modeling via Stan*. https://mc-stan.org/rstanarm/

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning* (2nd ed.). Springer.

Hastie, T., Tibshirani, R., & Wainwright, M. (2015). *Statistical learning with sparsity: The Lasso and generalizations*. Chapman; Hall/CRC.

Hester, J. (2024). *bench: High precision timing of R expressions*. https://bench.r-lib.org/

Higham, N. J. (2002). *Accuracy and stability of numerical algorithms* (2nd ed.). SIAM.

Hoffman, M. D., & Gelman, A. (2014). The No-U-Turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, *15*, 1593–1623.

Kucukelbir, A., Tran, D., Ranganath, R., Gelman, A., & Blei, D. M. (2017). Automatic differentiation variational inference. *Journal of Machine Learning Research*, *18*(14), 1–45.

Kuhn, M., & Silge, J. (2022). *Tidy modeling with R*. O'Reilly Media. https://www.tmwr.org

Lee, J. D., Sun, D. L., Sun, Y., & Taylor, J. E. (2016). Exact post-selection inference, with application to the lasso. *Annals of Statistics*, *44*(3), 907–927.

Lundberg, S. M., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. *Advances in Neural Information Processing Systems (NeurIPS)*, 4765–4774.

McElreath, R. (2020). *Statistical rethinking: A Bayesian course with examples in R and Stan* (2nd ed.). Chapman; Hall/CRC.

McLachlan, G. J., & Krishnan, T. (2008). *The EM algorithm and extensions* (2nd ed.). Wiley.

Meinshausen, N., & Bühlmann, P. (2010). Stability selection. *Journal of the Royal Statistical Society, Series B*, *72*(4), 417–473.

Meinshausen, N., Meier, L., & Bühlmann, P. (2009). P-values for high-dimensional regression. *Journal of the American Statistical Association*, *104*(488), 1671–1681.

Meng, X.-L., & Rubin, D. B. (1993). Maximum likelihood estimation via the ECM algorithm: A general framework. *Biometrika*, *80*(2), 267–278.

Munzner, T. (2014). *Visualization analysis and design*. CRC Press.

Nocedal, J., & Wright, S. J. (2006). *Numerical optimization* (2nd ed.). Springer.

Owen, A. B. (2013). *Monte carlo theory, methods and examples.* https://artowen.su.domains/mc/

Parikh, N., & Boyd, S. (2014). Proximal algorithms. *Foundations and Trends in Optimization*, *1*(3), 127–239.

Posit Software, PBC. (2024). *Quarto dashboards.* https://quarto.org/docs/dashboards/

Raasveldt, M., & Mühleisen, H. (2019). DuckDB: An embeddable analytical database. *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, 1981–1984.

Robert, C. P., & Casella, G. (2004). *Monte carlo statistical methods* (2nd ed.). Springer.

Rue, H., Martino, S., & Chopin, N. (2009). Approximate Bayesian inference for latent Gaussian models by using integrated nested Laplace approximations. *Journal of the Royal Statistical Society, Series B*, *71*(2), 319–392.

Saad, Y. (2003). *Iterative methods for sparse linear systems* (2nd ed.). SIAM.

Salvatier, J., Wiecki, T. V., & Fonnesbeck, C. (2016). Probabilistic programming in Python using PyMC3. *PeerJ Computer Science*, *2*, e55.

Taylor, J., & Tibshirani, R. (2018). Post-selection inference for ℓ_1- penalized likelihood models. *Canadian Journal of Statistics*, *46*(1), 41–61.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society, Series B*, *58*(1), 267–288.

Tibshirani, R., Bien, J., Friedman, J., Hastie, T., Simon, N., Taylor, J., & Tibshirani, R. J. (2012). Strong rules for discarding predictors in lasso-type problems. *Journal of the Royal Statistical Society, Series B*, *74*(2), 245–266.

Trefethen, L. N., & Bau III, D. (1997). *Numerical linear algebra.* SIAM.

Van Calster, B., McLernon, D. J., Smeden, M. van, Wynants, L., Steyerberg, E. W., & Topic Group 'Evaluating diagnostic tests and prediction models' of the STRATOS initiative, on behalf of. (2019). Calibration: The Achilles heel of predictive analytics. *BMC Medicine*, *17*(1), 230.

Vehtari, A., Gelman, A., & Gabry, J. (2017). Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC. *Statistics and Computing*, *27*, 1413–1432.

Vehtari, A., Gelman, A., Simpson, D., Carpenter, B., & Bürkner, P.-C.

(2021). Rank-normalization, folding, and localization: An improved $\hat{R}$ for assessing convergence of MCMC. *Bayesian Analysis*, *16*(2), 667–718.

Wang, J., Wonka, P., & Ye, J. (2015). Lasso screening rules via dual polytope projection. *Journal of Machine Learning Research*, *16*, 1063–1101.

Watanabe, S. (2010). Asymptotic equivalence of Bayes cross validation and widely applicable information criterion in singular learning theory. *Journal of Machine Learning Research*, *11*, 3571–3594.

Wickham, H. (2021). *Mastering Shiny*. O'Reilly Media. https://mastering-shiny.org

Wickham, H. (2024). *testthat: Unit testing for R*. https://testthat.r-lib.org/

Wickham, H., & Bryan, J. (2023). *R packages* (2nd ed.). O'Reilly Media. https://r-pkgs.org

Yuan, M., & Lin, Y. (2006). Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society, Series B*, *68*(1), 49–67.

Zeng, Y., & Breheny, P. (2017). The biglasso package: A memory- and computation-efficient solver for lasso model fitting with big data in R. *arXiv:1701.05936*.

Zhang, C.-H. (2010). Nearly unbiased variable selection under minimax concave penalty. *Annals of Statistics*, *38*(2), 894–942.

Zhang, C.-H., & Zhang, S. S. (2014). Confidence intervals for low dimensional parameters in high dimensional linear models. *Journal of the Royal Statistical Society, Series B*, *76*(1), 217–242.

Zhang, L., Carpenter, B., Gelman, A., & Vehtari, A. (2022). Pathfinder: Parallel quasi-Newton variational inference. *Journal of Machine Learning Research*, *23*(306), 1–49.

Zou, H. (2006). The adaptive lasso and its oracle properties. *Journal of the American Statistical Association*, *101*(476), 1418–1429.

Zou, H., & Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society, Series B*, *67*(2), 301–320.

Credits

The chapter list was constructed by surveying 12 publicly-available syllabi from major US biostatistics programmes (documented in `docs/syllabi-survey.md`) and adopting the topics that appeared in three or more of them.

Cover artwork generated procedurally by the Python script at `images/build-cover.py`. Watercolour palette: deep wine through plum and rose to warm amber, anchored on the brand burgundy `#7a2c4e`. Typography: Avenir family (Apple system font); body text in Source Serif 4.

Colophon

This book was produced with Quarto, typeset in Source Serif 4, with code blocks set in JetBrains Mono. Source code is in R 4.4 or later.

The book is hosted at https://scai-advanced.rgtlab.org on Netlify, with continuous deployment via GitHub Actions from the `rgt47/scai-advanced` repository on every push to `main`. The deployment recipe is in `HOSTING.md`.

The cover is generated procedurally by `images/build-cover.py` from a watercolour gradient anchored on `#7a2c4e` (the volume's brand burgundy) and overlaid with Avenir typography. Re-running the script regenerates the cover; the procedural approach makes the cover reproducible and editable in version control rather than dependent on a one-shot AI image-generation step.

Last rendered: see the build timestamp on the homepage.